第10回
情報科学ワークショップ

The 10th Workshop on Theoretical Computer Science
Fukuyama, Hiroshima, September 2014
(WTCS2014)

中野　浩嗣
伊藤　靖朗
高藤　大介

主担当：広島大学

目次

序言

本論文集は 2014 年 9 月 17 日〜19 日において，広島県福山市 ツネイシしまなみビレッジにて開催された第 10 回情報科学ワークショップでの発表原稿をまとめたものである．

本ワークショップは，日本の並列／分散計算の研究者が研究室以上の議論と研究会以下のフォーマルさを目指し，合宿形式でお互いを徹底的に切りまくる「ちゃんばら大会」を目的とし，大阪大学，九州大学，九州工業大学，京都工芸繊維大学，名古屋工業大学，奈良先端科学技術大学院大学の研究者有志によって 2005 年に始まったものである．第 1 回に出雲で開催して以来，第 2 回の瀬戸，第 3 回の門司，第 4 回の長浜，第 5 回の広島， 第 6 回の桑名，第 7 回の福岡，第 8 回の神戸，第 9 回の唐津と毎年 9 月に開催し，今回は記念の 10 回目の開催となる．

今回は大阪大学，国立情報学研究所，九州大学，九州工業大学，佐賀大学，豊橋技術科学大学，名古屋工業大学，奈良先端科学技術大学院大学，広島大学，ブラジリア大学，法政大学から 64 人の参加者があり，46 件の発表が行われた．今回の開催場所は，福山市鞆の浦，尾道市，しまなみ海道といういずれの観光地にもほど近い山あいにあるツネイシしまなみビレッジであり，周辺には三世代テーマパーク「みろくの里」や神勝寺温泉もある．山あいの静かな環境の中で夜遅くまで活発な討論が行われ，有意義な時間を共有できた．なお，2015 年度は名古屋工業大学が開催予定である．

最後に，今回の開催にあたりご協力をいただいた皆様，ワークショップ運営にご協力をいただいた参加者の皆様に厚く御礼申し上げる．

2015 年 1 月

中野 浩嗣
伊藤 靖朗
高藤 大介

第10回情報科学ワークショップ　プログラム

2014年9月17日（水）～19日（金） ツネイシしまなみビレッジ（〒720-0402　広島県福山市沼隈町中山南26-1）

9月17日

12:20 開会

セッションA：自律分散1(12:30-14:05)				座長：亀井 清華	
A-1	12:30	貝野 太地	名古屋工業大学	On the Worst-Case Initial Configuration for Conservative Connectivity Preservation	ミドル
A-2	12:50	寺井 智史	法政大学	状態をもつ自律分散ロボット群の能力について	ミドル
A-3	13:10	山内 由紀子	九州大学	Randomized Pattern Formation Algorithm for Autonomous Mobile Robots	ミドル
A-4	13:30	Anissa Lamani	九州大学	Oscillatory Population Protocols	ミドル
A-5	13:50	八神 貴裕	九州大学	オンライン木探索の最適性について	ショート
セッションB：自律分散2(14:15-15:55)				座長：山内 由紀子	
B-1	14:15	平野 拓弥	名古屋工業大学	共通座標系を持たない5台の非同期式ファットロボットによる集合問題について	ミドル
B-2	14:35	柴田 将拡	大阪大学	リングおよび木におけるモバイルエージェントの部分集合アルゴリズム	ミドル
B-3	14:55	李 絢	大阪大学	Move-efficient algorithms for group gossiping of mobile agents	ミドル
B-4	15:15	大野 陽香	名古屋工業大学	st-ordering問題を利用した極大DAG構成自己安定アルゴリズムに関する研究	ミドル
B-5	15:35	大川 浩昂	名古屋工業大学	Filling the Bandwidth Gap in Distributed Complexity for Global Problems	ミドル
セッションC：グラフ、アルゴリズム(16:05-17:30)				座長：田岡 智志	
C-1	16:05	橋村 勇志	九州大学	非負行列分解の絶対値誤差最小化について	ショート
C-2	16:20	三重野 琢也	九州大学	エネルギー制限つきエージェントによるデータ配送問題	ショート
C-3	16:35	玉谷 賢一	九州大学	Pebble Covering の数え上げ高速化	ショート
C-4	16:50	藤原一毅	国立情報学研究所	Pursuit of low-latency networks for supercomputers	ミドル
C-5	17:10	松前 進	佐賀大学	Random Address Permute-Shift Technique for the Shared Memory on GPUs	ミドル

18:30 夕食

セッションD：並列計算1(20:00-21:25)				座長：小池 敦	
D-1	20:00	藤田 徹	広島大学	GPUのPTXコードを用いたランダムアドレスシフトの厳密評価	ショート
D-2	20:15	岡本 悟史	広島大学	GPUの共有メモリアクセスのレイテンシとスループットの計測	ショート
D-3	20:30	高藤大介	広島大学	C2CU: A CUDA C Program Generator for Bulk Execution of a Sequential Algorithm	ミドル
D-4	20:50	谷 和也	広島大学	GPUを用いた高スループット計算システムの実装	ショート
D-5	21:05	笠置 明彦	広島大学	Parallel Algorithms for the Summed Area Table on the Asynchronous Hierarchical Memory Machine	ミドル

9月18日

セッションE：並列計算2(8:30-10:30)				座長：藤原 一毅	
E-1	8:30	田中 俊介	名古屋工業大学	編集距離空間における1-挿入被覆符号のサイズ限界式	ミドル
E-2	8:50	笹村 幸秀	九州工業大学	GPGPUを用いた連結センサカバーに対する蜂群最適化	ショート
E-3	9:05	小池 敦	国立情報学研究所	GPUを用いた並列ソートアルゴリズム	ミドル
E-4	9:25	松本 直之	広島大学	ハードウェアソーティングアルゴリズムのFPGA実装	ショート
E-5	9:40	中西 昭文	九州工業大学	生化学反応計算における基本演算およびソートの実現	ショート
E-6	9:55	前田 翔平	九州工業大学	酵素を用いた数値膜計算における基本演算およびソートの実現	ショート
E-7	10:10	中川 遼	大阪大学	MapReduceによる列挙木探索手法の提案と評価	ミドル
セッションF：ネットワーク(10:40-12:40)				座長：泉 泰介	
F-1	10:40	Jacir L. Bordim	ブラジリア大学	Topology Control in Cooperative Ad Hoc Wireless Networks	ミドル
F-2	11:00	藤田聡	広島大学	On Vertex Cover with Fractional Fan-Out Bound	ミドル
F-3	11:20	田岡智志	広島大学	Solvability for The Maximum Legal Firing Sequence Problem of Conflict-Free Petri Nets with Inhibitor Arcs	ショート
F-4	11:35	小島寛樹	広島大学	ニュースストリームの動的クラスタリング	ショート
F-5	11:50	神崎 僚太	広島大学	iBeaconモジュールを利用した2次元位置座標推定の考察	ショート
F-6	12:05	中島 大志	広島大学	P2Pシステムのためのスケーラブルな木構造ベースの整合性維持手法	ショート
F-7	12:20	亀井清華	広島大学	レビューを対象とした信頼性判断支援システムの提案	ミドル

13:30 自由討論会
19:00 懇親
21:30 自由討論会

9月19日

セッションG：分散アルゴリズム(8:20-10:35)				座長：中村 純哉	
G-1	8:20	麻田 優真	奈良先端科学技術大学院大学	An Efficient Silent Self-Stabilizing Algorithm for 1-Maximal Matching in Anonymous Network	ミドル
G-2	8:40	東 優樹	大阪大学	均一で密なユニットディスクグラフにおける局所情報に基づく経路の自己構成手法	ショート
G-3	8:55	安達 駿	大阪大学	局所的トリガ数え上げ問題に対する分散アルゴリズム	ミドル
G-4	9:15	清水 与也	名古屋工業大学	トピック別資源発見問題について	ミドル
G-5	9:35	後藤 千香行	名古屋工業大学	自律分散ロボットによる線分上の被覆問題について	ミドル
G-6	9:55	金 鎔煥	大阪大学	A Local Information Based Distributed Algorithm Constructing 3-Nodes Rectilinear Steiner Tree in Virtual Grid Networks	ミドル
G-7	10:15	伊藤 瑠美	大阪大学	自己安定平衡探索木構成アルゴリズム	ミドル
セッションH：最適化(10:45-12:40)				座長：大下 福仁	
H-1	10:45	Rifki Omar	九州大学	A hybrid approach of optimization and sampling for robust portfolio selection	ミドル
H-2	11:05	土中 哲秀	九州大学	A Fixed-Parameter Algorithm for Max Edge Domination	ミドル
H-3	11:25	山下 智大	九州大学	分割されたナップサック制約付き最大被覆問題に対する近似アルゴリズム	ショート
H-4	11:40	林 毅	法政大学	計算機アーキテクチャの教育を支援する可視化レベル変更可能なPDP-11シミュレータ	ミドル
H-5	12:00	小林 晴紀	法政大学	アセンブリ言語教育支援教材Sim AIの設計と実装	ミドル
H-6	12:20	中川路 克之	大阪大学	WebRTCシステムに適したテストフレームワークの開発	ミドル

12:40 閉会

On the Worst-Case Initial Configuration for Conservative Connectivity Preservation

Daichi Kaino

Department of Computer Science and Engineering
Nagoya Institute of Technology
Aichi, Japan
Email:cke17539@stn.nitech.ac.jp

Taisuke Izumi

Department of Computer Science and Engineering
Nagoya Institute of Technology
Aichi, Japan
Email: t-izumi@nitech.ac.jp

***Abstract*—We consider the system of robots with *limited-visibility*, where each robot can see only the robots within the unit visibility range (*a.k.a.* the unit distance range). In this model, we focus on the inherent cost we have to pay for connectivity preservation in the conservative way (i.e., in any execution, no edge of the visibility graph is deleted). We present a bad configuration with the visibility graph of diameter D for which any conservative algorithm requires $\Omega(D^2)$ rounds to make all robots movable, where D is the diameter of the initial visibility graph. This result implies that we inherently need edge-deletion mechanisms to solve many connectivity-preserving problems (as considered in [1], [2], [5]) within $o(D^2)$ rounds.**

***Keywords*-distributed algorithm; mobile robot; limited visibility; lower bound;**

I. Introduction

Algorithmic studies about autonomous mobile robots is recently emerging in the distributed computing community. In most of those studies, a robot is modeled as a point in a Euclidean plane, and its abilities are quite limited: It is usually assumed that robots are *oblivious* (*i.e.* no memory is used to record past situations), *anonymous* (*i.e.* no ID is available to distinguish two robots), and *uniform* (*i.e.* all robots run the same identical algorithm). In addition, it is also assumed that each robot has no direct means of communication. The communication between two robots is done in an implicit way by having each robot observe its environment, which includes the positions of the other robots.

More challenging settings of algorithmic robotics is the *limited visibility* model [1], [2], [5], where each robot can see only the robots within the unit visibility range (*a.k.a.* the unit distance range). The limited visibility is a practical assumption but makes the design of algorithms quite difficult because it prevents each robot from obtaining the global information about all other robots. Furthermore, it also brings another design issue, called *connectivity preservation* [4]: Oblivious robots cannot use the previous history of their execution. Hence, once some robot r_1 disappears from the visibility range of another robot r_2, r_2 can behave as if r_1 does not exist in the system and vice versa. Since the cooperation between r_1 and r_2 becomes impossible, it follows that completing any task starting from those situations is also impossible. This phenomenon can be formally described by using a *visibility graph*, which is the graph induced by the robots (as nodes) and their visibility relationship (as edges). The requirement we have to guarantee in the limited visibility model is that any task or sub-task in an algorithm must be achieved in the manner that preserves the connectivity of the visibility graph.

A standard approach to achieve the connectivity preservation is that we always disallow the movement of the robots which can bring the deletion of edges in the visibility graph. In this paper we call algorithms adopting this approach *conservative*. Since the deletion of an edge does not necessarily bring the disconnection of the visibility graph, conservative algorithms are overly safe in some sense. On the other hand, surprisingly, every known algorithm for the limited visibility model belongs to the class of conservative algorithms.

The main focus of this paper is to reveal the inherent cost we have to pay for the conservative connectivity preservation. As we stated, the conservativeness property restricts the movement of robots causing the edge deletion of visibility graphs. That kind of movement is characterized by the notion of *blocked locations* [3]. In any conservative algorithm, the robot on a blocked location cannot change its position. A simple example of blocked locations is as follows: A robot r_0 is placed at the origin of the global coordinate system, and r_1, r_2, r_3 are placed on $(-1, 0)$, $(1, 0)$, $(0, 0.1)$ respectively. In this case, a small movement by r_0 (e.g., the movement to $(0.1, 0)$) causes no disconnection of the visibility graph. However, that movement of r_0 causes the deletion of edges (r_0, r_1) and (r_0, r_2), and thus generally r_0 is possible to move but it is not possible in conservative algorithms. Any conservative algorithm must stop the movement of r_0 until r_1 or r_2 gets close to r_0. Thus, at least one extra round is incurred to resolve blocked locations of r_0. This can be seen as an extra cost of the conservative approach. From this observation, a natural question raises up: How much time is necessary to make all robots non-blocked in conservative algorithms? A trivial lower bound for this question is to place n robots at the coordinates $(0, 0), (1, 0), \cdots, (n - 1, 0)$. This configuration

obviously requires $\Omega(n)$ rounds for making all robots non-blocked. However, the visibility graph of this configuration has diameter $n-1$. It is not so surprising because we can embed a "long chain" of blocked locations if the visibility graph has a large diameter. More precisely, we can trivially have the configuration satisfying that (1) its visibility graph has diameter D and (2) $\Omega(D)$ rounds are required to make all robots non-blocked. On the other hand, the best known upper bound is $O(D^2)$ rounds for configurations with the visibility graphs of diameter D, which is shown in our prior work[3]. The problem of filling the complexity gap between $\Omega(D)$ and $O(D^2)$ has remained open. The contribution of this paper is to close this gap: We show a bad configuration with the visibility graph of diameter D for which any conservative algorithm requires $\Omega(D^2)$ rounds to make all robots non-blocked. This result implies that we inherently need edge-deletion mechanisms to solve many connectivity-preserving problems (as considered in [1], [2], [5]) within $o(D^2)$ rounds.

II. MODEL

The system consists of n robots, denoted by $r_0, r_1, r_2, \cdots, r_{n-1}$. Robots are *anonymous*, *oblivious* and *uniform*. That is, each robot has no identifier distinguishing itself and others, cannot explicitly remember the history of its execution, and works following a common algorithm independent of the value of n. In addition, no device for direct communication is equipped. The cooperation of robots is done in an implicit manner: Each robot has a sensor device to observe the environment (i.e., the positions of other robots). One robot is modeled as a point located on a two-dimensional space. Observing environment, each robot can see the positions of other robots transcripted in its *local coordinate system*. We assume *limited visibility*: Each robot can see only the robots located within unit distance. Each robot executes the deployed algorithm in *computational cycles* (or briefly *cycles*). At the beginning of a cycle, a robot observes the current environment (i.e., the positions of other robots) and determines the destination point based on the deployed algorithm. Then, the robot moves toward the computed destination. It is guaranteed that each robot necessarily reaches the computed destination at the end of the cycle. As the timing model, we assume fully-synchronous model. In fully synchronous worlds,any execution follows a discrete time $1, 2, 3 \cdots$. At the beginning of each time unit, every robot is activated and performs one cycle. Note that this assumption is stronger than the standard ones such as ATOM[6], but it leads more general results because we consider lower bounds. That is, our argument for the worst cases holds even for full-synchronous systems, and thus it clearly holds for other weaker models.

Throughout this paper, we use the following notations and terminology: To specify the location of each robot consistently, we use the global coordinate system. Notice that the global coordinate system is introduced only for ease the explanation, and thus robots are not aware of it. The origin of the global coordinate system is denotes by $\mathbf{o}$.For any two coordinates $\mathbf{a}$ and $\mathbf{b}$, $\overline{\mathbf{ab}}$ denotes the segment whose endpoints are $\mathbf{a}$ and $\mathbf{b}$, and $|\mathbf{ab}|$ denotes its length. A *configuration* is the multiset consisting of all robot locations. We define $C(t)$ as the configuration at t.

A. Visibility Graph

A visibility graph $G(t)$ is the graph where nodes represent robots and an edge between two robots implies the visibility between two robots(See fig1). More formally, the visibility graph at t consists of n nodes $\{v_0, v_1, v_2, \cdots v_{n-1}\}$. Nodes v_i and v_j are connected if and only if r_i and r_j are visible to each other.

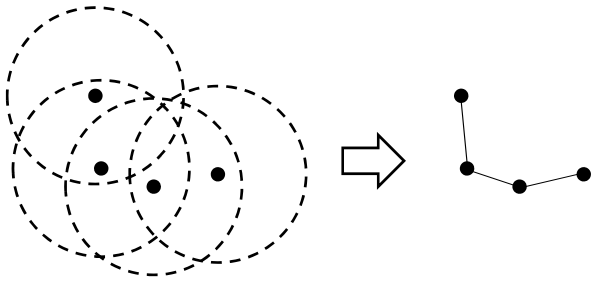


Figure 1. Visibility graph

B. Conservative Connectivity Preservation

The algorithms we consider in this paper belongs to a class called *conservative connectivity preservation*. Formally, an algorithm $\mathcal{A}$ is *conservatively connectivity-preserving* if in any execution of $\mathcal{A}$ edges of the visibility graph are never deleted, That is, let $G(t) = (V, E(t))$ be the visibility graph at t, any execution of $\mathcal{A}$ satisfies $E(t) \subseteq E(t+1)$.

C. Blocked location

We explain the notion of *blocked* locations. Intuitively, a blocked location is the place such that the robot on that location cannot move without deletion of edges in the visibility graph.

Definition 1: Let $\mathbf{p_c}$ be a location, C be the circle centered at $\mathbf{p_c}$ with diameter one, and $B = \{\mathbf{p_0}, \mathbf{p_1}, \mathbf{p_2}, \cdots \mathbf{p_j}\}$ be the set of all locations on the boundary of C. The location $\mathbf{p_c}$ is *blocked* if no arc of C with a center angle less than π can contain all locations in B.

Examples illustrating the notion of blocked locations are shown in Fig. 2, Fig.3.

Intuitively, for a robot r_i to be movable while preserving edges of the visibility graph, its destination must be within distance one from the robots that r_i sees before the

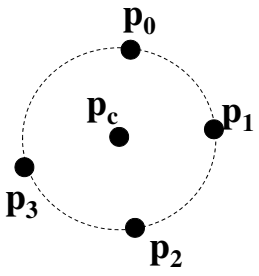


Figure 2. Blocked location

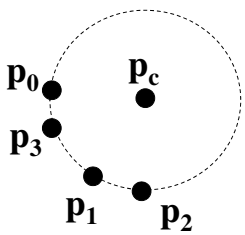


Figure 3. Non-blocked location

movement. For a robot at a blocked location there is no such destination. Assume the contrary, let r_i be blocked and move to some other point $\mathbf{p}(\neq \mathbf{r_i})$. Then, we take the line l which is orthogonal to the vector $\mathbf{p} - \mathbf{r_i}$ and passes through $\mathbf{r_i}$. This line cuts the circle C into two arcs with center angle π. From the definition of blocked points, both arcs have at least one robot. However, the arc in the opposite side of $\mathbf{p}$ (about l) is out of r_i's visibility after the movement to $\mathbf{p}$ (see Fig. 4. Thus, if a robot r_j is on a blocked location, it cannot move anywhere without deletion of edges.

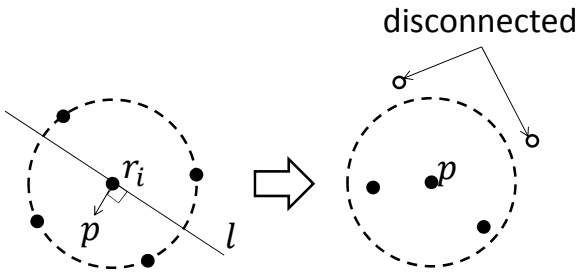


Figure 4. The deletion of edges in visibility graphs

The observation above implies the following lemma:

Lemma 1: In any execution of conservative connectivity-preserving algorithms, no robot on a blocked configuration can not move.

III. LOWER BOUND

In this section, we show that $\Omega(D^2)$ rounds are necessary to make all robots non-blocked. More precisely,given an arbitrary value D we construct an initial configuration for which any conservative connectivity-preservation algorithm takes $\Omega(D^2)$ rounds to make all robots non-blocked.

A. The Worst-Case Construction

We refer the initial configuration constructed in this section as $C_{BAD}(D)$. For any given $D > 0$. the configuration $C_{BAD}(D)$ consists of $(2 + D)2^{D^2}$ robots (and thus $D = O(\sqrt{\log n})$ must be satisfied).

We define π_k to be the circle of radius $\sqrt{k}$ whose center is at the origin $\mathbf{o}$ of the global coordinate system. For $k = 0$ π_k represents the origin $\mathbf{o}$. The construction is done by recursively placing robots on π_k to block all the robots in π_{k-1}. Let R_k be the set of points on π_k where a robot will be placed.

We first show a fundamental lemma for the construction of R_k ($k \geq 0$).

Lemma 2: We define $\mathbf{p}$ to be any points on the π_k,and $l_\mathbf{p}$ to be the tangent line of π_k at $\mathbf{p}$.Then letting $l_\mathbf{p} \cap \pi_{k+1} = \{\mathbf{q}_1, \mathbf{q}_2\}, |\mathbf{q}_1 - \mathbf{p}| = |\mathbf{q}_2 - \mathbf{p}| = 1$ is satisfied.

Proof: Because point $\mathbf{p}$ is placed on π_k, $|\mathbf{p}| = \sqrt{k}$. Similarly,$|\mathbf{q_1}| = |\mathbf{q_2}| = \sqrt{k+1}$. Also hold points $\mathbf{q_1}$ and $\mathbf{q_2}$ are placed on line $l_\mathbf{p}$ and thus $\overline{\mathbf{op}}$ is orthogonal to $l_\mathbf{p}$. We can apply the Pythagorean theorem to three points $\mathbf{o}$,$\mathbf{p}$,$\mathbf{q_1}(or\mathbf{q_2})$(See Fig5). Thus $|\mathbf{q}_1 - \mathbf{p}| = |\mathbf{q}_2 - \mathbf{p}| = 1$ holds. ■

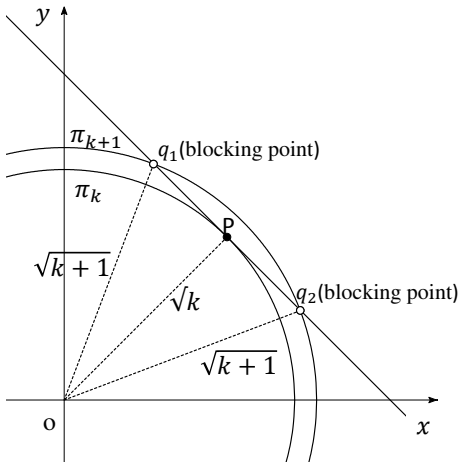


Figure 5. The proof of Lemma 2 and blocking points

The two points $l_\mathbf{p} \cap \pi_{k+1}$ defined in this lemma is called the *blocking points* of $\mathbf{p}$. This lemma naturally induces the construction of $C_{BAD}(D)$, which is defined as follows:

- Place one robot at the origin of the global coordinate system (i.e., on π_0). In addition, place two robots at the coordinate $(1, 0)$ and $(0, -1)$. The points in R_1 consists of these two robots.
- For any $\mathbf{p} \in \pi_k$ ($2 \leq k \leq D^2$) where a robot is located, place two robots on its two blocking points.
- To bound the diameter within D, for each segment between $\mathbf{p} \in R_k$ ($2 \leq k \leq D^2$) and the origin, place $\lfloor \sqrt{k} \rfloor$ robots with unit interval(See Fig:6).

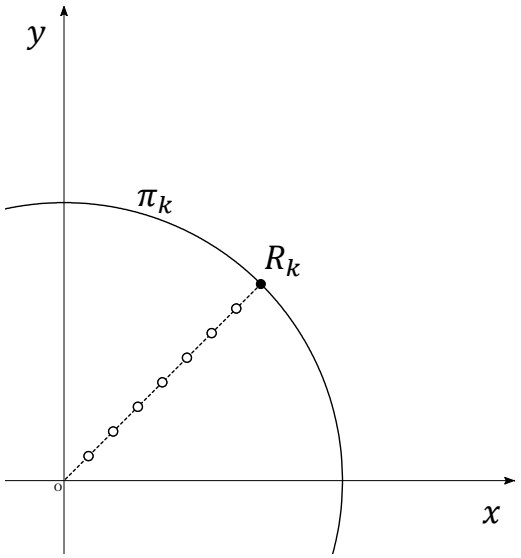


Figure 6. The third step of constructing $C_{BAD}(D)$

Figure 7 shows the construction for $D^2 = 3$. By the third step of the construction above, it is obvious that the visibility graph of $C_{BAD}(D)$ has diameter at most D. Thus remaining issue is to show that this configuration requires D^2 rounds to make all robots non-blocked. We show the example of C_{BAD}.

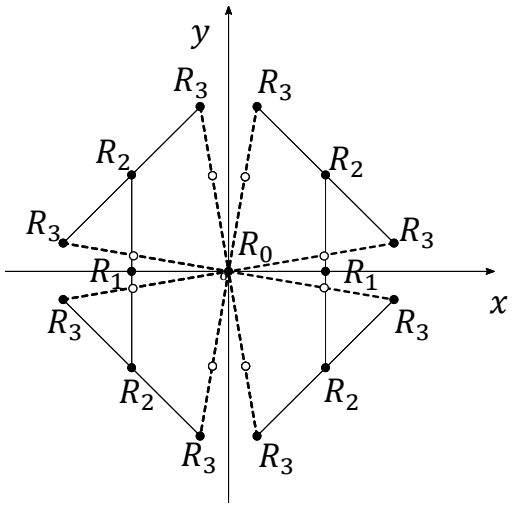


Figure 7. The illustration of C_{BAD} when k=3

Lemma 3: Any robots on R_{D^2-k} are blocked at round k.

Proof: We prove the lemma by induction on k. (**Basis**): If $k = 0$ and 1, the lemma trivially holds. (**Inductive step**): Suppose as the induction hypothesis that all robots in R_{D^2-k} are blocked at round k. In initial placement, any points $\mathbf{p} \in R_{D^2-(k+1)}$ are placed at each blocking points. From including step,robots on the these points blocked until round k, so they don't yet move at beginning time of round $k+1$. Therefor, robots on points $\mathbf{p} \in R_{D^2-(k+1)}$ are blocked at round $k + 1$. ■

Consequently, we have the main lemma below:

Theorem 1: For any D, the exists a configuration $C_{BAD}(D)$ with the visibility graph of diameter D such that in any execution of conservative algorithms starting from $C_{BAD}(D)$ requires $\Omega(D^2)$ rounds to make all the robots non-blocked.

This result implies that for many connectivity-preserving problems, no algorithm can achieve the running time sub-quadratic of D at the worst case unless it incurs edge deletions of visibility graphs, which can be seen as an inherent cost by conservative algorithms.

IV. CONCLUSION

In this paper, we presented a bad configuration with the visibility graph of diameter D for which any conservative algorithm requires $\Omega(D^2)$ rounds to make all robots movable, where D is the diameter of the initial visibility graph. Since we need $(2 + D)2^{D^2}$ robots, to construct the bad configuration, our result holds only for the case of $D = O(\sqrt{\log n})$. An open problem is to present the similar bad configuration for larger D (i.e. $D = \omega(\sqrt{\log n})$).

REFERENCES

[1] Hideki Ando and Yoshinobu Oasa and Ichiro Suzuki and Masafumi Yamashita, Distributed memoryless point convergence algorithm for mobile robots with limited visibility, IEEE Transactions on Robotics and Automation,volume 15(5), pages 818-828, 1999.

[2] Paola Flocchini and Giuseppe Prencipe and Nicola Santoro and Peter Widmayer, Gathering of asynchronous robots with limited visibility, Theoretical Computer Science, 337(1-3), pages 147-168, 2005.

[3] Taisuke Izumi and Maria GradinariuPoptop-Butucaru and Sebastien Tixeuil, Connectivity Preserving Scattering of Mobile Robots with Limited Visibility, Proc. of Intl. Symposium on Stabilization, In Safety and Security of Distributed Systems,volume 6336, pages 319-331, 2010.

[4] M. Ji and M. Egerstedt, Distributed Coordination Control of Multi-Agent Systems while Preserving Connectedness, IEEE Transactions on Robotics, volume 23(4), 2007.

[5] Souissi, Samia and Défago, Xavier and Yamashita, Masafumi, Using eventually consistent compasses to gather memory-less mobile robots with limited visibility, ACM Transactions on Autonomous and Adaptive Systems, volume 4(1), pages 1-27, 2009.

[6] Ichiro Suzuki and Masafumi Yamashita,Distributed Anonymous Mobile Robots: Formation of Geometric Patterns,SIAM Journal of Computing,volume 28(4),pages 1347-1363,1999.

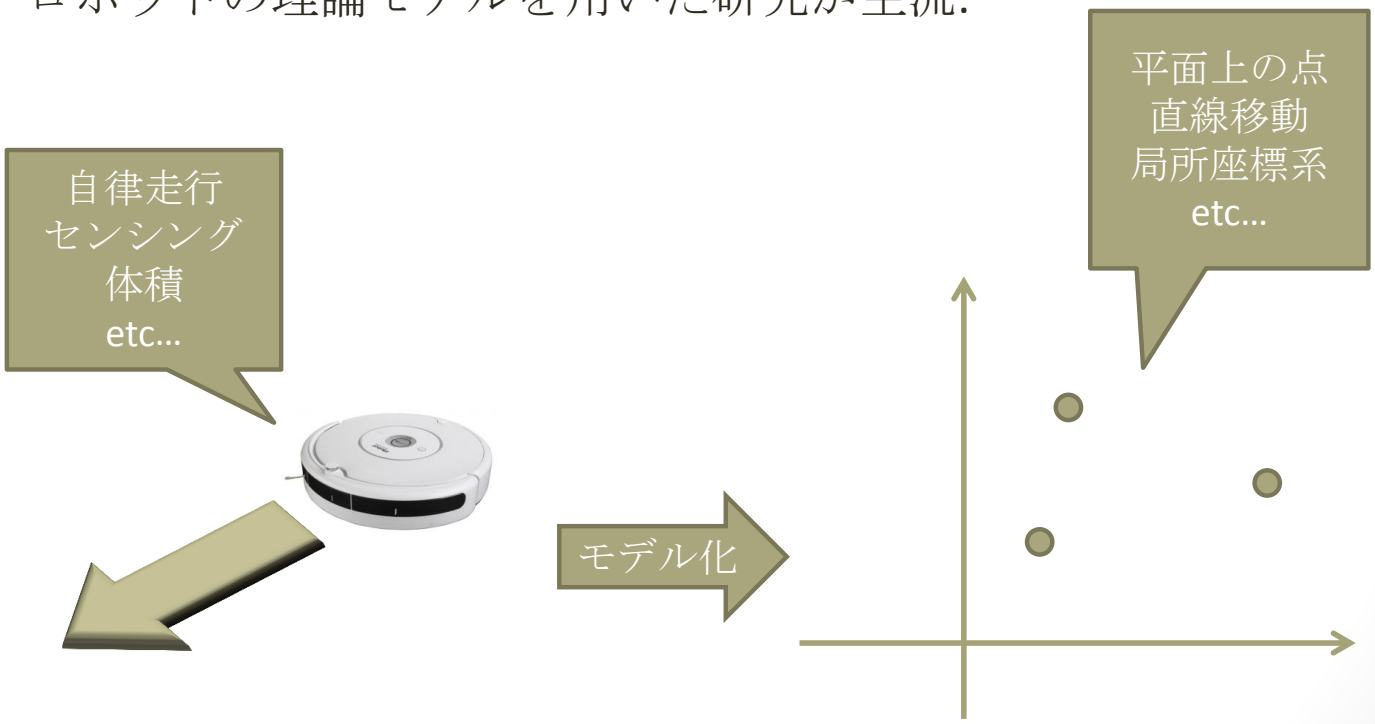
状態を持つ自律分散ロボット群の能力について

寺井 智史	法政大学大学院	◎
和田幸一	法政大学	
片山善章	名古屋工業大学大学院	
Shantanu Das	Aix-Marseille University	

はじめに

- 自律分散ロボット群の研究

　自律的に動作し,全体としては協調的に行動するロボットの理論モデルを用いた研究が主流.



はじめに

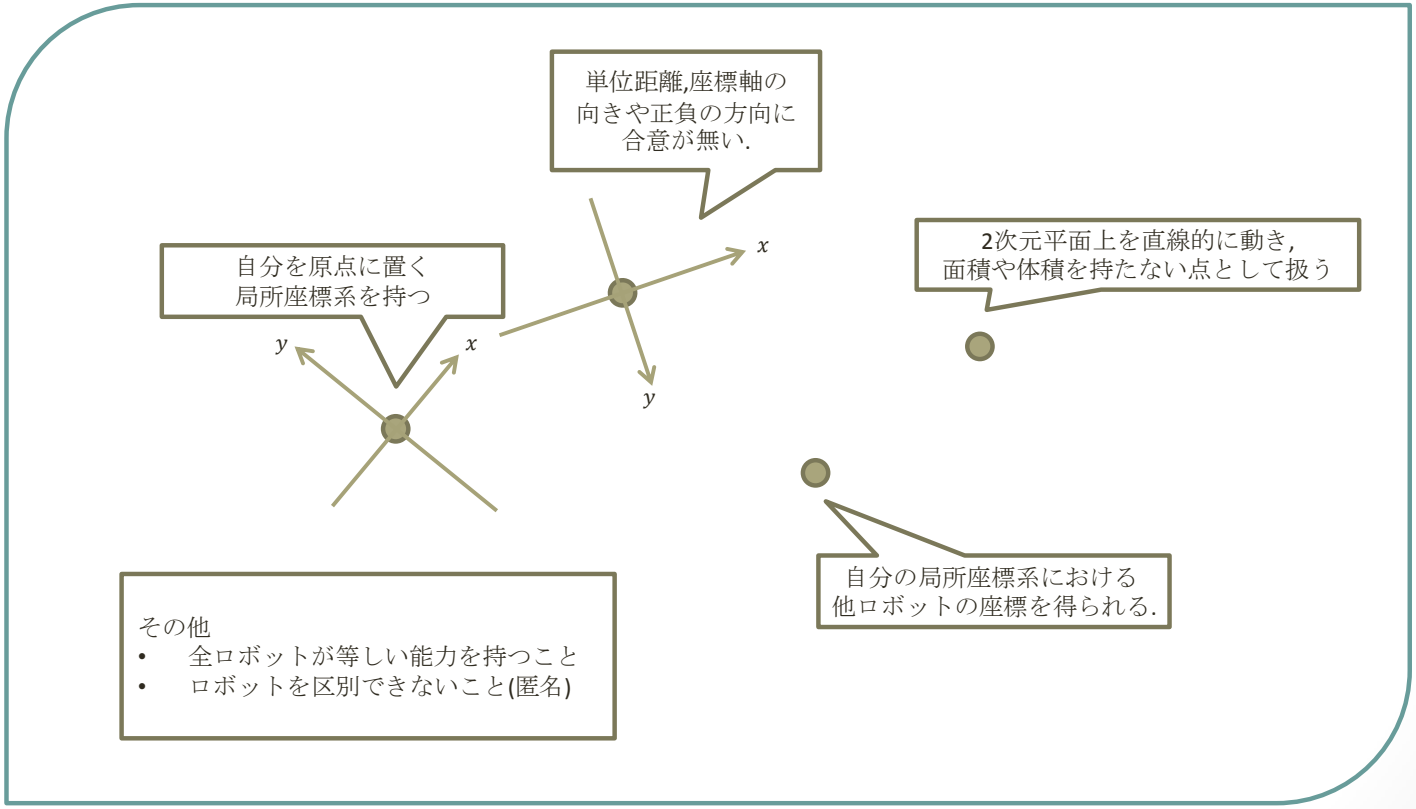
■研究背景

最も単純なモデルでは非可解な問題を解くために状態$(light)$を持つロボットモデルが考案された,

本研究では,既存の状態付きロボットを拡張したモデルを提案し,役割の異なる二種類の状態の間にある性能差を明らかにするための方針を示す.

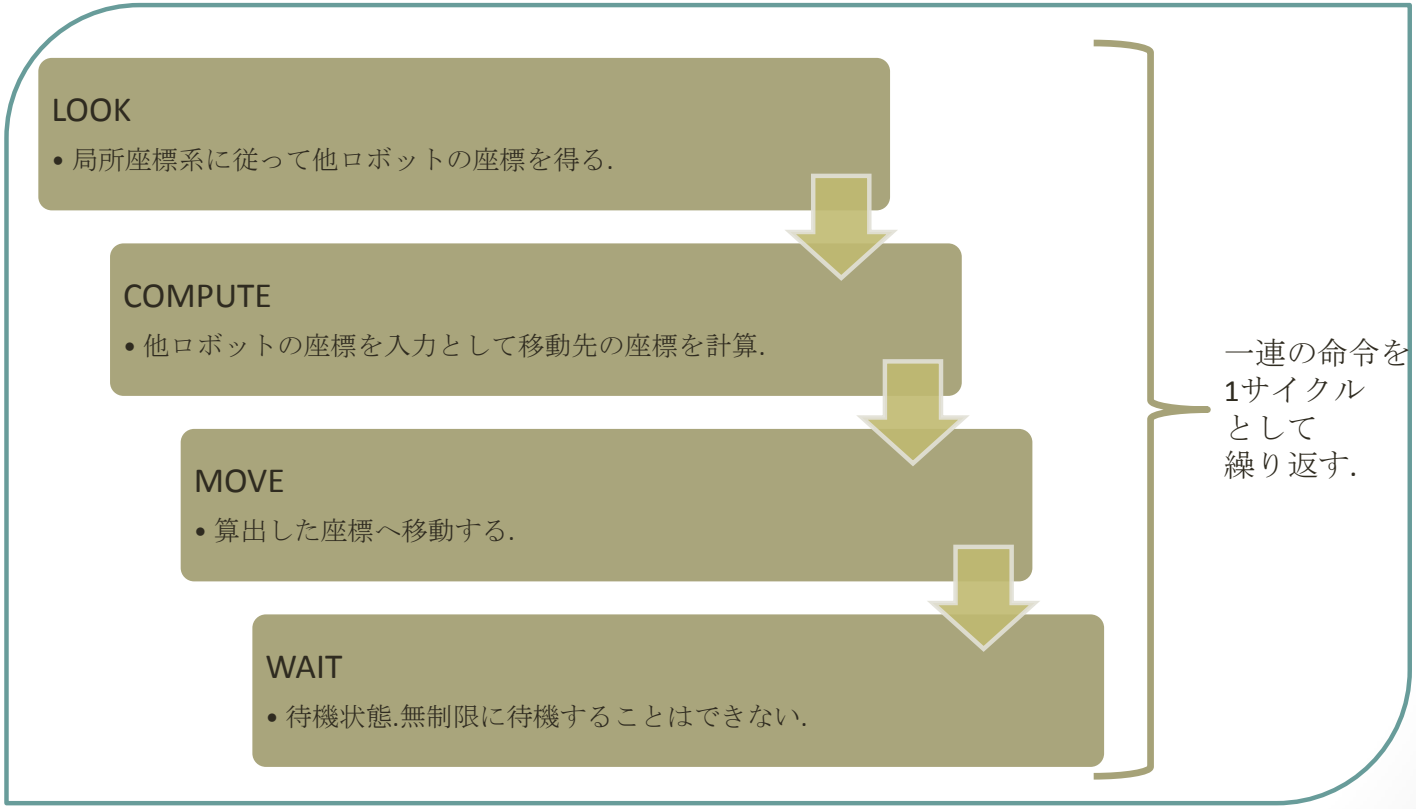
理論モデル

- 基本的なモデルの紹介
  - 能力



理論モデル

- 基本的なモデルの紹介
  - 動作



理論モデル

- 基本的なモデルの紹介
  - 動作に関する仮定

- 匿名

  ロボット同士を区別することができない.

- 無記憶

  常に最後に行ったLOOK命令によって得た情報のみに基づいてCOMPUTE命令を行う.

スケジュール

個々のロボットが持っている能力や動作を理論モデルによって定めた.

次にそのロボット達を「どう動かすか」を定める.

それには

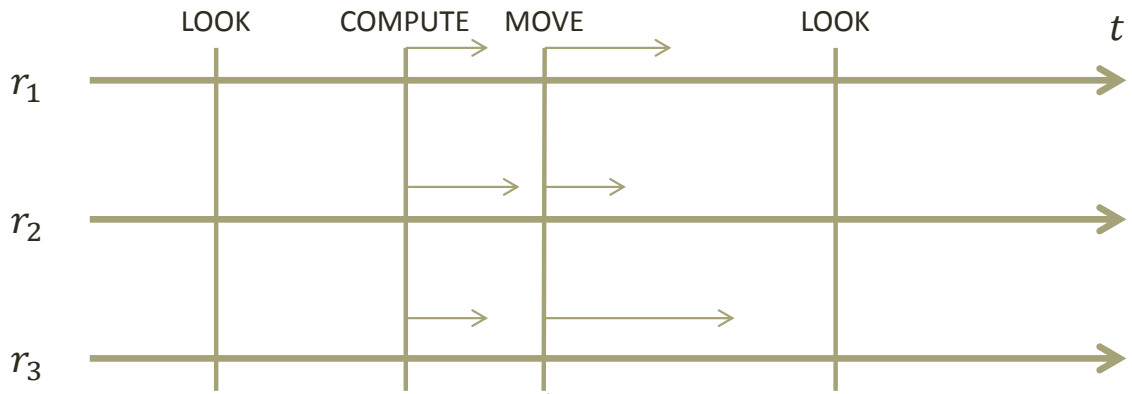
- $FSYNC(Fully\ syncronous)$
- $SSYNC(Semi\ syncronous)$
- $ASYNC(Asyncronous)$

の3種類のスケジュールがよく使われる.

スケジュール

- FSYNC

各ロボットの動作を時系列に沿って並べると…

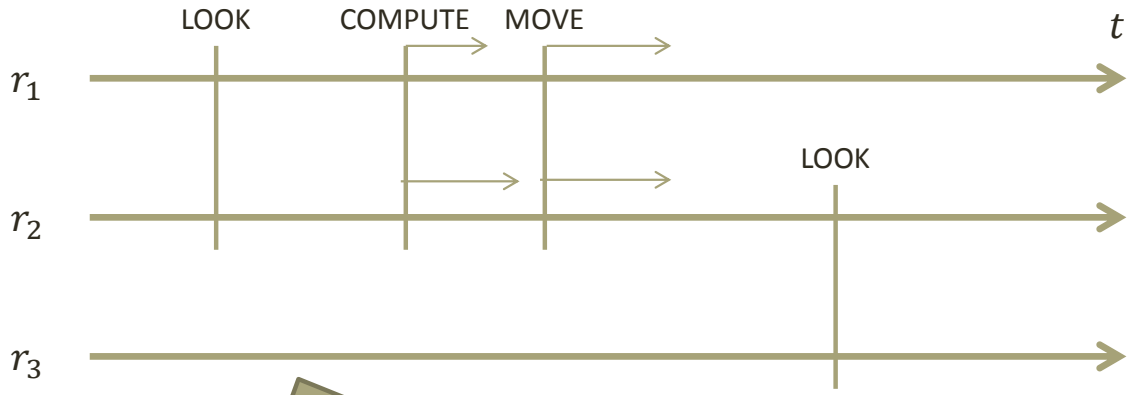


全サイクルで
全ロボットが同期して
動作する.

スケジュール

- SSYNC

各ロボットの動作を時系列に沿って並べると…

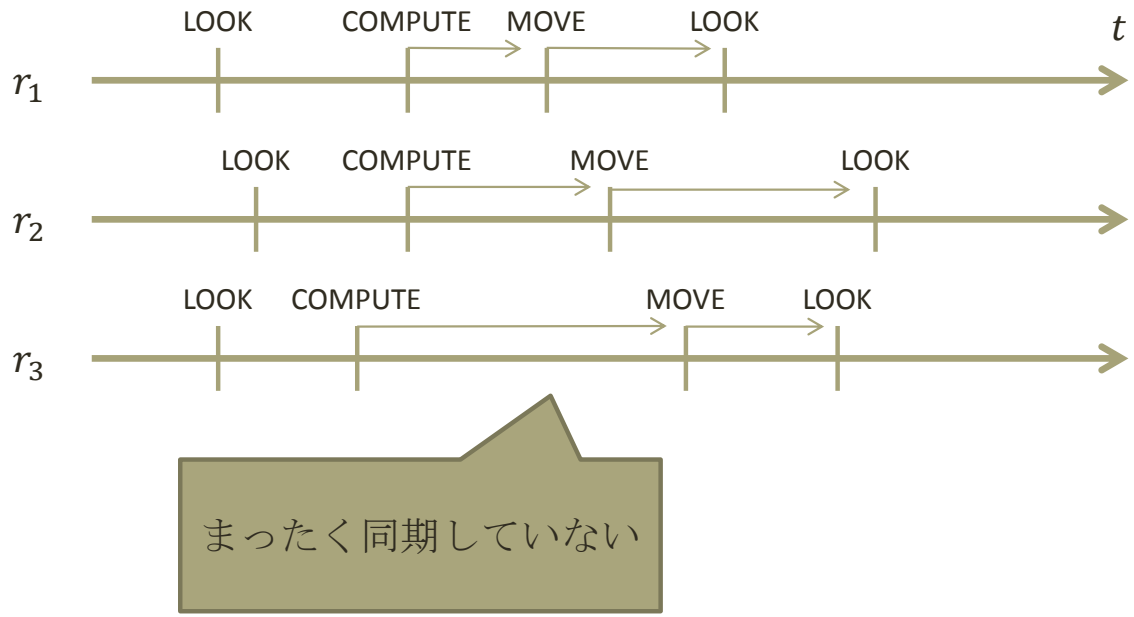


各サイクルでロボットの動作は同期しているが,サイクルを実行しないロボットが存在している.

スケジュール

- ASYNC

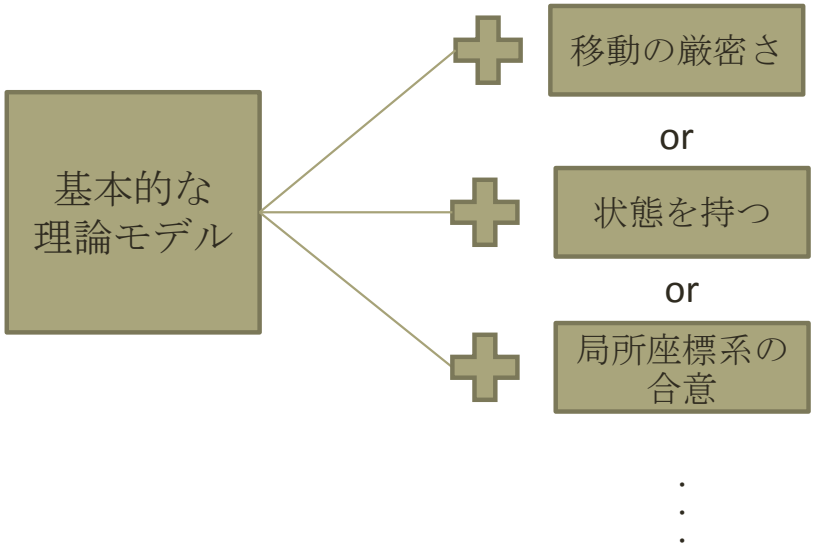
各ロボットの動作を時系列に沿って並べると…



モデルとスケジュール

上記のようなモデルとスケジュールを選んで問題を解く.

理論モデルには適宜追加の仮定を加えることがあり,今回用いる状態($light$)もその一つである.

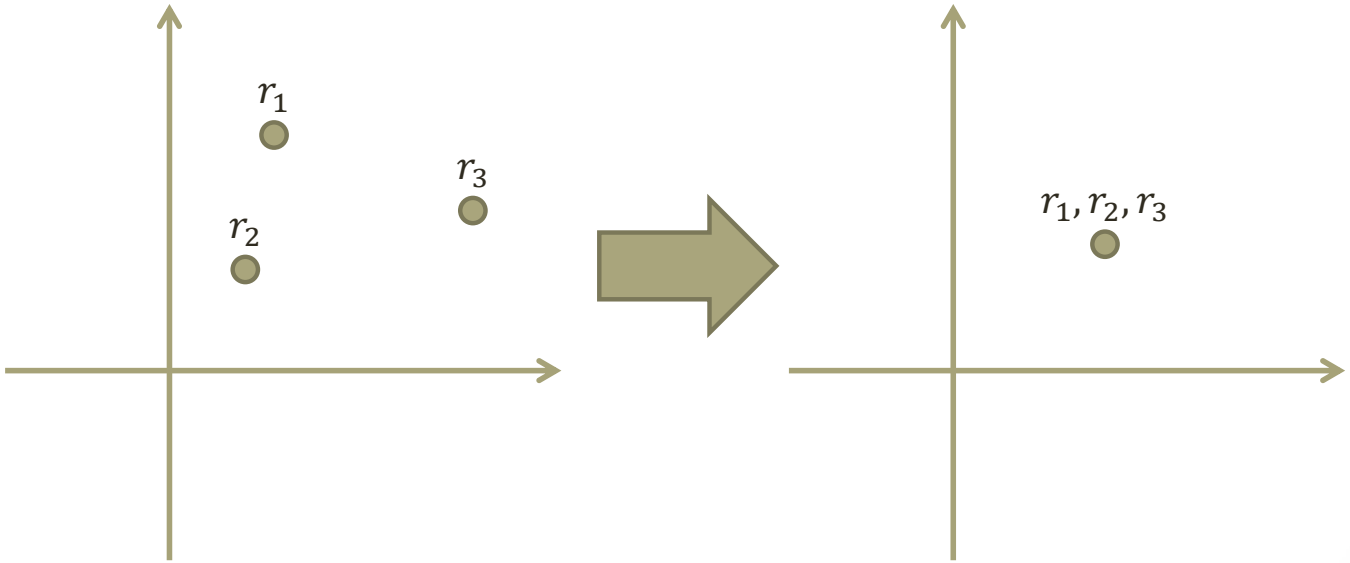


集合問題とランデブー問題

- 集合問題

  $n(\in \mathbb{N})$台のロボットが,任意の初期配置から予め決められていない1点に集まることができるか,という問題.

  $n = 2$のときを特別にランデブー問題と呼ぶ.



状態を持つモデル

これらの問題は基本的な理論モデルでは一般的には解くことができないことが知られている.

表1.基本的なモデルでの結果

	ASYNC	SSYNC	FSYNC
集合	非可解	非可解	可解
ランデブー	非可解	非可解	可解

状態を持つモデル

- 状態($light$)

ロボットの内部状態を記録できる定数ビットの記憶領域.

LOOK命令によって他ロボットの座標とともに状態を認識することが出来る.

状態の可視性によって以下のようなモデルに分ける.

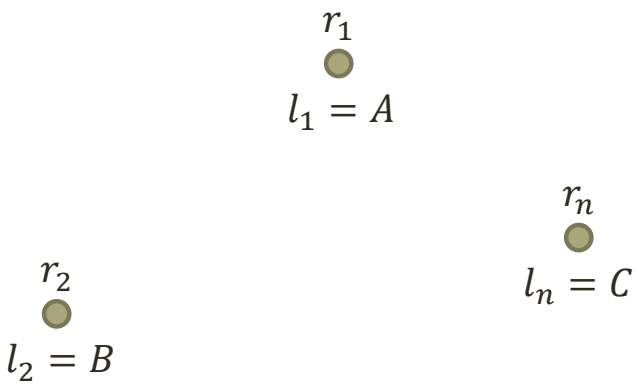
表2.モデルと参照できる状態

	自分の状態	相手の状態
$full-light$	○	○
$internal-light$	○	×
$external-light$	×	○

状態を持つモデル

- システムの外から見た各ロボットの状態$l_k(k = 1, \ldots, n)$

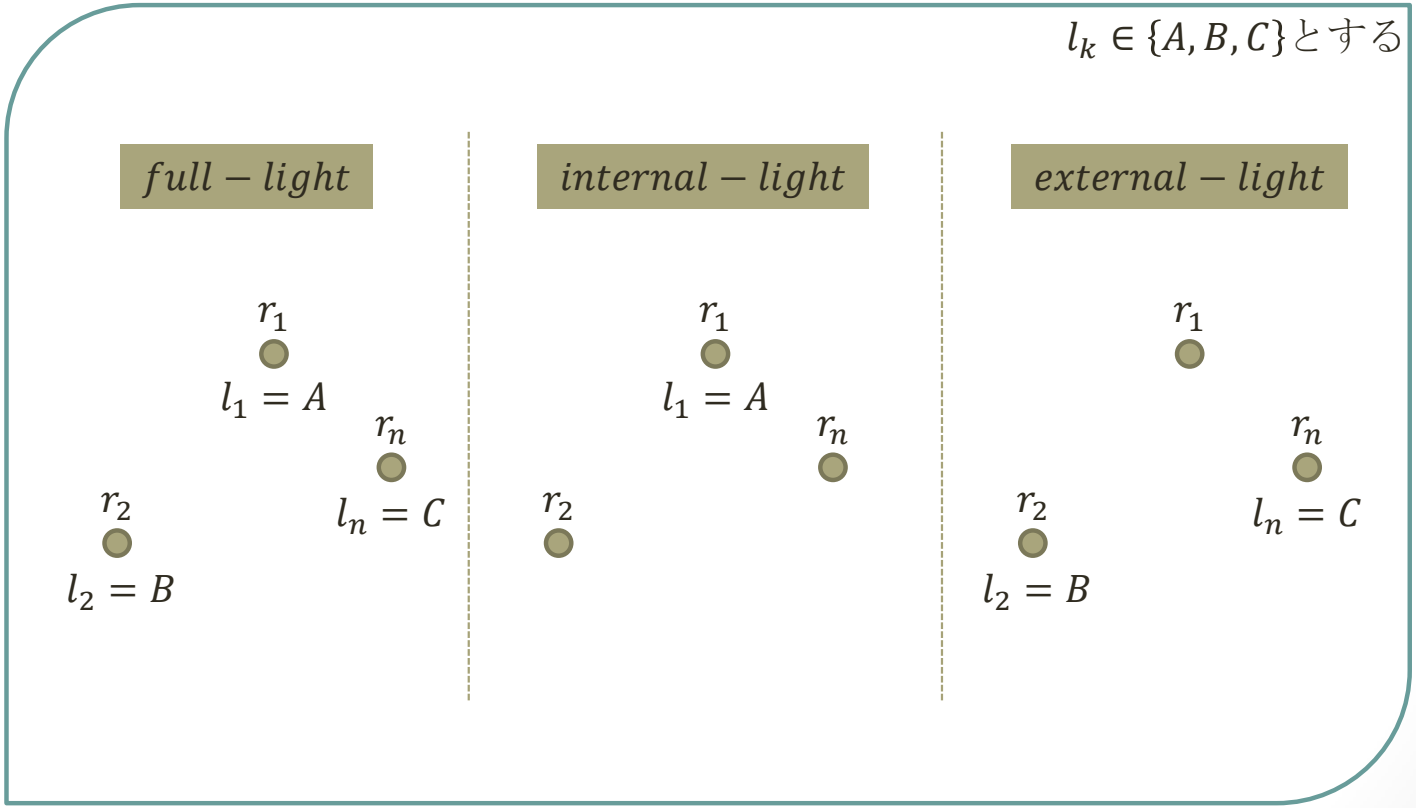
$l_k \in \{A, B, C\}$とする



上のような状況を考え,r_1が
LOOK命令を行った場合を考える.

状態を持つモデル

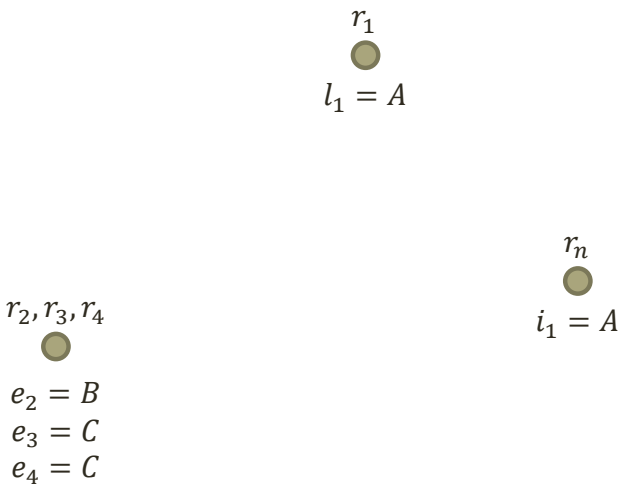
- 状態$l_k(k = 1, \ldots, n)$の見え方



状態を持つモデル

- 3台以上のときの外部状態の見え方

$l_k \in \{A, B, C\}$とする

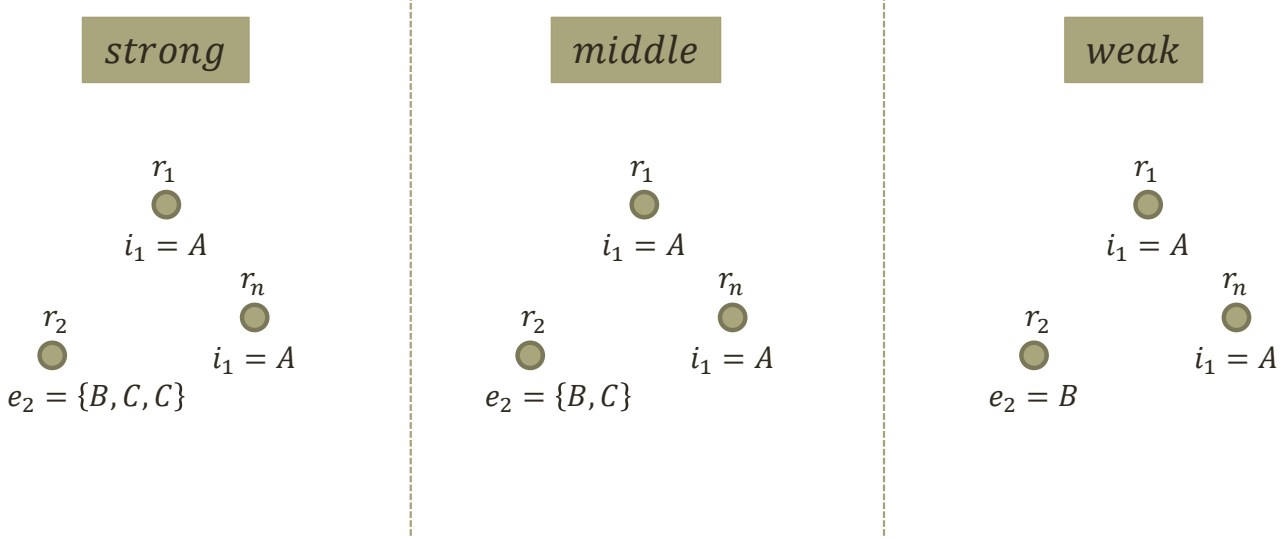


上のような状況を考え,r_1が
LOOK命令を行った場合を考える.

状態を持つモデル

- 3台以上のときの外部状態の見え方

$l_k \in \{A, B, C\}$とする



状態を持つモデル

■既存の結果

表3.移動が厳密な場合のランデブー問題

	ASYNC	SSYNC	FSYNC
$full-light$			1
$internal-light$		6	1
$external-light$	12		1

表4.移動が厳密でない場合のランデブー問題

	ASYNC	SSYNC	FSYNC
$full-light$	4	2	1
$internal-light$		3	1
$external-light$	3	3	1

※赤字はδの知識あり

状態を持つモデル

■既存の結果

表5.ASYNCによるシミュレーション

	SSYNC	$full, n$状態 SSYNC	FSYNC
$full - light$ ASYNC	6	$6n$	3

※過去のスナップショットを1回分保存できる記憶領域を持つモデル

モデルの拡張

- 状態の可視性に関する拡張

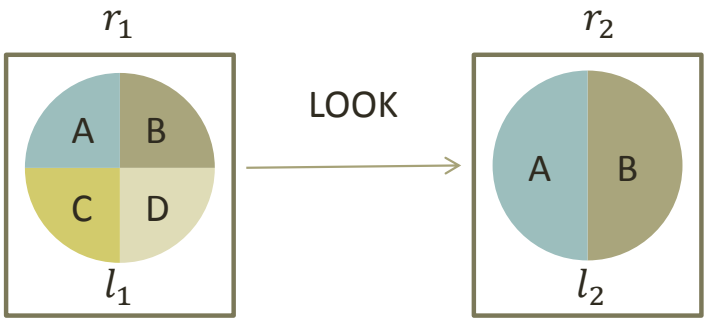
$(s_i, s_e) - slight$

状態を表すパラメータを1つ持つ.

取り得る状態の集合$\mathbb{S}$の中で,内部状態として区別できる状態と外部状態として区別できる状態の数をそれぞれs_i, s_eとする.

状態数は$|\mathbb{S}|$.

$(4,2) - slight$
$\mathbb{S} = \{A, B, C, D\}$
の例



モデルの拡張

先に紹介した3つのモデルは$(s_i, s_e) - slight$の特殊な場合である.

- $full - light$
  - $s_i = s_e$の場合.
- $internal - light$
  - $s_e = 1$の場合.
- $external - light$
  - $s_i = 1$の場合.

モデルの拡張

■状態の役割に関する拡張

□$(i, e) - dlight$

状態を表すパラメータを2つ持つ.

2つの状態集合

- 自分しか見られない内部状態の集合𝕀
- 他ロボットからしか見えない外部状態の集合𝔼

を持ち,これらを表す2つのパラメータを個別に変更できる.

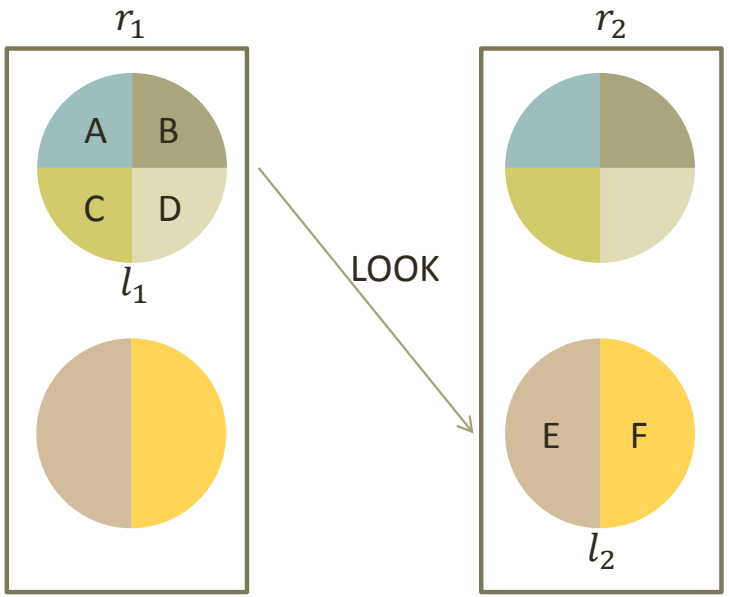
$i = |I|, e = |E|$とする.

状態数は$i * e$とする.

モデルの拡張

- $(i, e) - dlight$

$(4,2) - slight$
$I = \{A, B, C, D\}$
$E = \{E, F\}$
の例



既存研究との関連

- ランデブー問題

　従来の状態付きロボットモデルを用いた結果を新たに定義したモデルによって書き直すと，

表6.移動が厳密$(rigid)$である場合のランデブー問題

	ASYNC	SSYNC	FSYNC
$\boldsymbol{full-light}$			$(1,1)-slight$
$\boldsymbol{internal-light}$		$(6,1)-slight$	$(1,1)-slight$
$\boldsymbol{external-light}$	$(1,12)-slight$		$(1,1)-slight$

既存研究との関連

- ランデブー問題

　従来の状態付きロボットモデルを用いた結果を$(I, E) - light$モデルによって書き直すと

表7.移動が厳密でない場合のランデブー問題

	ASYNC	SSYNC	FSYNC
$full - light$	$(4,4) - slight$	$(2,2) - slight$	$(1,1) - slight$
$internal - light$		$(3,1) - slight$	$(1,1) - slight$
$external - light$	$(1,3) - slight$	$(1,3) - slight$	$(1,1) - slight$
※赤字はδの知識有			

既存研究との関連

- シミュレーション
  ASYNCスケジューラを用いて他のスケジューラを

表8.ASYNCによるシミュレーション

	SSYNC	$full, n$状態 SSYNC	FSYNC
$full - light$ ASYNC	$(6,6) - slight$	$(6n, 6n) - slight$	$(3,3) - slight$

※過去のスナップショットを1回分保存できる記憶領域を持つモデル

現在の状況

初期配置からアルゴリズムGatherNRobotsとElectOneLDS[1]を実行し,そこから状態を用いて集合を達成するという方法で一般の集合問題を解くアルゴリズムを考案した.

表9.移動が厳密でない場合の集合問題

	ASYNC	SSYNC	FSYNC
$(s_i, s_e) - light$		$(3,3) - slight$	
$(1, s_e) - light$		$(4,4) - slight$	

[1]Taisuke Izumi, Yoshiaki Katayama, Nobuhiro Inuzuka, and Koichi Wada, Gathering Autonomous Mobile Robots with Dynamic Compasses: An Optimal Result, 21st International Symposium on Distributed Computing (DISC 2007), Lecture note in Computer Science 4731, pp 298-312

現在の状況

■今後の方針

内部状態と外部状態の性能差を明らかにしたい.そのために,

1. どちらかの状態数を減らしたときに能力差が現れるような問題があるか
   - 例えば$(3,3)-slight$では解けるが$(3,2)-slight$では解けない,など.

2. モデル同士のシミュレーションが可能か
   - 例えば$(i,e)-dlight$で$(s_i,s_e)-slight$のアルゴリズムをシミュレートできるか,など.

という方向性を検討する.

Randomized Pattern Formation Algorithm for Autonomous Mobile Robots

Yukiko Yamauchi* Masafumi Yamashita†

Abstract

We present a randomized pattern formation algorithm for asynchronous oblivious (i.e., memoryless) mobile robots that enables formation of any target pattern. As for deterministic pattern formation algorithms, the class of patterns formable from an initial configuration I is characterized by the symmetricity (i.e., the order of rotational symmetry) of I, and in particular, every pattern is formable from I if its symmetricity is 1. The randomized pattern formation algorithm ψ_{PF} we present in this paper consists of two phases: The first phase transforms a given initial configuration I into a configuration I' such that its symmetricity is 1, and the second phase invokes a deterministic pattern formation algorithm ψ_{CWM} by Fujinaga et al. (DISC 2012) for asynchronous oblivious mobile robots to finally form the target pattern.

1 Introduction

Consider a distributed system consisting of anonymous, asynchronous, oblivious (i.e., memory-less) mobile robots that do not have access to a global coordinate system and are not equipped with communication devices. We investigate the problem of forming a given pattern F from *any* initial configuration I, whose goal is to design a distributed algorithm that works on each robot to navigate it so that the robots as a whole eventually form F from any I. However, a stream of papers [2, 3, 4, 5, 6, 7] have showed that the problem is not solvable by a deterministic algorithm, intuitively because the symmetry among robots cannot be broken by a deterministic algorithm. Specifically, let $\rho(P)$ be the (geometric) symmetricity of a set P of points, where $\rho(P)$ is defined as the number of angles θ (in $[0, 2\pi)$) such that rotating P by θ around the center of the smallest enclosing circle of P produces P itself.[1] Then F is formable from I by a deterministic algorithm, if and only if $\rho(I)$ divides $\rho(F)$, which suggests us to explore a *randomized solution*.

This paper presents a randomized pattern formation algorithm ψ_{PF}. Algorithm ψ_{PF} is *universal* in the sense that for any given target pattern F, it forms F from *any* initial configuration I (not only from I such that $\rho(I)$ divides $\rho(F)$). We however need the following assumptions; the number of robots $n \geq 5$, and both I and F do not contain multiplicities. The idea behind ψ_{PF} is simple and natural; first the symmetry breaking phase realized by randomized algorithm ψ_{SB} translates I into another configuration I' such that $\rho(I') = 1$ with probability 1 if $\rho(I) > 1$, and then the second phase invokes the (deterministic) pattern formation algorithm ψ_{CWM} in [5], which forms F from any initial configuration I' such that $\rho(I') = 1$.[2] Since randomization is a traditional tool to break symmetry, one might claim that ψ_{PF} is trivial. It is not the case at all, mainly because our robots are asynchronous.

*Faculty of Information Science and Electrical Engineering, Kyushu University, Japan. Email: yamauchi@inf.kyushu-u.ac.jp

†Faculty of Information Science and Electrical Engineering, Kyushu University, Japan. Email: mak@inf.kyushu-u.ac.jp

[1] That is, P is rotational symmetry of order $\rho(P)$.

[2] Of course we can also use the pattern formation algorithm in [2] since it keeps the terminal agreement of ψ_{SB} (i.e., the leader), during the formation.

2 System model

Let $R = \{r_1, r_2, \ldots, r_n\}$ be a set of anonymous robots in a two-dimensional Euclidean plane. Each robot r_i is a point and does not have any identifier, but we use r_i just for description.

Each robot repeats a Look-Compute-Move cycle, where it obtains the positions of other robots (in Look phase), computes the curve to a next position with a pattern formation algorithm (in Compute phase), and moves along the curve (in Move phase). We assume that the execution of each cycle ends in finite time. We assume discrete time $0, 1, \ldots$, and introduce three types of asynchrony. In the *fully-synchronous* (FSYNC) model, robots execute Look-Compute-Move cycles synchronously at each time instance. In the *semi-synchronous* (SSYNC) model, once activated, robots execute Look-Compute-Move cycles synchronously. We do not make any assumption on synchrony for the *asynchronous* (ASYNC) model.

A *configuration* is a set of positions of all robots at a given time. [3] Let $p_i(t)$ (in the global coordinate system Z_0) be the position of r_i ($r_i \in R$) at time t ($t \geq 0$). $P(t) = \{p_1(t), p_2(t), \ldots, p_n(t)\}$ is a configuration of robots at time t. The robots initially occupy distinct locations, i.e., $|P(0)| = n$.

The robots do not agree on the coordinate system, and each robot r_i has its own *x-y local coordinate system* denoted by $Z_i(t)$ such that the origin of $Z_i(t)$ is its current position.[4] We assume each local coordinate system is right-handed, and it has an arbitrary unit distance. For a set of points P (in Z_0), we denote by $Z_i(t)[P]$ the positions of $p \in P$ observed in $Z_i(t)$.

An algorithm is a function, say ψ, that returns a curve to the next location in the two-dimensional Euclidean plane when given a set of positions. Each robot has an independent private source of randomness and an algorithm can use it to generate a random rational number. A robot is *oblivious* in the sense that it does not remember past cycles. Hence, ψ uses only the observation in the Look phase of the current cycle.

In each Move phase, each robot moves at least $\delta > 0$ (in the global coordinate system) along the computed curve, or if the length of the curve is smaller than δ, the robot stops at the destination. However, after δ, a robot stops at an arbitrary point of the curve. All robots do not know this minimum moving distance δ. During movement, a robot always proceeds along the computed curve without stopping temporarily. We call this assumption *strict progress property*.

An execution is a sequence of configurations, $P(0), P(1), P(2), \ldots$. The execution is not uniquely determined even when it starts from a fixed initial configuration. Rather, there are many possible executions depending on the activation schedule of robots, execution of phases, and movement of robots. The *adversary* can choose the activation schedule, execution of phases, and how the robots move and stop on the curve. We assume that the adversary knows the algorithm, but does not know any random number generated at each robot before it is generated. Once a robot generates a random number, the adversary can use it to control all robots.

Pattern Formation. A target pattern F is given to every robot r_i as a set of points $Z_0[F] = \{Z_0[p] | p \in F\}$. We assume that $|Z_0[F]| = n$. In the following, as long as it is clear from the context, we identify $p \in F$ with $Z_0[p]$ and write, for example, "F is given to r_i" instead of "$Z_0[F]$ is given to r_i." It is enough emphasizing that F is not given to a robot in terms of its local coordinate system.

Let $\mathbb{T}$ be a set of all coordinate systems, which can be identified with the set of all transformations, rotations, uniform scalings, and their combinations. Let $\mathcal{P}_n$ be the set of all patterns of n points. For any $P, P' \in \mathcal{P}_n$, P is *similar* to P', if there exists $Z \in \mathbb{T}$ such that $Z[P] = P'$, denoted by $P \simeq P'$.

We say that algorithm ψ forms pattern $F \in \mathcal{P}_n$ from an initial configuration I, if for any execution $P(0)(= I), P(1), P(2), \ldots$, there exists a time instance t such that $P(t') \simeq F$ for all $t' \geq t$.

[3] In the ASYNC model, when no robot observes a configuration, the configuration does not affect the behavior of any robots. Hence, we consider the sequence of configurations, in each of which at least one robot executes a Look phase. In other words, without loss of generality, we consider discrete time $1, 2, \ldots$.

[4] During a Move phase, we assume that the origin of the local coordinate system of robot r_i is fixed to the position where the movement starts, and when the Move phase finishes, the origin is the current position of r_i.

For any $P \in \mathcal{P}_n$, let $C(P)$ be the smallest enclosing circle of P, and $c(P)$ be the center of $C(P)$. Formally, the *symmetricity* $\rho(P)$ of P is defined by

$$\rho(P) = \begin{cases} 1 & \text{if } c(P) \in P, \\ |\{Z \in \mathbb{T} : P = Z[P]\}| & \text{otherwise.} \end{cases}$$

We can also define $\rho(P)$ in the following way [6]: P can be divided into regular k-gons centered at $c(P)$, and $\rho(P)$ is the maximum of such k. Here, any point is a regular 1-gon with an arbitrary center, and any pair of points $\{p, q\}$ is a regular 2-gon with its center $(p + q)/2$.

For any configuration P ($c(P) \notin P$), let $P_1, P_2, \ldots, P_{n/\rho(P)}$ be a decomposition of P into the above mentioned regular $\rho(P)$-gons centered at $c(P)$. Yamashita and Suzuki [7] showed that even when each robot observes P in its local coordinate system, all robots can agree on the order of P_i's such that the distance of the points in P_i from $c(P)$ is no greater than the distance of the points in P_{i+1} from $c(P)$, and each robot is conscious of the group P_i it belongs to. We call the decomposition $P_1, P_2, \ldots, P_{n/\rho(P)}$ ordered by this condition the *regular $\rho(P)$-decomposition* of P.

A point on the circumference of $C(P)$ is said to be "on circle $C(P)$" and "the interior of $C(P)$" ("the exterior", respectively) does not include the circumference. We denote the interior (exterior, respectively) of $C(P)$ by $Int(C(P))$ ($Ext(C(P))$). We denote the radius of $C(P)$ by $r(P)$. Given two points p and p' on $C(P)$, we denote the arc from p to p' in the clockwise direction by $arc(p, p')$. When it is clear from the context, we also denote the length of $arc(p, p')$ by $arc(p, p')$. The largest empty circle $L(P)$ of P is the largest circle centered at $c(P)$ such that there is no robot in its interior, hence there is at least one robot on its circumference.

Algorithm with termination agreement. A robot is *static* when it is not in a Move phase, i.e., in a Look phase or a Compute phase, or not executing a cycle. A configuration is *static* if all robots are static. Because robots in the ASYNC model cannot recognize static configurations, we further define stationary configurations. A configuration P is *stationary* for an algorithm ψ, if in any execution starting from P, configuration does not change.

We say algorithm ψ guarantees termination agreement if in any execution $P(0), P(1), \ldots$ of ψ, there exists a time instance t such that $P(t)$ is a stationary configuration, in $P(t')$ ($t' \geq t$), ψ outputs $\emptyset$ at any robot, and all robots know the fact. Specifically, $\psi(Z'[P(t')]) = \emptyset$ in any local coordinate system Z'. This property is useful when we compose multiple algorithms to complete a task.

3 Randomized pattern formation algorithm

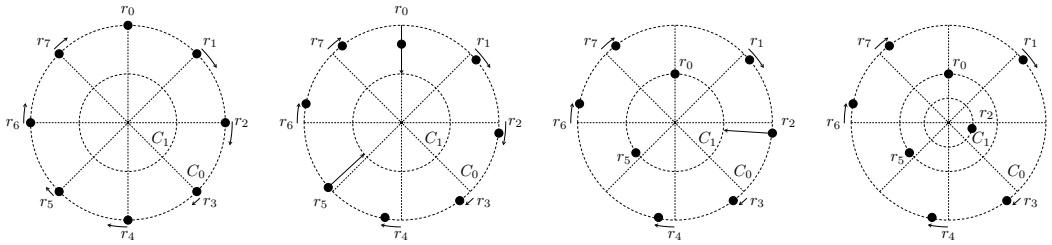
The idea of the proposed universal pattern formation algorithm is to translate a given initial configuration I with $\rho(I) > 1$ into a configuration I' with $\rho(I') = 1$ with probability 1, and after that the robots start the execution of a pattern formation algorithm. We formally define the problem.

Definition 1 The symmetricity breaking problem *is to change a given initial configuration I into a stationary configuration I' with $\rho(I') = 1$.*

In Section 3.1, we present a randomized symmetricity breaking algorithm ψ_{SB} with termination agreement. In the following, we assume $n \geq 5$ and I and F do not contain any multiplicities. Additionally, we assume that for a given initial configuration I, no robot occupies $c(I)$, i.e., $c(I) \cap I = \emptyset$.[5] Due to the page limitation, we omit the pseudo code of ψ_{SB}.

In Section 3.2, we present a randomized universal pattern formation algorithm ψ_{PF}, that uses ψ_{SB} and a pattern formation algorithm ψ_{CWM} [5] with slight modification.

[5] If there is a robot on $c(I)$, we move the robot by some small distance from $c(I)$ to satisfy the conditions of the terminal configuration of ψ_{SB}.



(a) Random movement (b) Selected robots proceed (c) Following robot proceeds (d) Single leader is selected

Figure 1: Random selection

3.1 Randomized symmetricity breaking algorithm ψ_{SB}

In the proposed algorithm ψ_{SB}, robots elect a single leader that occupies a point nearest to the center of the smallest enclosing circle. Clearly, the symmetricity of such configuration is one.

We use a sequence of circles to show the progress of ψ_{SB}. In configuration P, let $C_i(P)$ be the circle centered at $c(P)$ with radius $r(P)/2^i$. Hence, $C_0(P) = C(P)$. We denote the radius of $C_i(P)$ by γ_i. We call the infinite set of circles $C_0(P), C_1(P), \ldots$ the set of *binary circles*. Because ψ_{SB} keeps the smallest enclosing circle of robots unchanged during any execution, we use C_i instead of $C_i(P)$. We call C_i the *front circle* if C_i is the largest binary circle in $L(P)$ including the circumference of $L(P)$, and we call C_{i-1} the *backward circle*. We denote the number of robots in C_i and on C_i by n_i. Hence, if the current front circle C_i is the largest empty circle, n_i is the number of robots on C_i, otherwise it is smaller than the number of robots on C_i.

Recall that all local coordinate systems are right handed. Hence, all robots agree on the clockwise direction on each binary circle. For C_i ($i \geq 0$) and a robot r on C_i, we call the next robot on C_i in its clockwise direction *predecessor*, denoted by $pre(r)$, and the one in the counter-clockwise direction *successor*, denoted by $suc(r)$. When there are only two robots r and r' on C_i, $pre(r) = suc(r) = r'$. We say r is neighboring to r' if $r' = pre(r)$ or $r' = suc(r)$. For example, in Fig. 1(a), $pre(r_0)$ is r_1, $suc(r_0)$ is r_7, and r_1 and r_7 are neighbors of r_0.

During an execution of the proposed algorithm, robot r moves to an inner binary circle along a half-line starting from the center of the smallest enclosing circle and passing r's current position. We call this half-line the *radial track* of r. When r moves from a point on C_i to C_{i+1} along its radial track, we say r *proceeds to* C_{i+1}.

Algorithm ψ_{SB} first sends each robot to its inner nearest binary circle along its radial track if the robot is not on any binary circle. Hence, the current front circle is also the largest empty circle.

Then, ψ_{SB} probabilistically selects at least one robot on the current front circle C_i, and make them proceed to C_{i+1}. These selected robots repeat the selection on C_{i+1}. By repeating this, the number of robots on a current front circle reaches 1 with probability 1. The single robot on the front circle is called the *leader*.

We will show the detailed selection procedure on each front circle. We have two cases depending on the positions of robots when the selection of a front circle C_i starts. One is the *regular polygon case* where robots on C_i form a regular n_i-gon, and the other is the *non-regular polygon case* where n_i robots on C_i form a non-regular polygon.

Selection in the regular polygon case. When robots on the current front circle C_i form a regular n_i-gon (i.e., for all robot r on C_i, $arc(suc(r), r) = 2\pi\gamma_i/n_i$), it is difficult to select some of the robots. Especially, when the symmetricity of the current configuration is n_i, it is impossible to deterministically select some of the robots. In a regular n_i-gon case, ψ_{SB} makes these robots randomly circulate on C_i.

Then, a robot that do not catch up with its predecessor and caught by its successor is selected and proceeds to C_{i+1}.

First, if robot r on C_i finds that the robots on C_i form a regular n_i-gon, r randomly selects "stop" or "move." If it selects "move," it generates a random number v in (0..1], and moves $v(1/4)(2\pi\gamma_i/n_i)$ along C_i in the clockwise direction (Fig. 1(a)). This procedure ensures that the regular n_i-gon is broken with probability 1. When r finds that the regular n_i-gon is broken, r stops.

Uniform moving direction ensures the following invariants:

1. Once r finds that it is caught by $suc(r)$, i.e., the following inequality holds, r never leave from $suc(r)$.
$$Caught(r) = arc(suc(r), r) \leq 2\pi\gamma_i/n_i$$
2. Once r finds that it missed $pre(r)$, i.e., the following inequality holds, r never catch up with $pre(r)$.
$$Missing(r) = 2\pi\gamma_i/n_i < arc(r, pre(r)) \leq (5/4)(2\pi\gamma_i/n_i)$$

We say robot r is *selected* if it finds that the following predicate holds.

$$Selected(r) = Caught(r) \wedge Missing(r)$$

Then, a selected robot proceeds to C_{i+1} (Fig. 1(b)). Since no two neighboring robots satisfy *Selected* at a same time, while $Selected(r)$ holds at r, $suc(r)$ and $pre(r)$ wait for r to proceed to C_1. Even when $n_i = 2$, when they are not in the symmetric position, just one of the two robots becomes selected. Note that other robots cannot check whether r is selected or not in the ASYNC model because they do not know whether r has observed the configuration and found that $Selected(r)$ holds.

After some selected robots proceed to C_{i+1}, other robots might be still moving on C_i and may become selected later. However, in the ASYNC model, no robot can determine which robot is moving on C_i. For the robots on C_{i+1} to ensure that no more robot will join C_{i+1}, ψ_{SB} makes some of the non-selected robots on C_i proceed to C_{i+1}. The robots on C_i are classified into three types, rejected, following, and undefined.

The predecessor and the successor of a selected robot are classified into *rejected*, and each rejected robot stays on C_i. All robots can check whether robot r is rejected or not with the following condition:

$$\begin{aligned} &Rejected(r) = \\ &(arc(r, pre(r)) > (5/4)(2\pi\gamma_i/n_i)) \vee (arc(suc(r), r) > (5/4)(2\pi\gamma_i/n_i)). \end{aligned}$$

Non-rejected robot r becomes *following* if r finds that at least one of the following three conditions hold:

$$\begin{aligned} FollowPre(r) &= \neg Rejected(r) \wedge Rejected(pre(r)) \wedge Caught(r) \\ FollowSuc(r) &= \neg Rejected(r) \wedge Rejected(suc(r)) \wedge Missing(r) \\ FollowBoth(r) &= \neg Rejected(r) \wedge Rejected(pre(r)) \wedge Rejected(suc(r)). \end{aligned}$$

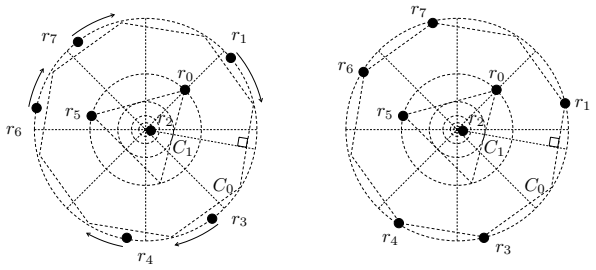
Hence, we have

$$Following(r) = FollowPre(r) \vee FollowSuc(r) \vee FollowBoth(r).$$

Intuitively, the predecessor and the successor of a following robot never become selected nor following. Algorithm ψ_{SB} makes each following robot proceed to C_{i+1} (Fig. 1(c)).

Finally, robots on C_i that are neither selected, rejected nor following are classified into *undefined*.

Note that $Rejected(r)$ implies $\neg Selected(r)$ and $\neg Following(r)$. Additionally, $Selected(r)$ and $Following(r)$ may hold at a same time.



(a) Embedding on C_0 (b) Terminal configuration

Figure 2: Stopping rejected robots when the leader is first generated on C_2. (a) The leader embeds a regular octagon on C_0 by its position on C_4. (b) After all robots C_0 have reached the corners of embedded polygons, r_L proceeds to C_5.

Eventually, all robots on C_i recognize their classification from selected, following, and rejected. We can show that once a robot finds its classification, it never changes. Then, selected robots and following robots leave C_i and only rejected robots remain on C_0. During the random selection phase, n_i does not change since robots moves in $Int(C_i) \cup C_i$. Hence, all robots can check whether a robot r on C_i is rejected or not with $Rejected(r)$, and the robots on C_{i+1} agree that no more robot proceeds to C_{i+1}. These robots start a new (random) selection on C_{i+1}.

Consider the case where $i = 0$. When $n = 5$, the length of the random movement is largest, and each robot circulates at most $\pi/10$. Hence, no two robots form a diameter. Additionally, ψ_{SB} guarantees that no two neighboring robots leave C_0. Hence, ψ_{SB} keeps C_0 during the random selection. In the same way, when $n \geq 5$, the random selection does not change C_0.

Selection for non-regular polygon case. When robots on the current front circle C_i does not form a regular n_i-gon, ψ_{SB} basically follows the random selection. Thus, robots do not circulate on C_i randomly, but check their classification with the three conditions.

Because robots do not form a regular n_i-gon on C_i, there exists a robot r on C_i that satisfies $arc(suc(r), r) < 2\pi\gamma_i/n_i$. However, there exists many positions of n_i robots on C_i where all such robot r are also rejected, i.e., $arc(r, pre(r)) > (5/4)(2\pi\gamma_i/n_i)$, from which no robot becomes selected nor following.

In this case, we add one more condition $NRSelected(r)$. We say r satisfies $NRSelected(r)$ when r is on the front circle C_i, all robots on C_i do not satisfy $Selected$ nor $Following$, and $arc(r, pre(r)) > (5/4)(2\pi\gamma_i/n_i)$ and $arc(suc(r), r) \leq 2\pi\gamma_i/n_i$ hold. We note that no two neighboring robots satisfies $NRSelected$. Robot r proceeds half way to C_{i+1}, and waits for all robots satisfying $NRSelected$ to proceed.[6] Robots in between C_i and C_{i+1} can reconstruct the non-regular polygon on C_i with their radial tracks and after all robots satisfied $NRSelected$ leaves C_i, the robots in $Ext(C_{i+1}) \cap Int(C_i)$ proceeds to C_{i+1}. Note that during a random selection, no robot on C_i satisfies $NRSelected$.

We consider one more exception case for initial configurations where robots form a non-regular polygon on C_0. In this case, each robot r first examines $NRSelected(r)$. If proceeding all robots satisfying $NRSelected$ changes C_0, the successor of such robot proceeds to C_1 instead of them. Assume that r is one of such robots satisfying $NRSelected(r)$. Because C_0 is broken after all robots satisfying $NRSelected$ proceeds, in the initial configuration $arc(r, pre(r)) = \pi\gamma_0$. Otherwise, there exists a rejected robot that does not satisfy $NRSelected$ in the initial configuration. Hence, proceeding $suc(r)$ does not change C_0.

After that, robots on C_i determine their classification by using $Rejected$, $Following$, and following robots proceed to C_{i+1}. Eventually all following robots leave C_i, and only rejected robots remain on C_i.

[6] Otherwise, r cannot distinguish how many robots satisfied $NRSelected$.

Termination agreement. By repeating the above procedure on each binary circle, with probability 1, only one robot reaches the inner most binary circle, with all other robots rejected (Fig. 1(d)). We say this robot is selected as a single leader. However, rejected robots may be still moving on the binary circles. Thus, the leader robot starts a new phase to stop all rejected robots, so that the terminal configuration is stationary.

Let r_L be the single leader and C_i be the front circle for $R \setminus \{r_L\}$ (this implies the leader is selected during the random selection on C_i). Intuitively, r_L checks the termination of C_{i-j} $(i-j \geq 0)$ when r_L is on C_{i+j+2}. Given a current observation, all robots on C_{i-j} are expected to move at most $(1/4)(2\pi\gamma_{i-j}/n_{i-j})$ from corners of some regular n_{i-j}-gon. Hence, there exists an embedding of regular n_{i-j}-gon onto C_{i-j} so that its corners does not overlap these expected tracks. If there is no such embedding, then randomized selection has not been executed on C_{i-j}, and r_L embeds an arbitrary regular n_{i-j}-gon on C_{i-j}. Robot r_L shows the embedding by its position on C_{i+j+2}, i.e., r_L's radial track is the perpendicular bisector of an edge of the regular n_{i-j}-gon (Fig. 2(a)).

Then, ψ_{SB} makes robots on C_{i-j} occupy distinct corners of the regular n_{i-j}-gon. The target points of these robots are determined by the clockwise matching algorithm [4]. We restrict the matching edges before we compute the clockwise matching. Specifically, we use arcs on C_{i-j} instead of direct edges, and direction of each matching edge (from a robot to its destination position) is always in the clockwise direction. Note that under this restriction, the clockwise matching algorithm works correctly on C_{i-j}. The robots on C_{i-j} has to start a new movement with fixed target positions. Because robots can agree the clockwise matching irrespective of their local coordinate systems, r_L can check whether robots on C_{i-j} finish the random movement.

Then, r_L calculates its next position on C_{i+j+3} in the same way for robots on C_{i-j-1}, and moves to that point.

The leader finishes checking all binary circles on C_{2i+2}, then it proceeds to C_{2i+3} to show the termination of ψ_{SB} (See Fig. 2(b)). However, ψ_{SB} carefully moves robots on C_0 to keep the smallest enclosing circle. When there are just two robots on C_0, then the random selection has not been executed on C_0, and r_L does not check the embedding. When there are more than three robots, there is at least one robot that can move toward its destination with keeping the smallest enclosing circle, and ψ_{SB} first moves such a robot.

For any configuration P satisfying the following two conditions, ψ_{SB} outputs $\emptyset$ at any robot irrespective of its local coordinate system. Hence, such configuration P is a stationary configuration of ψ_{SB}.

1. P contains a single leader on the front circle, denoted by C_b.
2. All other robots are in $Ext(C_k) \cup C_k$, satisfying $b \geq 2k + 3$.

Clearly, ψ_{SB} guarantees terminal agreement among all robots.

Algorithm ψ_{SB} guarantees the reachability to a terminal configuration with probability 1, and the terminal configuration is deterministically checkable by any robots in its local coordinate system.

3.2 Randomized pattern formation algorithm ψ_{PF}

We present a randomized pattern formation algorithm ψ_{PF}. Algorithm ψ_{PF} executes ψ_{SB} when the configuration does not satisfy the two conditions of the terminal configuration of ψ_{SB}. When the current configuration satisfies the two terminal conditions of ψ_{SB}, ψ_{PF} starts a pattern formation phase.

Fujinaga et al. proposed a pattern formation algorithm ψ_{CWM} in the ASYNC model, which uses the clockwise minimum weight perfect matching between the robots and an embedded target pattern [5]. The embedding of the target pattern is determined by the robots on the largest empty circle. Additionally, when there is a single robot on the largest empty circle, ψ_{CWM} keeps this robot the nearest robot to

the center of the smallest enclosing circle during any execution. We use this property to separate the configurations that appears executions of ψ_{SB} and those of ψ_{CWM}.

Algorithm ψ_{PF} uses ψ_{CWM} after ψ_{SB} terminates, however, to compose ψ_{SB} and ψ_{CWM}, we modify the terminal configuration of ψ_{SB} to keep the leader showing the termination of ψ_{SB}. Let P be a given terminal configuration of ψ_{SB}, and the single leader be r_L on the front circle C_L. Given a target pattern F, let $F_1, F_2, \ldots, F_{n/\rho(F)}$ be the regular $\rho(F)$-decomposition of F. Then, ψ_{CWM} embeds F so that $f \in F_1$ lies on the radial track of r_L, and $r(F) = r(P)$. When $c(F) \in F$, ψ_{CWM} also perturbs this target point. Let F' be this embedding.

Then, ψ_{PF} first moves r_L as follows: Let $L(F')$ be the largest empty circle of F' and $\ell(F')$ be its radius. Let k ($k > 0$) be an integer such that C_k be the largest binary circle in $L(F')$. If C_{2k+3} is in C_L, r_L proceeds to C_{2k+3}. When C_{2k+3} is in $Ext(C_L)$, r_L does not move. Then, ψ_{PF} starts the execution of ψ_{CWM}. After $R \setminus \{r_L\}$ reach their destination positions, r_L moves to its target point along its radial track.

Finally, we obtain the followin theorem.

Theorem 2 *For $n \geq 5$ robots, algorithm ψ_{PF} forms any target pattern from any initial configuration with probability* 1*.*

4 Conclusion

We present a randomized pattern formation algorithm for oblivious robots in the ASYNC model. The proposed algorithm consists of a randomized symmetricity breaking algorithm and a pattern formation algorithm proposed by Fujinaga et al. [5]. One of our future directions is to extend our results to the robots with limited visibility, where oblivious robots easily increase the symmetricity [8].

References

[1] Y. Dieudonné, F. Petit, and V. Villain, Leader election problem versus pattern formation problem. *Proc. of DISC 2010*, pp.267–281 (2010).

[2] P. Flocchini, G. Prencipe, N. Santoro, and P. Widmayer, Arbitrary pattern formation by asynchronous, anonymous, oblivious robots, *Theor. Comput. Sci.,* **407**, pp.412–447 (2008).

[3] N. Fujinaga, H. Ono, S. Kijima, and M. Yamashita, Pattern formation through optimum matching by oblivious CORDA robots, *Proc. of OPODIS 2010*, pp.1–15 (2010).

[4] N. Fujinaga, Y. Yamauchi, S. Kijima, and M. Yamashita, Asynchronous pattern formation by anonymous oblivious mobile robots, *Proc. of DISC 2012*, pp.312–325 (2012).

[5] I. Suzuki, and M. Yamashita, Distributed anonymous mobile robots: Formation of geometric patterns, *SIAM J. on Comput.*, **28**(4), pp.1347–1363 (1999).

[6] M. Yamashita, and I. Suzuki, Characterizing geometric patterns formable by oblivious anonymous mobile robots, *Theor. Comput. Sci*, **411**, pp.2433–2453 (2010).

[7] Y. Yamauchi, and M. Yamashita, Pattern formation by mobile robots with limited visibility, *Proc. of SIROCCO 2013*, pp.201-212, (2013).

Oscillatory Population Protocols

Colin Cooper[1] Colin.cooper@kcl.ac.uk
Anissa Lamani[2] lamani@csce.kyushu-u.ac.jp
Giovanni Viglietta [3] viglietta@gmail.com
Masafumi Yamashita[2] mak@csce.kyushu-u.ac.jp
Yukiko Yamauchi[2] yamauchi@inf.kyushu-u.ac.jp

[1] Department of informatics, Kings College, UK
[2] Graduation school of information science and electrical engineering, Kyushu University, Japan
[3] School of computer science, Carleton University, Ottawa, Canada

Keywords: Population protocols, Oscillatory behavior, Distributed algorithms, Molecular robots

Understanding how autonomy emerges in biological systems and applying it in giving artificial distributed systems autonomous properties motivate our study. Precisely, we focus on self-oscillations that play crucial roles in autonomous biological reactions, and investigate them as a phenomenon in distributed computing. Self-oscillations are often understood as a chemical oscillator provided, for example, by the Belousov–Zhabotinsky reaction. In biological systems, the oscillatory behavior is used as a natural clock to transmit signals and hence transfer information. In artificial distributed systems, self-oscillations could be used to distributely and autonomously implement a clock. This problem emerges in the project of designing molecular robots [2].

In our investigation, we use the population protocol (PP) model introduced by Angluin et al. [1]. PP is used as a theoretical model of a collection of finite-state mobile agents that interact with each other in order to solve a given problem in a cooperative fashion. Computations are done through pairwise interactions and the interaction pattern is unpredictable. PPs can represent not only artificial distributed systems as sensor networks and mobile agent systems, but also natural distributed systems such as chemical reactions and biological systems. We aim in our work at designing algorithms that make a given population exhibits an oscillatory behavior by itself whatever its initial state.

[1] D. Angluin, J. Aspnes, Z. Diamadi, M J. Fischer, and R. Peralta. Computation in networks of passively mobile finite-state sensors. In Proc of PODC, pp 290–299, 2004.
[2] S. Murata, A. Konagaya, S. Kobayashi, H. Saito, and M. Hagiya. Molecular robotics: A new paradigm for artifacts. New Generation Computing, 31(1):27–45, 2013.

オンライン木探索の最適性について

八神貴裕 * 山内由紀子 † 来嶋秀治 † 山下雅史 †
* 九州大学工学部電気情報工学科
† 九州大学大学院システム情報科学研究院

1 はじめに

多角形内の侵入者を探索者によって見つける問題に対して多くの研究が行われている．多角形内のすべての点が見えるようにカメラを配置する美術館問題，動き回る侵入者を視界の限られた移動しない探索者で発見するサーチライトスケジューリング問題や，移動できる探索者で侵入者を発見する多角形探索問題などがある [1]．また，このような探索問題は多角形内だけでなく，グラフ内を逃げ回る侵入者を探索者によって捕獲するグラフ探索問題も研究されてきた [3]．これらの探索問題の身の回りの応用例としては，建物内の監視を人間ではなくロボットに任せたい場合や，動物園から逃げ出した動物の捕獲などが挙げられる．

本稿では複数の探索者による多角形探索のオンラインアルゴリズムを考えるための準備として，まず多角形を単純化し木と見なした．複数の探索者によるオフラインの多角形探索はすでに結果が知られており，探索者数の上界を得る手段の一つとしてグラフ探索問題への帰着が行われている [2]．木は頂点と辺の集合であるが，ここでは平面上に埋め込まれた木を想定する．本稿では一人の探索者と，侵入者の移動を制限するためのバリケードをいくつか用いて，木内の侵入者を発見するオンラインアルゴリズムを提案し，その正当性を示す．

2 章では問題の定義，3 章ではアルゴリズムの紹介，4 章ではアルゴリズムの解析を行う．

2 諸定義

探索する対象となる木を $T = (V, E)$ とする．T は頂点集合 V と辺集合 E からなる連結で閉路を持たないグラフである．また，T の任意の頂点 $v \in V$ を根とした根付き木を $T(v)$ と表す．T の頂点，辺の初期状態を非クリアと呼ぶ．探索者がある頂点や辺にいるとき，その頂点や辺の状態をクリアと呼ぶ．探索者は侵入者の侵入を防ぐためのバリケードを持ち，今いる頂点にバリケードを設置することができる．非クリアな頂点や辺と接していて，探索者やバリケードが置かれていない辺や頂点は再び非クリアとなる．クリアであった頂点・辺が再び非クリアになることを再汚染と呼ぶ．

ある時刻において，すべての頂点・辺の状態がクリアであるとき探索成功とする．

探索者は以下の 5 個の基本動作を組み合わせて探索する．

- 辺を通過して接続する頂点に移動する．
- 頂点内で向きを変える．
- 今いる頂点にバリケードを設置する．
- 今いる頂点のバリケードを回収する．
- 今いる頂点のバリケードのメモリの読み出しと書き込み．

侵入者も探索者同様に辺を通過して接続する頂点に移動できる．しかし，侵入者は探索者やバリケードがいる頂点・辺には侵入できない．

また，本稿では次のように記号を定義する．r は探索開始時に探索者の持つバリケードの個数，b_i は i 番目に設置したバリケード，$v(b_i)$ は b_i が設置されている頂点とする．

本稿では根付き木 $T(v)$ のある任意の頂点 $u \in V$ の子である各頂点を根とした部分木を頂点 u に隣接する部分木と呼ぶことにする．

探索開始時，探索者はすべてのバリケード (r 個) を持っていて，与えられた任意の頂点 $v \in V$ から探索を開始する．

3 提案アルゴリズム

まずアルゴリズムのアイデアを紹介する．探索者は木 T の次数 3 以上の頂点にバリケードを設置する．T は 3 つ以上の部分木に分解さるので，探索者はそれぞれの部分木に対して同様にバリケードを設置し探索する．このような考え方はすでに [3] で用いられている．この考え方をもとにしたオンラインアルゴリズムを示す．このアルゴリズムの重要な部分はバリケードを設置するのに適した頂点を探索者自身で探すことである．

このアルゴリズムでは探索者と各バリケードにいくつかメモリを与えている．それらのメモリを紹介する．

探索者のメモリ

- $mode$ 探索者の動作や出力を決定する．アルゴリズム中では A, B, C の値をとるように記述しているが，それぞれ異なる整数を割り当ててもよいので整数型とする．3 つの値が区別できればよいのでサイズは 2bit.
- $flag$ それぞれのバリケードが最適な位置にいるか記録する配列．配列の大きさは r．バリケード b_i の $flag$ を $fag[b_i]$ と表す．-1, 0, 1 の 3 つの値を区別できればよいので整数型で各サイズは 2bit．全体で $2r$bit.
- $barr\_num$ 探索者が今持っているバリケードの個数．0 から r までの自然数が区別できればよいので整数型でサイズは $\lceil \log 2(r+1) \rceil$
- $deg1$ 次数 1 の頂点を訪れた回数を記録する．グラフの最高次数を調べるときのみ使われる．0 から 2 までの整数が記録できればよいので整数型でサイズは 2bit.

バリケードのメモリ

- $counter$ 未探索の部分木を順に正しく探索するための情報を記録する．T 内の頂点の最大次数を d とすると，0 から d までの整数が書き込めればよい．整数型でサイズは $\lceil \log_2(d+1) \rceil$bit.
- $name$ 設置されているバリケードを区別するためのメモリ．バリケードに 1 から r まで順番に整数で名前を付けるとしたら，整数型でサイズは $\lceil \log_2 r \rceil$bit.
- $failure$ 探索に失敗しバリケードに戻ってきたときの $coumter$ の値を書き込む．よって，型やサイズは $counter$ と同じ．

メモリを操作できるのは探索者だけである．つまり，探索中で探索者がいない頂点に設置されているのバリケードのメモリの値が変化することはない．

探索者の基本的な移動の仕方について説明する．このアルゴリズムでは頂点を部屋，辺を通路のようにとらえている．探索者には向きがあり，移動するときは進行方向を向いている．探索者が向きを変えるのは進行方向を変更するときのみである．探索者がある頂点 u から隣接する頂点 v に移動して，まだ向きを変えていないとき，探索者が 180 度向きを変えると辺 (u, v) がある向きを向く．また，この辺 (u, v) を入口となった辺と呼ぶ．次数 2 以上の頂点から隣接する頂点へ移動するとき，入口となった辺を向いている状態から，その場で反時計回りに向きを変え 1 つ目に見つけた辺に向かって進む．

メモリ $counter$ と探索者の動作について説明する．探索者はバリケード b_i を設置するとき，設置するバ

リケードの $counter$ に今いる頂点の次数を書き込む（ただし，アルゴリズム中で最初にバリケードを置いたときのみ次数より 1 だけ大きい値を書き込む）．探索者が隣接する部分木の探索のために $v(b_i)$ から移動する直前に $counter$ の値を 1 減らす．このとき探索者は，上で説明した移動方法で移動することで，隣接する部分木を順に探索できる．$v(b_i)$ に戻ってきたとき，$counter = 1$ であれば探索すべき部分木をすべて探索したことになる．

また，メモリ $name$ は探索者が本来操作すべきではないバリケードを操作しようとするのを防ぐためのものである．

オンラインアルゴリズムにおける真部分木の探索とバリケードの個数に関わる関数を定義する．$T(v)$ から v 以外の任意の次数 3 以上の頂点 $u \in V$ を選ぶ．u を根とした部分木を S とする．S 内の侵入者に S 以外の領域への経路を与えず，S の探索がバリケード k で十分であるとき，$m_{flag,S}(k) = \text{true}$ と表す．S 以外の領域の再汚染が一時的に生じるが，部分木 S と S から逃げ出した侵入者がいる可能性のある領域の探索が k 個で十分であるとき，$p_S(k) = \text{true}$ と表す．

木探索のオンラインアルゴリズムを GDB(Graph Decomposition by Barricade) と呼ぶことにし，main 関数を Algorithm 1 に示す．

main ではメモリの初期化，探索の中心となる関数 search(k) の呼び出し，結果の出力を行う．ここで用いる k はメモリの $barr\_num$ を表す変数である．4 行目，最大次数を調べる（次数 3 以上の頂点に移動）について補足をする．木 T の最大次数が 2 以下の場合，探索者はバリケードを持っていなくても探索を成功することができる．このときメモリ $deg1$ を使う．T の最大次数の調べ方は Algorithm 5 に示す．

ある部分木 S に対する search(k) の動作は $flag$ の値によって変化し，$flag = -1$ または 0 のときの関数 S_flagleq0 を，$flag = 1$ のとき関数 S_flag1 を呼び出す．また，S_flagleq0，S_flag1 は S 内の部分木に対して search($k-1$) を呼び出していて，再帰的探索を行うアルゴリズムである．search(k) を呼び出して，

Algorithm 1 GDB の main 関数

```
1: 各バリケードの flag ← 0
2: deg1 ← 0
3: mode ← A
4: T の最大次数を調べる（次数 3 以上の頂点に移動）
   最大次数が 2 以下なら mode = B になっている
5: if mode = A then
6:    バリケード b_1 を設置する
7:    //反時計回りに v(b_1) につながっている部分
      木を S_1, ..., S_n とする．
8:    for 1 ≤ i ≤ n do
9:       S_i に対して search(r − 1)
10:      if P_{S_i}(r − 1) = false then
11:         flag[b_1] ← −1
12:         b_1 を回収して S_i へ進む
13:         search(r)
14:         break
15:      else if P_{S_i}(r-1) = true かつ i = n then
16:         mode ← B
17:      end if
18:      i++
19:   end for
20: end if
21: //結果の出力
22: if mode = B then
23:    探索成功
24: else if mode = C then
25:    探索失敗
26: end if
```

search(k) が $mode = B$ で終了すれば $p_S(k) = \text{true}$，$mode = C$ で終了すれば $p_S(k) = \text{false}$ であることを意味する．search(r) の終了時の $mode$ の値によって木全体探索の結果がわかる．search(k) を Algorithm 2，S_flagleq0(k) を Algorithm 3，S_flag1(k) を Algorithm 4 に示す．

Algorithm 2 search(k)

```
1: //入力としてバリケードの個数を受け取る
2: mode ← A
3: failure ← 0
4: while mode = A do
5:    次数 3 以上の頂点に移動する
6:    バリケード b_{r−k+1} を設置する
7:    if flag[b_{r−k+1}] = −1 または 0 then
8:       S_flagleq0(k)
9:    else if flag[b_{r−k+1}] = 1 then
10:      S_flag1(k)
11:   end if
12: end while
13: flag[b_{r−k+1}] ← 0
14: b_{r−k+1} を回収する
```

Algorithm 3 S_flagleq0(k)

```
1: //入口となった辺から反時計回りに v(b_{r−k+1}) につながっている部分木を S_1, ..., S_n とする．
   （入口は含まない）
2: for 1 ≤ i ≤ n do
3:    S_i に search(k − 1)
4:    if P_{S_i}(k − 1) = false then
5:       flag[b_{r−k+1}] ← −1
6:       b_{r−k+1} を回収し，S_i の方へ進む
7:       break
8:    else if P_{S_i}(k − 1) = true かつ i = n then
9:       if flag = −1 then
10:         flag[b_{r−k+1}] ← 1
11:         b_{r−k+1}を回収し 非クリアな領域の方へ進む
12:      else if flag=0 then
13:         mode ← B
14:      end if
15:   end if
16:   i + +
17: end for
```

Algorithm 4 S_flag1

```
1: //入口となった辺から反時計回りに v(b_{r−k+1}) につながっている部分木を S_1, ..., S_n とする．
   （入口は含まない）
2: for 1 ≤ i ≤ n do
3:    S_i に search(k − 1)
4:    if P_{S_i}(k − 1) = false then
5:       if failure = 0 then
6:          failure ← i
7:       else if failure ≠ 0 then
8:          mode ← C
9:          break
10:      end if
11:   end if
12:   if i = n then
13:      if failure = 0 then
14:         mode ← B
15:      else
16:         b_{r−k+1} を回収し，S_{failure} の方へ進む
            (反時計回りに failure + 1 個目の辺)
17:      end if
18:   end if
19:   i + +
20: end for
```

Algorithm 5 最大次数を調べる

```
1: //次数 3 の頂点が見つかる前に異なる 2 つの次数 1 の頂点を見つければ T の最高次数は 2 以下である
2: while 今いる頂点の次数が 1 または 2，かつ deg1 < 2 do
3:    deg1 ← deg1 + 2 − (今いる頂点の次数)
4:    すでに説明した移動方法で移動する
5: end while
6: if 今いる頂点の次数が 0 または deg1 ≥ 2 then
7:    mode ← B
8: end if
```

4　アルゴリズムの解析

根付き木 $T(v)$ の v 以外の任意の次数 3 以上の頂点を選び u とする．u を根とする部分木を S とする．

S 内の u に隣接する部分木を $S_1, ..., S_l$ と表すと自然数 i を用いて，以下の 2 つの補題が成立する．

補題 1. S が部分木 $S_1, ..., S_l$ のうち $p_{S_i}(k-1) =$ false となる異なる S_i が 2 つ以上，または $m_{1,S_i}(k) =$ false となる S_i が 1 つ以上で構成されているならば $m_{1,S}(k) =$ false である．

補題 2. S が部分木 $S_1, ..., S_l$ のうち $p_{S_i}(k-1) =$ false となる異なる S_i が 3 つ以上で構成されているならば $p_S(k) =$ false である．

オンラインアルゴリズム GDB の正当性について定理 1 が成り立つ．

定理 1. GDB の出力は探索開始時の探索者の位置 $v \in V$ によって変化しない．

Proof. GDB で成功または失敗がわかったとき，探索中最後に b_1 を設置していた頂点を v とし，$T(v)$ を考える．v に隣接する部分木を $S_1, ..., S_l$ と表す．このアルゴリズムで探索が成功するときの木の構造は自然数 i, j を用いて，$p_{S_i}(r-1) =$ false かつ $m_{1,S_i}(r) =$ true となるような S_i は高々 1 つで，$\forall j (j \neq i), p_{S_j}(r-1) =$ true．このアルゴリズムで探索が失敗するときの木の構造には自然数 i を用いて，$p_{S_i}(r-1) =$ false となる異なる S_i が少なくとも 3 個存在する．

失敗したときの木の構造から任意に頂点を 1 つ選び u とし，$T(u)$ を考える．変形後の木には v を根とした部分木 S' が存在し，その S' 内には $p(k-1) =$ false の部分木が少なくとも 2 つ存在する．補題 1 より，S' は $m_{i,S'}(k) =$ false．よって，失敗したときの木はどの頂点を根としても探索ができない． □

オンラインアルゴリズム GDB と他の任意のオンライン，オフラインのアルゴリズムとの関係について定理 2 が成り立つ．

定理 2. GDB で n 個 のバリケードで T の探索に失敗するならば，どのようなオンライン，オフラインのアルゴリズムでも n 個 のバリケードで T の探索に失敗する．

Proof. 数学的帰納法で証明する．

$r = 0$ のとき，GDB を用いてバリケードなしで探索に失敗した木 T を任意に 1 つ選ぶ．T には次数 3 以上の頂点が含まれているので，どのようなアルゴリズムでもバリケードなしでは T の探索に成功しない．また T の任意の次数 1 の頂点を 1 つ選び v とする．v 以外の頂点からなる T の部分木を S とすると，$p_S(0) =$ false となる．どのようなアルゴリズムでもこの部分木はバリケードなしのとき探索に失敗する．

$r = k-1$ で命題成立を仮定し，$r = k$ のときを考える．GDB で $r = k$ で失敗した木 T を任意に選ぶ．T ある特定の頂点 u を根とする $T(u)$ を考え，u に隣接する部分木を $S_1, ..., S_l$ と表す．このとき，$p_{S_i}(k-1) =$ false のとなる異なる S_i が少なくとも 3 つ出現するような u が少なくとも 1 つ存在する．

T をバリケード k 個 で探索に成功するアルゴリズムが存在すると仮定する．$p_{S_i}(k-1) =$ false となる異なる S_i を 3 つ選び，一般性を失わず探索が終了する順に A, B, C とする．探索終了とは，ある領域のすべての頂点，辺がクリアになり，それ以降再汚染されないことをいう．探索過程において，ある領域 R の探索が終了する時刻を $f(R)$ と表す．また，R は A, B, C のどれかであるとき，$p_R(k-1) =$ false なので R 内にバリケードを k 個 設置している時間が存在する．その時間の中で最後の時刻を $e(R)$ と表す．このとき，

(1) $e(X) < e(Y) < f(A)$ (X, Y は A, B, C のいずれかで $X \neq Y$)

(2) $e(A) < f(A) < e(B)$

の 2 つの場合を考えれば十分である．

(1) について，時刻 $e(Y)$ で k 個のバリケードはすべて Y 内 に設置されている．このとき，X, Y 以外の領域から X 内への侵入者の経路ができるので，X 内はすべて再汚染される．定義より $e(X)$ 以降で X 内にバリケードを k 個設置することはないので，探索は成功しない．仮定に矛盾する．

(2) について，時刻 $e(B)$ で k 個のバリケードはすべて B 内に設置されている．このとき C から A へ

の侵入者の経路ができ，A 内はすべて再汚染される．探索は成功せず，仮定に矛盾する．

また $r = 0$ のときと同様に T の部分木を S とすると，$p_S(k) = \text{false}$ である． □

上の定理 2 の証明は，[3] ですでに証明されている以下の定理を参考にした．

定理 3. $k \geq 1$ において，木 T の探索が探索者 $k+1$ 人以上必要であるとき，かつそのときに限り，T には頂点 $v \in V$ の枝のうち探索者が k 人必要な枝が少なくとも 3 つ存在するような v が存在する．

ここで，頂点 v の枝 T' とは，v を次数 1 の頂点と見なしたときの極大の T の部分木 T' と定義されている．つまり，$T(v)$ において v の子のひとつを u とするとき，u を根とする部分木を $S' = (V', E')$ と表す．このとき v の枝 T' は $T' = (V' \cup \{u\}, E' \cup \{(v, u)\})$ と表すことができる．本稿のバリケードを探索者と見なすことで，定理 3 の k はこの問題におけるバリケードの個数（r 個）と探索者の人数（1 人）の和 $r + 1$ と見なすことができる．

5 まとめ

オンライン木探索において，バリケードの個数が他のどのようなアルゴリズムと同じ，もしくはそれ以下のアルゴリズムを得ることができた．今後は探索者やバリケードのメモリをより少なくすることや，多角形探索への応用を課題とする．

6 参考文献

参考文献

[1] 山下雅史 ,「捜索—移動する対象を探索する」, 室田一雄編,『離散構造とアルゴリズム III』, 近代科学社, pp. 115-162, 1994.

[2] M. Yamashita, H. Umemoto, I. Suzuki and T. Kameda, “Searching for Mobile Intruders in a Polygonal Reagion by Group of Mobile Searchers”, Algorithmica, pp. 208-236, 2001.

[3] T. D. Parsons, “Pursuit-evasion in a graph” , In *Proceedings of the Theory and Applications of Graphs*, pp. 426-441, 1976.

共通座標系を持たない5台の非同期式ファットロボットによる集合問題について

平野　拓弥[1]　片山 喜章[1]　和田　幸一[2]

概要：本稿では，連続平面上において大きさを持つ 5 台の自律分散ロボットによる集合問題を扱う．ロボットは共通の座標系を持たず，匿名で，通信を行わない．透明で視界距離に関しての制限を持たず，非同期に動作する．ロボットは他のロボットの視界を遮らないが，移動時の障害物となりうるため，衝突を防ぐアルゴリズムの設計が重要となる．同モデルにおける n 台の集合問題に対するアルゴリズムはすでに提案されているが [8]，このアルゴリズムには問題点が存在する．そこで，我々は台数を 5 台に限定することで文献 [8] とは異なる手法で問題を解決した．

1. はじめに

近年，自律分散ロボット群の研究が盛んに行われている．自律分散ロボットとは自律分散システムの一つで，低機能なロボットを複数台用いて，協調して動作させることでロボット群全体で一つの目的を達成させるシステムである．ロボット能力を低く抑えることで，コストの低く抑え，システムの拡張を容易にできるといった利点がある．また，それぞれのロボットが自律的に動作するため，人による制御が難しい場所でも安定して動作することができる．研究の焦点はロボットの持つ情報や能力と問題の可解性との関係であり，より低機能なロボットで問題を解くことが目標とされている．

ロボットは他のロボットの位置を観測し (Look)，観測した情報を用いてアルゴリズムに従って移動先を計算し (Compute)，計算した移動先に移動する (Move)，というサイクルを繰り返して問題を解決する．全てのロボットは同じアルゴリズムで動作する．よく扱われる問題として，一点集合問題が挙げられる．一点集合問題とは，あらかじめ決められていない一点に全てのロボットを集合させる問題である．

集合問題の中でも，2 台のロボットの集合を考えた問題を特にランデブーと呼ぶ．ランデブーは特別な能力がなければ集合を達成できないことは広く知られており，コンパスやメモリを有したロボットを用いて研究が行われている [2][3]．また，3 台以上のロボットによる集合問題をギャザリングと呼ぶ．こちらは，重複検知能力を持たせることで解決可能であると証明されている [4]．

純粋に理論的な研究は，ロボットを平面上を動く点として表現してきた．一方で，現実のロボットシステムへの適用を考える場合，より現実的なモデルの導入が必要である．そこでロボットの持つコンパスやセンサの誤差を考慮したモデル [2] や，ロボットの視野に制限を設けたモデル [5] などの研究も行われている．さらに，現実のロボットは大きさを持つため，ロボットが他のロボットの観測および移動について障害物となりうる．そこで，ロボットを点ではなく円盤で表したモデルであるファットロボット (fat robot) も提案されており，ファットロボットに関しても研究が進められている．ファットロボットは大きさを持ち，お互いに重なりあうことができないため，2 台以上のロボットが同一点を共有することは不可能である．よって，集合の定義についても再考慮する必要がある．

文献 [6] は最初にファットロボットの集合問題を扱った論文である．ロボットが 3 台または 4 台の場合についてのアルゴリズムを提案している．また，文献 [7] はキラリティを持つ 5 台以上のファットロボットに関しての集合問題を取り扱っている．どちらの論文も，ロボットは他のロボットの視界を遮る (不透明)，という条件の元でアルゴリズムを設計しており，集合の条件は「全てのロボットが連結であり，互いに見ることができる」というものである．文献 [10] では，離散平面上でのファットロボットの集合問題に関しての研究が行われている．

我々はキラリティを持たず，ロボットが他のロボットの

[1] 名古屋工業大学大学院工学研究科情報工学専攻
Nagoya Institute of Technology, Graduate School of Computer Science and Engineering
[2] 法政大学理工学部応用情報工学科
Hosei University, Faculty of Science and Engineering Department of Applied Informatics

視界を遮らない，つまりロボットを透明と仮定したモデルについてのアルゴリズムを提案する．我々と同様のモデルにおいて，任意の台数のロボットの集合問題に関する研究が文献 [8] でなされているが，この論文中で提案されているアルゴリズムは二つの問題点がある．一つ目は，最初に集合の中心となる部分に 1 台のロボットが移動する際に，他のロボットに移動を妨げられる場合を考慮していない点．二つ目は，最後の数台が集合する際にロボットの配置が対称となる可能性があり，移動するロボットが一意に決定できず，想定通りの動作を実行できない点である．

我々は，ロボットの台数を 5 台に限定することで，文献 [8] とは異なる手法を用いて集合を達成した．

2. モデルと問題定義

2.1 ロボットのモデルと仮定

5 台のファットロボットの集合を $R = \{r_1, r_2, ..., r_5\}$ で表す．ロボットは半径 1 の単位円であり，連続二次元平面上を自由に移動する．このとき，ロボットには大きさがあるため，移動時に他のロボットと衝突しうる．また，ロボットは任意の直線もしくは円周に沿って移動ができるものとする．ロボットの移動は目的地まで到達する前に任意の位置で停止しうるが，少なくとも最低移動距離 Δ 以上は移動できるとする．なお，ロボットの位置はロボットを表現する単位円の中心点と考える．したがって，例えば互いに接する 2 台のロボット間の距離は 2 である．

ロボットは匿名で，外観や識別子等によって他のロボットと区別することはできない．また，他のロボットと直接通信を行うこともできない．自身の持つカメラやセンサ等を用いて他のロボットの位置を観測することでのみ，他のロボットの情報を得ることができる．

ロボットは共通の座標系 (軸，原点，キラリティ等) に関する一切の知識を持たないが，ロボットの半径が同一なため，単位長についての合意は持っているとする．ロボットは観測した他のロボットの位置情報を自身のローカル座標系に当てはめ，その情報をもとに移動先を計算し，移動する．ロボットは他のロボットのローカル座標系について知ることはできない．

ロボットは透明であり，他のロボットの視界を妨げない．また，視野範囲や視界距離は無限であり，他のロボット全ての位置を正確に観測できる．ロボットは過去の動作を記憶できず，常に最新の観測結果のみに基づいて行動する．また，全てのロボットは同一のアルゴリズムで動作する．

ロボットは Look，Compute，Move の 3 つのフェイズから成るサイクルを繰り返して問題を解決する．Look フェイズでは，他のロボットの位置を自身の持つカメラやセンサ等で観測する．Compute フェイズでは，観測した情報をもとにアルゴリズムを実行し，自分の移動先を決定する．Move フェイズでは，Compute フェイズで決定した移動先に向かって移動する．サイクルを実行するタイミングについては 3 つのモデルが提案されており，それぞれ完全同期 (FSYNC)，半同期 (SSYNC)，非同期 (ASYNC) と呼ぶ．

図 1 dns=7 のときの図

FSYNC モデルでは，全てのロボットが完全に同じタイミングでサイクルの各フェイズを実行する．SSYNC モデルでは，FSYNC と同様に各ロボットは同じタイミングでサイクルを実行するが，その際に動作しないロボットが存在する．つまり，ロボットの部分集合が完全に同じタイミングで動作する．ASYNC モデルでは，ロボットのサイクルの実行に関して一切の仮定を置かない．よって，例えばあるロボットが Compute しているタイミングで別のロボットが Move をしていると，古い位置情報に従って移動先を決定し移動する可能性がある．本研究では，ASYNC モデルを仮定する．

また，初期状況として，接し合っている (=距離が 2 である) ロボットは存在せず，全てのロボットは静止しているものとする．

2.2 問題定義

ロボット間の距離が 2 である組の数を密度といい，dns で表す．密度が最大である状態で集合することを，本研究での達成目標とする．ロボットの台数が 5 台の場合について，密度が最大となる状態を**図 1** に示す．図からわかる通り，ロボットの台数 n が 5 のとき，密度の最大値は $dns = 7$ である．

定義 2.1. 集合問題 : 5 台のロボットにおいて，dns=7 の状態で全てのロボットが静止したとき，ロボットは集合したという．

2.3 諸定義

ロボット r 上で実行されるアルゴリズムで用いる変数や関数などを以下に定義する．

- $GatheringSet$: 互いに接し合っているロボットの集合．
- L_1 : $dns = 1$ のとき，$GatheringSet$ に含まれる 2 台のロボットの接点を通る共通接線 (**図 2**).
- L_2 : $dns = 1$ のとき，$GatheringSet$ に含まれる 2 台のロボットの中心を通る直線 (図 2).
- L_3 : $dns = 2$ であり，$GatheringSet$ に含まれる 3 台が一直線上に並んでいるとき，その 3 台の中心を通る

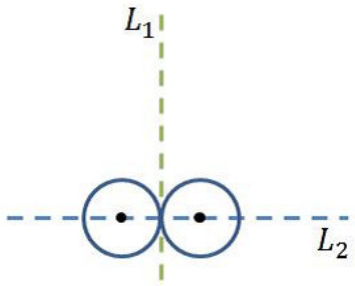


図 2 L_1 と L_2

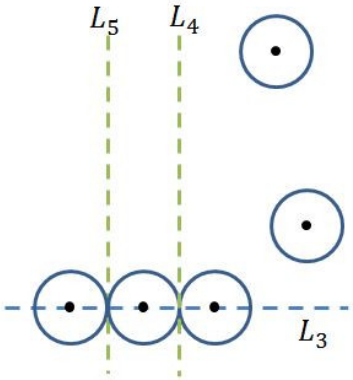


図 3 L_3 と L_4 と L_5

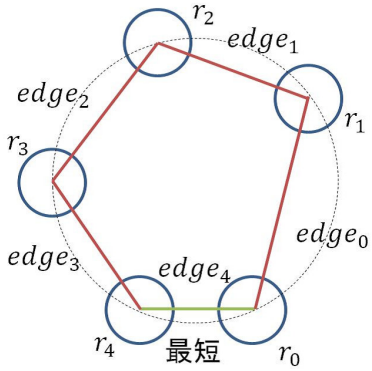


図 4 同一円周上に 5 台のロボットが存在する場合の例

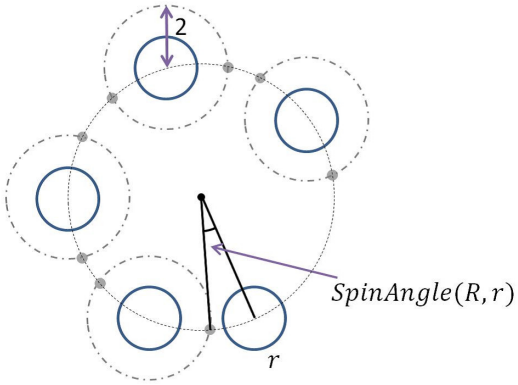


図 5 $SpinAngle(R,r)$ の例

直線 (**図 3**).

- L_4, L_5 : $dns = 2$ であり，$GatheringSet$ に含まれる 3 台が一直線上に並んでいるとき，接し合っているロボットの組それぞれの共通接線のうち，$GatheringSet$ に含まれない 2 台のうち，L_3 により近い方ロボットに対して，近い方の直線を L_4，遠い方の直線を L_5 という (図 3).
- $dist(r, L_i)$: ロボット r と直線 L_i との距離.
- $NotOnCircle(R)$: 5 台全てのロボットが同一円周上に存在する場合には false，それ以外の場合 true となる boolean 関数.
- $Pent(R)$: 同一円周上に 5 台のロボットが存在しているとき，その 5 台による凸五角形.
- $Edge(r_i)[4]$: $Pent(R)$ の最短辺を成すロボット r_i の，最短辺の逆方向周りの 4 本の辺の長さを自分に近い順に格納する配列.
  図 4 の例において，$Edge(r_0)$ は
  $Edge(r_0)[0] = edge_0, Edge(r_0)[1] = edge_1,$
  $Edge(r_0)[2] = edge_2, Edge(r_0)[3] = edge_3$
  である．同様に，$Edge(r_4)$ は
  $Edge(r_4)[0] = edge_3, Edge(r_4)[1] = edge_2,$
  $Edge(r_4)[2] = edge_1, Edge(r_4)[3] = edge_0$
  である．ただし，$edge_i$ はそれぞれの辺の長さであり，最短辺を成さないロボット r_i の $Edge(r_i)$ は NULL とする.
- SS : $Pent(R)$ の最短辺を成すロボットの集合.
- $SpinAngle(R, r)$: 5 台のロボットが同一円周上に存在するとき，その円周上において，$r_i \in R - \{r\}$ からの距離が 2 である点の中で r に最も近い点と，r が成す角度 (**図 5**).
- $Spin(angle)$: 同一円周上に 5 台のロボットが存在するとき，その円周上に沿って角度 angle だけ最も近いロボットの方向へ移動する命令.
- $Gather()$: L_2 上かつ，ロボット $r' \in GathringSet$ と接する点のうち，最も近い点へ直線移動する命令.
- $Slide(length, L_i, dir)$: 距離 $length$ だけ，直線 L_i の垂直方向に，dir 向き (far : L_i から遠ざかる，もしくは $near$: L_i に近づく) に直線移動する命令.

3. アルゴリズム

3.1 基本戦略

アルゴリズムの基本的な戦略は以下の通りである.

(1) 全てのロボットを同一円周上へ移動させる
(2) 同一円周上にある 5 台の中から 1 台を移動させ，2 台を接触させる
(3) $GatheringSet$ に含まれていない 3 台のうちの 1 台を，接触している 2 台のロボットと一直線になるように接触させる

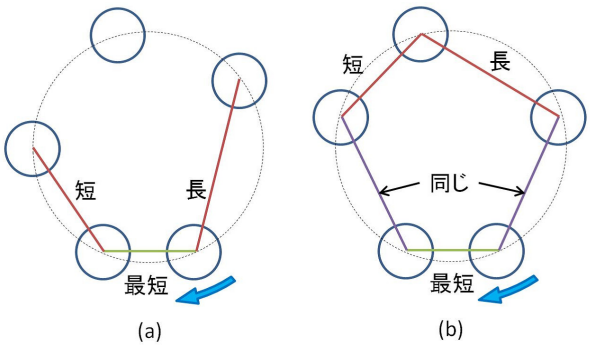


図 6　dns=0 の場合に移動するロボット

(4) 残り 2 台のロボットを順番に集合させる

全てのロボットを最初に同一円周上へ移動させることで，初期状況に関わらずその後のアルゴリズムの設計や正当性の証明を考えることができる．同一円周上へ全てのロボットが移動した後は，移動時のロボット同士の衝突を防ぐため，1 台のロボットのみが移動するようにアルゴリズムを設計する．1 台のロボットのみを移動させるためには，対称性の発見とその解消が重要となる．

3.1.1　同一円周上への移動

同一円周上への移動に関しては，円形成アルゴリズムを利用することで容易に達成することができる．文献 [9] において，本研究と同じモデルで円形成問題を解決するアルゴリズムが提案されているため，そちらを参照する．$dns = 0$ かつ 5 台のロボットが同一円周上に存在しない場合に限り，円形成を行う．

3.1.2　2 台の接触

$dns = 0$ かつ，同一円周上に 5 台のロボットが存在しているとき，1 台がその円周に沿って移動し，2 台が接触している状況を作り出す．移動するのは，同一円周上にある 5 台で凸五角形を計算し，その最短辺を構成するロボットのうちのいずれかとする．このとき，ロボットが移動することで，移動する条件に合致するロボットが変化しないように注意する必要がある．ASYNC モデルではロボットが移動中に他のロボットによって観測されうるため，もしロボットの移動中に，そのロボットの移動によって条件に合うロボットが変化した場合，複数のロボットが移動し，結果的に不都合な状況が生じる可能性がある．それを防ぐため，最短辺を構成するロボットのうち，最短辺と逆側の辺の長さが最も大きいロボットが移動することとする (**図 6**-a)．もしこの値が等しい場合，最短辺の逆側の辺を更に 2 本目，3 本目と比較していき，辞書式順で最大となるロボットが移動する (図 6-b)．

このとき，ロボットの配置が対称 (線対称・回転対称) の場合，移動するべき 1 台のロボットを一意に決定することができない可能性がある．ただし，今回は台数を 5 台に限定したため，回転対称となるのはロボットの配置が正五角形になっている場合のみであり，この場合は集合の達成は

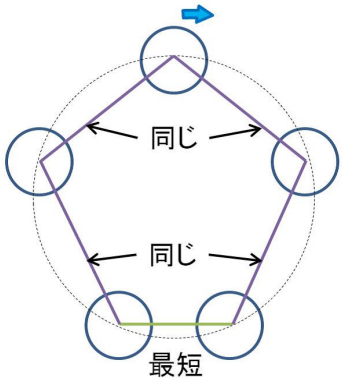


図 7　dns=0 かつ対称の場合に移動するロボット

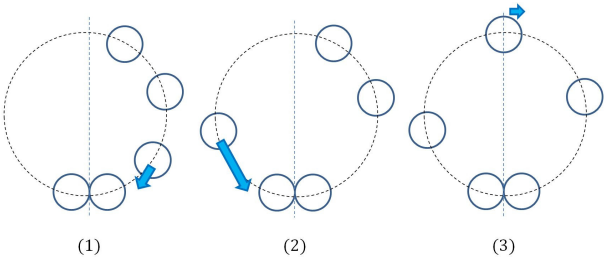


図 8　dns=1 の場合の 3 つのパターン

不可能とする．また，線対称については対称軸上に 1 台のロボットが存在し，残りの 4 台のロボットが対称軸を挟んで 2 台ずつ，対称の位置に存在する場合に限られる．よって，対称軸上のロボットが一意に決定され，そのロボットが移動することで，対称性を崩すことが可能である．対称軸上のロボットは，任意の方向に円周に沿って微小距離移動する (**図 7**)．ただし，微小距離を移動する途中で再び対称となる場合は，対称となるポイントの直前で停止することで対称となるのを防ぐ．非対称な形状であれば動くべき 1 台を一意に決定することが可能となる．

3.1.3　3 台目の移動

一組のロボットが接触した後，その 2 台のロボットによって定義される直線 (L_1, L_2) を基準として残りの 3 台を順番に集合させる．*GatheringSet* に含まれていない 3 台のロボットの中から 1 台のロボットが移動するが，このとき，L_1 を基準として，3 台のロボットの配置については 3 つのパターンが考えられる (**図 8**)．

(1) 3 台のロボットが L_1 を基準にして同じ側に存在する

(2) L_1 によって 2 台のロボットと 1 台のロボットに分けられる

(3) L_1 上に 1 台のロボットが存在する

この時点では全てのロボットは同一円周上に存在するため，その円の直径である L_1 上に複数のロボットが存在することはない．また，パターン 3 については，L_1 上に位置するロボットが L_1 に対して垂直方向に，任意の向きに移動することで，パターン 1 かパターン 2 に推移することができる．

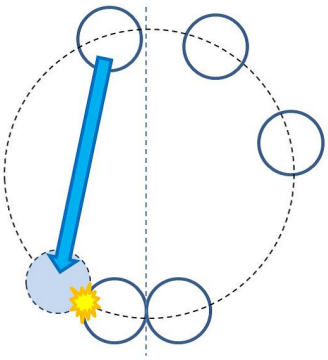


図 9　衝突が発生する例

パターン 1 の場合については，3 台のうち，L_2 に最も近いロボットが移動し集合する．移動先は L_2 上の，$GatheringSet$ に含まれるロボットに接することができる点のうち，自分に近い方の点である．このとき，全てのロボットは同一円周上にいるため他のロボットに移動を妨げられることはない．(図 8-1)

パターン 2 の場合については，L_1 を基準にして 1 台のみが存在する側のロボットが移動する．この場合も移動先はパターン 1 と同様である．ただし，ロボットと L_1 との距離が 3 未満の場合，他のロボットと衝突する可能性がある (**図 9**)．よって，その場合は L_1 との距離が 3 となる位置へ移動してから集合することで解決する．

このように移動することで，3 台が一直線上 (L_3 上) に並んだ状態で，残りの 2 台は L_3 からの距離が異なる状況を作り出すことができる．よって，これを利用して残りの 2 台を順序付けすることができる．

3.1.4　4 台目と 5 台目の集合

残り 2 台を集めるとき，L_3 から遠いロボットから移動する．もし L_3 からの距離が 4 未満の場合，L_3 から 2 離れる．その後，L_5 上へ移動する．L_3 に近い方のロボットは，遠い方のロボットが L_5 上に位置しないとき静止しつづける (**図 10**-a)．遠い方のロボットが L_5 上に位置する場合は，L_4 上へ移動する (図 10-b)．このとき，他のロボットに移動を妨げられる場合 (L_3 との距離が 2 未満の場合) は，L_3 から 2 離れることで解決する．このとき，遠い方のロボットと L_3 との距離は必ず 4 以上であるので，近い方のロボットと遠い方のロボットの L_3 に対しての位置関係が変わることはない．

L_3 からの距離が遠いロボットが L_5 上に，近い方のロボットが L_4 上に乗っている場合，L_3 に近いロボットが先に L_4 に沿って集合する．最後の 1 台は，自分以外の全てのロボットが集合している状態でのみ，集合するために移動する．

以上の基本戦略を実現したアルゴリズムを**図 11**，**図 12** に示す．

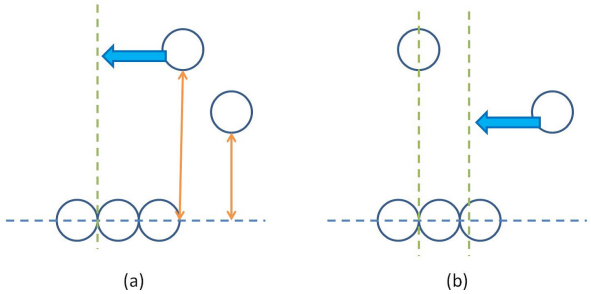


図 10　dns=2 の場合に移動するロボット

3.2　正当性の証明

アルゴリズムを実行することで，有限時間内に 5 台のロボットが dns=7 の状態で静止することを示す．

補題 3.1. dns=0 かつ $NotOnCircle(R) = true$ のとき，有限時間内に dns=0 かつ $NotOnCircle(R) = false$ となる

証明 1. 仮定より，初期状況では接しているロボットは存在しないため，dns=0 である．アルゴリズムより，dns=0 かつ $NotOnCircle(R) = true$ のとき，ロボットは文献 [9] に示された円形成アルゴリズムを実行する．円形成アルゴリズムが有限時間内に完了することは，文献 [9] による．

よって，dns=0 かつ $NotOnCircle(R) = true$ のとき，全てのロボットは有限時間内に同一円周上へ移動する．つまり，有限時間内に dns=0 かつ $NotOnCircle(R) = false$ となる．□

補題 3.2. dns=0 かつ $NotOnCircle(R) = false$ かつ $Pent(R)$ が正五角形でないとき，有限時間内に dns=1 になる

証明 2. $NotOnCircle(R) = false$ であるとき，5 台のロボットは凸五角形を成している ($Pent(R)$)．$Pent(R)$ が非対称の場合，各ロボットが自分から見て時計回り・反時計回りに $Pent(R)$ の辺を並べたとき，その系列は必ず異なる．ロボットは他の全てのロボットの位置情報を持つため，他ロボットから見た時計回り・反時計回りの辺の系列も計算することが可能で，各系列はそれぞれ異なっている．

Require: ロボットの集合 R, ロボット r
Ensure: Boolean 変数 (true,false)
  $Pent(R)$ の最短辺を成すロボットの集合 SS を調べる
  if r が成す辺が最短辺 **then**
    $j \Leftarrow 0$
    while $r \in SS$ **do**
      $\forall r_i \in SS$ について，$Edge(r_i)[j]$ の値を比較
      if $Edge(r)[j]$ が唯一の最大値 **then**
        return(true)
      end if
      $Edge(r_i)[j]$ が最大値でない r_i を SS から取り除く
      $j++$
    end while
  end if
  return(false)

図 11　CheckMove(R,r)

よって，自分を含めた全てのロボットが持つ系列を比較可能であり，それを用いてロボットに順番づけをし，全ロボットが共通の 1 台を選択することができる．アルゴリズムより，$Pent(R)$ の最短辺を成すロボットは，最短辺を成す全てのロボットの $Edge(r_i)$ の値を計算・比較することで，移動するべき条件を満たす 1 台のロボットを一意に決定し，そのロボットのみが最短辺の向きに移動する．よって，最短辺が複数あった場合でも，$Pent(R)$ が非対称ならば移動するべき唯一のロボットが決定可能であり，さらに移動の最中にも移動する条件を満たすロボットは変化しない．また，移動は 5 台のロボットが乗っている円周に沿って行われ，移動距離は有限なので，有限時間内に 2 台のロボットの距離が 2 となる．よって，5 台のロボットが成す $Pent(R)$ が非対称の場合，1 台のロボットのみが移動し，有限時間内に dns=1 になる．

また，$Pent(R)$ が線対称でありかつ正五角形でない場合，その形状は 1 台のロボットのみが対称軸上に存在する線対称に限られる．よって，対称軸上のロボットが円周に沿って任意の方向へ微小距離移動することで，形状を非対称にすることができる．

以上から，dns=0 かつ $NotOnCircle(R) = false$ かつ $Pent(R)$ が正五角形でないとき，有限時間内に dns=1 になる． □

補題 3.3. dns=1 のとき，有限時間内に dns=2 になる

証明 3. アルゴリズムにより，dns=1 になった瞬間，全てのロボットは同一円周上に存在する．よって，このときのロボットの配置は以下の 3 パターンに分けられる．

(1) 3 台のロボットが L_1 を基準にして同じ側に存在する
(2) L_1 によって 2 台のロボットと 1 台のロボットに分けられる
(3) L_1 上に 1 台のロボットが存在する

どのパターンについても，移動するべきロボットを一意に決定可能なのは明らかである．パターン 1 については，L_2 まで最も近いロボットについて目的地まで障害物が存在せず，直接移動可能なのは明らかであり，また移動の最中でも移動する条件を満たすロボットは変化しない．パターン 2 については，目的地まで直接移動できない可能性はあるが，L_1 から離れる向きに移動することでそれを解決できる．その際にも移動する条件を満たすロボットは変化せず，また，目的地まで移動できるのは明らかである．パターン 3 については，L_1 上のロボットが微小距離移動することでパターン 1 かパターン 2 に移行する．どの場合においても移動距離は有限であるため，移動が有限時間内に終わることは明らかである．

以上から，dns=1 のとき，有限時間内に dns=2 になる． □

補題 3.4. dns=2 のとき，有限時間内に dns=4 となる

証明 4. dns=1 のとき L_1 上にロボットがいない場合，dns=2 になった時点で残り 2 台が L_3 からの距離が異なるのは明らかである．また，L_1 上にロボットが存在する場

Require: ロボットの集合 R, ロボット r
Ensure: ロボットの移動先 T
```
if r ∈ GatheringSet then
  exit
else if R が正五角形 then
  exit
else if dns = 0 then
  if NotOnCircle(R) then
    CircleFormation(R, r)
  else if R が対称 then
    if dist(r, L0) = 0 then
      Spin(δ)
    end if
  else if CheckMove(R, r) then
    Spin(SpinAngle(R, r))
  end if
else if dns = 1 then
  if パターン 3 then
    if (dist(r, L1) = 0) then
      Slide(δ, L1, far)
    end if
  else if パターン 1 then
    if GatheringSet に含まれていない 3 台に関して，L1 を基準として自分と同じ側にロボットが存在しない then
      if dist(r, L1) < 3 then
        Slide(3 − dist(r, L1), L1, far)
      else
        Gather()
      end if
    end if
  else if GatheringSet に含まれていない 3 台のうち，自分が最も L2 に近い then
    Gather()
  end if
else if dns = 2 then
  GatheringSet に含まれていない 2 台のうち，自身を r，もう 1 台を r′ とする
  if dist(r, L3) > dist(r′, L3) then
    if dist(r, L3) < 4 then
      Slide(2, L3, far)
    else
      Slide(dist(r, L5), L5, near)
    end if
  else
    if dist(r′, L5) = 0 then
      if dist(r, L4) = 0 then
        Slide(dist(r, L3) − √3, L3, near)
      else if dist(r, L3) < 2 then
        Slide(2, L3, far)
      else
        Slide(dist(r, L4), L4, near)
      end if
    end if
  end if
else if dns = 4 then
  Slide(dist(r, L3) − √3, L3, near)
end if
```

図 12 GatheringRobots(R,r)

合でも，L_1 上のロボットは L_1 に垂直に移動するため，L_1 上のロボットが移動した場合でも残り 2 台の L_3 からの距離は異なる．よって，L_3 からの距離を用いて移動するべき 1 台を一意に決定することは可能である．

L_3 に近い方のロボットは，GatheringSet に含まれるロボットに移動を妨げられる可能性があるため，それを防ぐために L_3 に対して垂直方向へ移動する必要がある．しかし，近い方のロボットのみが移動すると，遠い方のロボットとの位置関係が逆転する可能性があるため，位置関係が逆転する可能性がある場合は遠い方のロボットも L_3 から垂直方向に遠ざかる．よって位置関係は逆転しない．

遠い方のロボットが L_5 上へ移動するとき，移動上に障害物は存在しない．同様に近い方のロボットが L_4 上へ移動するときにも障害物は存在しない．また，L_4 上へ移動したロボットが集合しようとするとき，その経路に障害物が存在しないことは明らかである．よって，dns=2 のとき，有限時間内に dns=4 となる． □

補題 3.5. dns=4 のとき，1 台のロボットのみが移動し，有限時間内に dns=7 となる

証明 5. dns=4 のとき，4 台が集合している状態で，最後の 1 台は集合地点まで直線移動するだけで dns=7 の状態で集合を達成する．この経路には障害物は存在せず，その移動距離は有限である．また，その間他のロボットは移動しない．よって，dns=4 のとき，有限時間内に dns=7 となる． □

定理 3.1. アルゴリズムは，有限時間内に 5 台のロボットを $dns = 7$ の状態で集合させる．

証明 6. 補題 3.1，補題 3.2，補題 3.3，補題 3.4，補題 3.5 による． □

4. おわりに

本稿では，連続平面上において大きさを持つ 5 台の自律分散ロボットによる集合問題を解決するアルゴリズムを提案し，その正当性を証明した．

今後の課題として，同モデルにおける $n \geq 6$ の場合や，共通座標系を持たない 5 台以上の不透明なロボットによる集合問題の解決が挙げられる．

参考文献

[1] Alberto Bandettini, Fabio Luiporini, Giovanni Viglietta.: A Survey on Open Problems for Mobile Robots : arXiv:1111.2259v1 [cs.RO] 7 Nov 2011

[2] Taisuke Izumi, Samia Souissi, Yoshiaki Katayama, Nobuhiro Inuzuka, Xavier Defago, Koichi Wada, and Masafumi Yamashita : The Gathering Problem for Two Oblivious Robots with Unreliable Compasses : SIAM J. Comput., 41(1), 26-46

[3] Paola Flocchini, Nicola Santoro, Giovanni Viglietta, Masafumi Yamashita : Rendezvous of Two Robots with Constant Memory : Structural Information and Communication Complexity Lecture Notes in Computer Science Volume 8179, 2013, pp 189-200

[4] Mark Cieliebak, Paola Flocchini, Giuseppe Prencipe, Nicola Santoro : Solving the Robots Gathering Problem : Automata, Languages and Programming, Lecture Notes in Computer Science Volume 2719, 2003, pp 1181-1196

[5] Ando H, Suzuki I, Yamashita M : Distributed Memoryless Point Convergence Algorithm for Mobile Robots with Limited Visibility : Robotics and Automation, IEEE Transactions on (Volume:15 , Issue: 5)

[6] Jurek Czyzowicz, Leszek Gasieniec, Andrzej Pelc : Gathering Few Fat Mobile Robots in the Plane, Theoretical Computer Science 410 (2009) 481-499

[7] Chrysovalandis Agathangelou, Chryssis Georgiou, Marios Mavronicolas : A Distributed Algorithm for Gathering Many Fat Mobile Robots in the Plane, arXiv:1209.3904v1 [cs.DC] 18 Sep 2012

[8] Sruti Gan Chaudhuri and Krishnendu Mukhopadhyaya : Leader Election and Gathering for Asynchronous Transparent Fat Robots without Chirality : arXiv:1208.4484v1 [cs.DC] 22 Aug 2012

[9] Suparno Datta, Ayan Dutta, Sruti Gan Chaudhuri, and Krishnendu Mukhopadhyaya : Circle Formation by Asynchronous Transparent Fat Robots : Distributed Computing and Internet Technology Lecture Notes in Computer Science Volume 7753, 2013, pp 195-207

[10] 伊藤公一, 片山喜章, 和田幸一 : 共通座標系を有するファットロボットのグリッド上での集合について, 第 143 回アルゴリズム研究会, 研究報告アルゴリズム (AL),2013-AL-143(2),1-8(2013-02-22)

リングおよび木における
モバイルエージェントの部分集合アルゴリズム

柴田 将拡 †, 大下 福仁 †, 角川 裕次 †, 増澤 利光 †

† 大阪大学 大学院情報科学研究科,565-0871, 大阪府吹田市山田丘 1-5
{m-sibata, s-kawai, f-oosita, kakugawa, masuzawa}@ist.osaka-u.ac.jp

Abstract

In this paper, we consider the partial gathering problem of mobile agents in asynchronous unidirectional ring networks and asynchronous tree networks. The partial gathering problem is a new generalization of the total gathering problem which requires that all the agents meet at the same node. The partial gathering problem requires, for given input g, that each agent should move to a node and terminate so that at least g agents should meet at the same node. The requirement for the partial gathering problem is weaker than that for the (well-investigated) total gathering problem, and thus, we have interests in clarifying the difference on the move complexity between them. We assume that n is the number of nodes and k is the number of agents. For ring networks, we propose three algorithms to solve the partial gathering problem. The rst algorithm is deterministic but requires unique ID of each agent. This algorithm achieves partial gathering in $O(gn)$ total moves. The second algorithm is randomized and requires no unique ID of each agent (i.e., anonymous). This algorithm achieves the partial gathering in expected $O(gn)$ total moves. The third algorithm is deterministic and works for anonymous agents. In this case, we show that there exist initial con gurations in which no algorithm can solve the problem for this setting, and agents can achieve the partial gathering in $O(kn)$ total moves for other initial con gurations. For tree networks, we consider three model variants to solve the partial gathering problem. First, we show that there exist no algorithms to solve the partial gathering problem in the weak multiplicity detection and non-token model. Next, we propose two algorithms to solve the partial gathering problem. First, we consider the strong multiplicity detection and non-token model. In this model, we show that agents require $\Omega(kn)$ total moves to solve the partial gathering problem and we propose an algorithm to achieve the partial gathering in $O(kn)$ total moves. Second, we consider the weak multiplicity detection and removable-token model. In this model, we propose an algorithm to achieve the partial gathering in $O(gn)$ total moves. It is known that the total gathering problem requires $\Omega(kn)$ total moves. Hence, our results show that it is possible that the g-partial gathering problem can be solved with fewer total moves compared to the total gathering problem.

keyword: distributed system, mobile agent, gathering problem, partial gathering

1 Introduction

1.1 Background and our contribution

A *distributed system* is a system that consists of a set of computers (*nodes*) and communication links. In recent years, distributed systems have become large and design of distributed systems has become complicated. As a way to design e cient distributed systems, (mobile) agents have attracted a lot of attention [1, 2, 3, 4, 5]. Agents simplify design of distributed systems because they can traverse the system and process tasks on each node.

The gathering problem is a fundamental problem for cooperation of agents [1, 6, 7, 8, 9]. The gathering problem requires all agents to meet at a single node in nite time. The gathering problem is useful because, by meeting at a single node, all agents can share information or synchronize behaviors among them.

In this paper, we consider a variant of the gathering problem, called the *partial gathering problem*. The partial gathering problem does not always require all agents to gather at a single node, but requires agents to gather partially at several nodes. More precisely, we consider the problem which requires, for given input g, that each agent should move to a node and terminate so that at least g agents should meet at the same node. We de ne this problem as the *g-partial gathering problem*. Clearly, if $k/2 < g$ k holds, the g-partial gathering problem is equal to the ordinary gathering problem. If 1 g $k/2$ holds, the requirement for the g-partial gathering problem is weaker than that for the ordinary gathering problem, and thus it seems possible to solve the g-partial gathering problem with fewer total moves. In addition, the g-partial gathering problem is still useful because agents can share information and process tasks cooperatively among at least g agents.

Table 1: Results for the g-partial gathering problem in asynchronous unidirectional rings

Model	Algorithm 1	Algorithm 2	Algorithm 3
Unique ID	Available	Not available	Not available
Deterministic/Randomized	Deterministic	Randomized	Deterministic
Knowledge of k	Not available	Available	Available
The total moves	$O(gn)$	$O(gn)$	$O(kn)$
Note			There exist unsolvable con gurations

Table 2: Results for the g-partial gathrering problem in asynchronous trees

	Model 1	Model 2	Model 3
Multiplicity detection	Weak	Strong	Weak
Removable-token	Not available	Not available	Available
Solvable / Unsolvable	Unsolvable	solvable	Unsolvable
The total moves	-	$O(kn)$	$O(gn)$

In this paper, we consider the g-partial gathering problem for asynchronous unidirectional ring networks and asynchronous tree networks. We assume that n is the number of nodes and k is the number of agents. The contributions of this paper are summarized in Tables 1 and 2. For asynchronous unidirectional ring networks, we propose three algorithms to solve the g-partial gathering problem. First, we propose a deterministic algorithm to solve the g-partial gathering problem for the case that agents have distinct IDs. This algorithm requires $O(gn)$ total moves. Second, we propose a randomized algorithm to solve the g-partial gathering problem for the case that agents have no IDs but agents know the number k of agents. This algorithm requires expected $O(gn)$ total moves. Third, we consider a deterministic algorithm to solve the g-partial gathering problem for the case that agents have no IDs but agents know the number k of agents. In this case, we show that there exist initial con gurations in which the g-partial gathering problem is unsolvable. Next, we propose a deterministic algorithm to solve the g-partial gathering problem for any solvable initial con gurations. This algorithm requires $O(kn)$ total moves. Note that the total gathering problem requires $\Omega(kn)$ total moves regardless of deterministic or randomized settings. Hence, the rst and second algorithms imply that the g-partial gathering problem can be solved in fewer total moves compared to the total gathering problem for the both cases. In addition, we show that the total moves is $\Omega(gn)$ for the g-partial gathering problem if g 2. This means the rst and second algorithms are asymptotically optimal in terms of the total moves.

For asynchronous tree networks, we consider two multiplicity detection models and two token models. First, we consider the case of the weak multiplicity detection and non-token model, where in the weak multiplicity detection model each agent can detect whether another agent exists at the current node or not but cannot count the exact number of agents. In this case, we show that there exist no algorithms to solve the g-partial gathering problem in this model. Next, we consider the case of the strong multiplicity detection and non-token model, where in the strong multiplicity detection model each agent can count the number of agents at the current node. In this case, we show that agents require $\Omega(kn)$ total moves to solve the g-partial gathering problem. In addition, we propose a deterministic algorithm to solve the g-partial gathering problem in $O(kn)$ total moves, that is, this algorithm is asymptotically optimal in terms of the total moves. Finally, we consider the case of the weak multiplicity detection and removable-token model. In this case, we propose a deterministic algorithm to solve the g-partial gathering problem in $O(gn)$ total moves. This result shows that the total moves can be reduced by using tokens. Since agents require $\Omega(gn)$ total moves to solve the g-partial gathering problem also in tree networks, this algorithm is also asymptotically optimal in terms of the total moves.

1.2 Related works

Many fundamental problems for cooperation of mobile agents have been studied in literature. For example, the searching problem [2, 10, 5], the gossip problem [3], the election problem [11], the map construction problem [4], and the total gathering problem [1, 6, 7, 8, 9] have been studied.

In particular, the total gathering problem has received a lot of attention and has been extensively studied in many topologies, which include lines [12, 13], trees [1, 3, 14, 7, 8, 9], tori [1, 15], arbitrary graphs [16, 17, 12] and rings [1, 18, 3, 6, 12]. The total gathering problem for rings and trees has been extensively studied because these networks are utilized in a lot of applications. To solve the total gathering problem, it is necessary to select exactly one gathering node, i.e., a node where all agents meet. There are many ways to select the gathering node. For example, in [1, 19, 20, 21, 15, 18], agents leave marks (tokens) on their initial nodes and select the

gathering node based on every distance of neighboring tokens. In [2, 10], agents have distinct IDs and select the gathering node based on the IDs. In [6], agents can use random numbers and select the gathering node based on IDs generated randomly. In [1, 3, 11], agents execute the leader agent election and the elected leader decides the gathering node. In [14, 7, 8, 9, 16], agents explore graphs and decide which node they meet at.

2 Preliminaries

2.1 Network and Agent Model

2.1.1 Unidirectional Ring Network

A *unidirectional ring network* R is a tuple $R = (V, L)$, where V is a set of nodes and L is a set of communication links. We denote by n $(= |V|)$ the number of nodes. Then, ring R is de ned as follows.

- $V = \{v_0, v_1, \ldots, v_{n\ \ 1}\}$
- $L = \{(v_i, v_{(i+1) \bmod n}) \mid 0 \quad i \quad n \quad 1\}$

We de ne the direction from v_i to v_{i+1} as a *forward* direction, and the direction from v_{i+1} to v_i as a *backward* direction. In addition, we de ne the i-th forward (resp.,) backward agent of the agent $a_{h'}$ as the agent that exist in the a_h's forward (resp., backward) direction and there are $i \quad 1$ agents between a_h and $a_{h'}$.

In this paper, we assume nodes are anonymous, i.e., each node has no ID. In a unidirectional ring, every node $v_i \in V$ has a whiteboard and agents on node v_i can read from and write to the whiteboard of v_i. We de ne W as a set of all states of a whiteboard.

Let $A = \{a_1, a_2, \ldots, a_k\}$ be a set of agents. We consider three model variants. In the rst model, we consider agents that are distinct (i.e., agents have distinct IDs) and execute a deterministic algorithm. We model an agent as a nite automaton $(S, \ , s_{initial}, s_{final})$. The rst element S is the set of the agent a_h's all states, which includes initial state $s_{initial}$ and nal state s_{final}. After an agent changes its state to s_{final}, the agent terminates the algorithm. The second element is the state transition function. Since we treat deterministic algorithms, is described as $: S \times W \to S \times W \times M$, where $M = \{1, 0\}$ represents whether the agent moves forward or not in the next movement. The value 1 represents movement to the next node and 0 represents stay at the current node. Since rings are unidirectional, each agent only moves to its forward node. We assume that agents move instantaneously, that is, agents always exist at nodes (do not exist at links). Moreover, we assume that each agent cannot detect whether other agents exist at the current node or not. Notice that S, , $s_{initial}$ and s_{final} can be dependent on the agent's ID.

In the second model, we consider agents that are anonymous (i.e., agents have no IDs) and execute a randomized algorithm. We model an agent similarly to the rst model except for state transition function . Since we treat randomized algorithms, is described as $: S \times W \times R \to S \times W \times M$, where R represents a set of random values. In addition, we assume that each agent knows the number of agents. Notice that all the agents are modeled by the same state machine.

In the third model, we consider agents that are anonymous and execute a deterministic algorithm. We also model an agent similarly to the rst model. We assume that each agent knows the number of agents. Note that all the agents are modeled by the same machine.

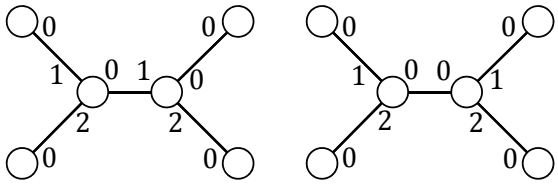
In unidirectional ring network model, we assume that agents move instantaneously, that is, agents always exist at nodes (do not exist at links). Moreover, we assume that each agent cannot detect whether other agents exist at the current node or not.

2.1.2 Tree Network

A *tree network* T is a tuple $T = (V, L)$, where V is a set of nodes and L is a set of communication links. We denote by n $(= |V|)$ the number of nodes. Let d_v be the degree of v. We assume that each link l incident to v_j is uniquely labeled from the set $\{0, 1, \ldots, d_{v_j} \quad 1\}$. We call this label *port number*. Since each communication link connects to two nodes, it has two port numbers. However, port numbering is *local*, that is, there is no coherence between two port numbers of each communication link. The path $P(v_0, v_k) = (v_0, v_1, \ldots, v_k)$ with length k is a sequence of nodes from v_0 to v_k such that $\{v_i, v_{i+1}\} \in L$ $(0 \quad i < k)$ and $v_i \neq v_j$ if $i \neq j$. Note that, for any $u, v \in V$, $P(u, v)$ is unique in a tree. The *distance* from u to v, denoted by $dist(u, v)$, is the length of the path from u to v. The *eccentricity* $r(u)$ of node u is the maximum distance from u to an arbitrary node, i.e., $r(u) = \max_{v \in V} dist(u, v)$. The *radius* R of the network is the minimum eccentricity in the network. A node with eccentricity R is called a *center*. We use the following theorem about a center later [22].

Theorem 2.1 *There exist one or two center nodes in a tree. If there exist two center nodes, they are neighbors.*

Next we de ne symmetry of trees, which is important to consider solvability in Section 4.1



(a) A non-symmetric tree (b) A symmetric tree

Figure 1: Non-symmetric and symmetric tree

De nition 2.1 *A tree T is symmetric iff there exists a function $g : V \to V$ such that all the following conditions hold (See Figure. 1):*

- *For any $v \in V, v \neq g(v)$ holds.*
- *For any $u, v \in V, u$ is adjacent to v iff $g(u)$ is adjacent to $g(v)$.*
- *For any $\{u, v\} \in E$ and $\{g(u), g(v)\}$,a port number labeled at u is equal to a port number labeled $g(u)$.*

When tree T is symmetric, we say nodes u and v in T are symmetric if $u = g(v)$ holds.

It is well known (cf. ex.[23]) that the following lemma holds because agents cannot distinguish u and v if u and v are symmetric.

lemma 2.1 *Assume that nodes u and v are symmetric in tree T. If agents a_1 and a_2 start an algorithm from u and v respectively, there exists an execution in which they act in a symmetric fashion.*

Let $A = \{a_1, a_2, \ldots, a_k\}$ be a set of agents. We assume that each agent does not know the number n of nodes and the number k of agents. We consider the *strong multiplicity detection model* and the *weak multiplicity detection model* in tree networks. In the strong multiplicity detection model, each agent can count the number of agents at the current node. In the weak multiplicity detection model, each agent can recognize whether another agent stays at the same node or not, but cannot count the number of agents on its current node. However, in both models, each agent cannot detect the states of agents at the current node. Moreover, we consider the *non-token model* and the *removable-token model*. In the non-token model, agents cannot mark the nodes or the edges in any way. In the removable-token model, each agent initially has a token and can leave it on a node, and agents can remove such tokens.

We assume that agents are anonymous (i.e., agents have no IDs) and execute a deterministic algorithm. We model an agent as a nite automaton $(S, , s_{initial}, s_{final})$. The rst element S is the set of all states of agents, which includes initial state $s_{initial}$ and nal state s_{final}. When an agent changes its state to s_{final}, the agent terminates the algorithm. The second element is the state transition function. In the weak multiplicity detection and non-token model, is described as $: S \times M_T \times EX_A \to S \times M_T$. In the de nition, set $M_T = \{\perp, 0, 1, \ldots, \Delta \quad 1\}$ represents the agent's movement, where Δ is the maximum degree of the tree. In the left side of , the value of M_T represents the port number of the current node that the agent observes in visiting the current node (The value is $\perp$ in the rst activation). In the right side of , the value of M_T represents the port number through which the agent leaves the current node to visit the next node. If the value is $\perp$, the agent does not move and stays at the current node. in addition, $EX_A = \{0, 1\}$ represents whether another agent stays at the current node or not. The value 0 represents that no other agents stay at the current node, and the value 1 represents that another agent stays at the current node.

In the strong multiplicity detection and non-token model, is described as $: S \times M_T \times N \to S \times M_T$. In the de nition, N represents the number of other agents at the current node. In the weak multiplicity detection and removable-token model, is described as $: S \times M_T \times EX_A \times EX_T \to S \times EX_T \times M_T$. In the de nition, in the left side of , $EX_T = \{0, 1\}$ represents whether a token exists at the current node or not. The value 0 of EX_T represents that there does not exist a token at the current node, and the value 1 of EX_T represents that there exists a token at the current node. In the right side of , $EX_T = \{0, 1\}$ represents whether an agent remove a token at the current node or not. If the value of EX_T in the left side is 1 and the value of EX_T in the right side is 0, it means that an agent removes a token at the current node. Otherwise, it means that an agent does not remove a token at the current node. Note that, in each model we assume that each agent is not imposed any restriction on the memory.

In the tree network model, we assume that agents do not move instantaneously, that is, agents may exist in links. Moreover, agents move through a link in a FIFO manner, that is, when an agent a_i leaves v_j after a_h

leaves v_j through the same communication link as a_h, then a_i reaches v_i's neighboring node $v_{i'}$ after a_h reaches $v_{i'}$. In addition, if a_h reaches v_j before a_i reaches v_j through the same link as a_h, a_h takes a step before a_i takes a step, where we explain the mean of a step later.

2.2 System configuration

If the network is a ring, (global) *con guration* c is de ned as a product of states of agents, states of nodes (whiteboards), and locations of agents. In initial con guration c_0, we assume that no pair of agents stay at the same node. We assume that each node v_j has boolean variable $v_j.initial$ that indicates existence of agents in the initial con guration. If there exists an agent on node v_j in the initial con guration, the value of $v_j.initial$ is true. Otherwise, the value of $v_j.initial$ is false.

If the network is a tree, in the non-token models con guration c is de ned as a product of states of agents and locations of agents. In the removable-token model, con guration c is de ned as a product of states of agents, states of nodes (tokens), and locations of agents. Moreover, in the initial con guration c_0, we assume that the node v_j has a token if there exists an agent at v_j, and v_j does not have a token if there exists no agents at v_j. In both network models, we assume that no pair of agents stay at the same node in the initial con guration c_0.

Let A_i be an arbitrary non-empty set of agents. When con guration c_i changes to c_{i+1} by a step of every agent in A_i, we denote the transition by $c_i \overset{A_i}{\longrightarrow} c_{i+1}$. If the network is a ring, in c_i, each $a_j \in A_i$ reads values written on its node's whiteboard, executes local computation, writes values to the node's whiteboard, and moves to the next node or stays at the current node. If the network is a tree, each $a_j \in A_i$ reaches some node (if a_j exists in some link), executes local computation, leaves the node or stays at the node as one common atomic step in each model. Concretely, in the weak multiplicity detection and non-token model, each $a_j \in A_i$ reaches some node (if a_j exists in some link), detects whether there exists another agent at the current node or not, executes local computation, decides the port number, and moves to the node through the port number or stays at the current node. In the strong multiplicity detection and non-token model, each $a_j \in A_i$ reaches some node (if a_j exists in some link), counts the number of agents at the current node, executes local computation, decides the port number, and moves to the node through the port number or stays at the current node. In the weak multiplicity detection and the removable-token model, each $a_j \in A_i$ reaches some node (if a_j exists in some link), detects whether there exists another agent at the current node or not, detects whether there exists a token at the current node or not, executes local computation, decides whether the a_j removes the token or not (if any), decides the port number, and moves to the node through the port number or stays at the current node. When a_j completes this series of events, we say that a_j takes one step. If the network is a ring and multiple agents at the same node are included in A_i, the agents take steps in an arbitrary order. When $A_i = A$ holds for any i, all agents take steps. This model is called the *synchronous model*. Otherwise, the model is called the *asynchronous model*.

If sequence of con gurations $E = c_0, c_1, \ldots$ satis es $c_i \overset{A_i}{\longrightarrow} c_{i+1}$ ($i \quad 0$), E is called an *execution* starting from c_0. Execution E is in nite, or ends in nal con guration c_{final} where every agent's state is s_{final}.

2.3 Partial gathering problem

The requirement of the partial gathering problem is that, for a given input g, each agent should move to a node and terminate so that at least g agents should meet at the node. Formally, we de ne the g-partial gathering problem as follows.

De nition 2.2 *Execution E solves the g-partial gathering problem when the following conditions hold:*

- *Execution E is nite.*
- *In the nal con guration, for any node v_j such that there exist some agents on v_j, there exist at least g agents on v_j.*

In addition, we have the following lower bound in the ring networks.

Theorem 2.2 *The total moves required to solve the g-partial gathering problem in the ring networks is $\Omega(gn)$ if $g \quad 2$.*

Proof. We consider an initial con guration such that all agents are scattered evenly. We assume $n = ck$ holds for some positive integer c. Let V' be the set of nodes where agents exist in the nal con guration, and let $x = |V'|$. Since at least g agents meet at v_j for any $v_j \in V'$, we have $k \quad gx$.

For each $v_j \in V'$, we de ne A_j as the set of agents that meet at v_j and T_j as the total moves of agents in A_j. Then, among agents in A_j, the i-th smallest number of moves to get to v_j is at least $(i \quad 1)n/k$. So, we have

$$\begin{aligned} T_j & \quad \sum_{i=1}^{g}(i \quad 1) \cdot \frac{n}{k} + (|A_j| \quad g) \cdot \frac{gn}{k} \\ & = \quad \frac{n}{k} \cdot \frac{g(g \quad 1)}{2} + (|A_j| \quad g) \cdot \frac{gn}{k} \end{aligned}$$

Therefore, the total moves is at least

$$\begin{aligned} T & = \sum_{v_j \in V'} T_j \\ & \quad x \cdot \frac{n}{k} \cdot \frac{g(g \quad 1)}{2} + (k \quad gx) \cdot \frac{gn}{k} \\ & = gn \quad \frac{gnx}{2k}(g+1). \end{aligned}$$

Since $k \quad gx$ holds, we have

$$T \quad \frac{n}{2}(g \quad 1).$$

Thus, the total moves is at least $\Omega(gn)$.

Note that, we can also show the theorem for the case the network is tree by assuming that the network is line.

3 Partial Gathering in Ring Networks

We propose three algorithms to solve g-partial gathering problem. The rst algorithm is deterministic and assumes unique ID of each agent. The second algorithm is randomized and assumes anonymous agents. The last algorithm is deterministic and assumes anonymous agents.

3.1 A Deterministic Algorithm for Distinct Agents

In this section, we propose a deterministic algorithm to solve the g-partial gathering problem for distinct agents (i.e., agents have distinct IDs). The basic idea to solve the g-partial gathering problem is that agents select a leader and then the leader instructs other agents which node they meet at. However, since $\Omega(n \log k)$ total moves is required to elect one leader [3], this approach cannot lead to the g-partial gathering in asymptotically optimal total moves (i.e., $O(gn)$). To overcome this lower bound, we select multiple agents as leaders by executing leader agent election partially. By this behavior, our algorithm solves the g-partial gathering problem in $O(gn)$ total moves.

The algorithm consists of two parts. In the rst part, agents execute leader agent election partially and elect some leader agents. In the second part, the leader agents instruct the other agents which node they meet at, and the other agents move to the node by the instruction.

3.1.1 The rst part: leader election

The aim of the rst part is to elect leaders that satisfy the following properties: 1) At least one agent is elected as a leader, 2) at most $\lfloor k/g \rfloor$ agents are elected as leaders, and 3) there exist at least $g \quad 1$ non-leader agents between two leader agents. To attain this goal, we use a traditional leader election algorithm [24]. However, the algorithm in [24] is executed by nodes and the goal is to elect exactly one leader. So we modify the algorithm to be executed by agents, and then agents elect at most $\lfloor k/g \rfloor$ leader agents by executing the algorithm partially.

During the execution of leader election, the states of agents are divided into the following three types:

- *active*: The agent is performing the leader agent election as a candidate of leaders.
- *inactive*: The agent has dropped out from the candidate of leaders.
- *leader*: The agent has been elected as a leader.

First, we explain the idea of leader election by assuming that the ring is synchronous and bidirectional. The algorithm consists of several phases. In each phase, each active agent compares its own ID with IDs of its forward and backward neighboring active agents. More concretely, each active agent writes its ID on the whiteboard

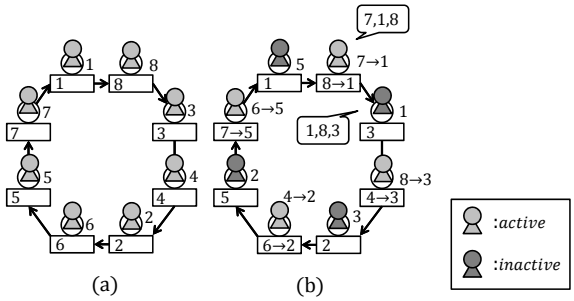


Figure 2: An example for a g-partial gathering problem($k = 9, g = 3$)

of its current node, and then moves forward and backward to observe IDs of the forward and backward active agents. If its own ID is the smallest among the three agents, the agent remains active (as a candidate of leaders) in the next phase. Otherwise, the agent drops out from the candidate of leaders and becomes inactive. Note that, in each phase, neighboring active agents never remain as candidates of leaders. So, at least half active agents become inactive and the number of inactive agents between two active agents at least doubles in each phase. And from [24], executing j phases, there exists at least 2^j 1 inactive agents between two active agents. Thus, after executing $\lceil \log g \rceil$ phases, the following properties are satis ed: 1) At least one agent remains as a candidate of leaders, 2) at most $\lfloor k/g \rfloor$ agents remain as candidates of leaders, and 3) the number of inactive agents between two active agents is at least g 1. Therefore, all remaining active agents become leaders. Note that, during the execution of the algorithm, the number of active agents may become one. In this case, the active agent immediately becomes a leader.

In the following, we implement the above algorithm in asynchronous unidirectional rings. First, we apply a traditional approach [24] to implement the above algorithm in a unidirectional ring. Let us consider the behavior of active agent a_h. In unidirectional rings, a_h cannot move backward and so cannot observe the ID of its backward active agent. Instead, a_h moves forward until it observes IDs of two active agents. Then, a_h observes IDs of three successive active agents. We assume a_h observes id_1, id_2, id_3 in this order. Note that id_1 is the ID of a_h. Here this situation is similar to that the active agent with ID id_2 observes id_1 as its backward active agent and id_3 as its forward active agent in bidirectional rings. For this reason, a_h behaves as if it would be an active agent with ID id_2 in bidirectional rings. That is, if id_2 is the smallest among the three IDs, a_h remains active as a candidate of leaders. Otherwise, a_h drops out from the candidate of leaders and becomes inactive. After the phase, a_h assigns id_2 to its ID if it remains active as a candidate. For example, consider the initial con guration in Fig. 2 (a). In the gures, the number near each agent is the ID of the agent and the box of each node represents the whiteboard. First, each agent writes its own ID to the whiteboard on its initial node. Next, each agent moves forward until it observes two IDs, and then the con guration is changed to the one in Fig. 2 (b). In this con guration, each agent compares three IDs. The agent with ID 1 observes IDs (1, 8, 3), and so it drops out from the candidate because the middle ID 8 is not the smallest. The agents with IDs 3, 2, and 5 also drop out from the candidates. The agent with ID 7 observes IDs (7, 1, 8), and so it remains active as a candidate because the middle ID 1 is the smallest. Then, it updates its ID to 1. The agents with IDs 8, 4, and 6 also remain active as candidates and similarly update their IDs.

Next, we explain the way to treat asynchronous agents. To recognize the current phase, each agent manages *a phase number*. Initially, the phase number is zero, and it is incremented when each phase is completed. Each agent compares IDs with agents that have the same phase number. To realize this, when each agent writes its ID to the whiteboard, it also writes its phase number. That is, at the beginning of each phase, active agent a_h writes a tuple $(phase, id_h)$ to the whiteboard on its current node, where $phase$ is the current phase number and id_h is the ID of a_h. After that, agent a_h moves until it observes two IDs with the same phase number as that of a_h. Note that, some agent a_h may pass another agent a_i. In this case, a_h waits until a_i catches up with a_h. We explain the ditails later. Then, a_h decides whether it remains active as a candidate or becomes inactive. If a_h remains active, it updates its own ID. Agents repeat these behaviors until they complete the $\lceil \log g \rceil$-th phase.

Pseudocode. The pseudocode to elect leader agents is given in Algorithm 1. All agents start the algorithm with active states. The pseudocode describes the behavior of active agent a_h, and v_j represents the node where agent a_h currently stays. If agent a_h changes its state to an inactive state or a leader state, a_h immediately moves to the next part and executes the algorithm for an inactive state or a leader state in section 3.1.2. Agent a_h uses variables $a_h.id_1$, $a_h.id_2$, and $a_h.id_3$ to store IDs of three successive active agents. Note that a_h stores its ID on $a_h.id_1$ and initially assigns its initial ID $a_h.id$ to $a_h.id_1$. Variable $a_h.phase$ stores the phase number of a_h. Each node v_j has variable $(v_j.phase, v_j.id)$, where an active agent writes its phase number and its ID. For any v_j, variable $(v_j.phase, v_j.id)$ is $(0, 0)$ initially. In addition, each node v_j has boolean variable $v_j.inactive$.

Algorithm 1 The behavior of active agent a_h (v_j is the current node of a_h.)

Variables in Agent a_h
int $a_h.phase$;
int $a_h.id_1$,$a_h.id_2$,$a_h.id_3$;
Variables in Node v_j
int $v_j.phase$;
int $v_j.id$;
boolean $v_j.inactive = false$;
Main Routine of Agent a_h
1: $a_h.phase = 1$ and $a_h.id_1 = a_h.id$
2: $(v_j.phase, v_j.id) = (a_h.phase, a_h.id_1)$
3: $BasicAction()$
4: $a_h.id_2 = v_j.id$
5: $BasicAction()$
6: $a_h.id_3 = v_j.id$
7: **if** $a_h.id_2 \quad min(a_h.id_1, a_h.id_3)$ **then**
8: $v_j.inactive = true$ and become inactive
9: **else**
10: **if** $a_h.phase = \lceil \log g \rceil$ **then**
11: change its state to a leader state
12: **else**
13: $a_h.phase = a_h.phase + 1$
14: $a_h.id_1 = a_h.id_2$
15: **end if**
16: return to step 2
17: **end if**

This variable represents whether there exists an inactive agent on v_j or not. That is, agents update the variable to keep the following invariant: If there exists an inactive agent on v_j, $v_j.inactive = ture$ holds, and otherwise $v_j.inactive = false$ holds. Initially $v_j.inactive = false$ holds for any v_j. In Algorithm 1, a_h uses procedure $BasicAction()$, by which agent a_h moves to node $v_{j'}$ satisfying $v_{j'}.phase = a_h.phase$. During the movement, a_h may pass some agent a_i. In this case, *BasicAction()* guarantees that a_h waits until a_i catches up with a_h.

We give the pseudocode of $BasicAction()$ in Algorithm 2. In $BasicAction()$, the main behavior of a_h is to move to node $v_{j'}$ satisfying $v_{j'}.phase = a_h.phase$. To realize this, a_h skips nodes where no agent initially exists (i.e., $v_j.initial = false$) or an inactive agent whose phase number is not equal to a_h's phase number currently exists (i.e., $v_j.inactive = true$ and $a_h.phase \neq v_j.phase$), and continues to move until it reaches a node where some active agent starts the same phase (lines 2 to 4). During the execution of the algorithm, it is possible that a_h becomes the only one candidate of leaders. In this case, a_h immediately becomes a leader (lines 9 to 11).

Since agents move asynchronously, agent a_h may pass some active agents. To wait for such agents, agent a_h makes some additional behavior (lines 5 to 8). First, like the transition from the con guration of Fig. 3(a) to that of Fig. 3(b), consider the case that a_h passes a_b with a smaller phase number. Let $x = a_h.phase$ and $y = a_b.phase$ $(y < x)$. In this case, a_h detects the passing when it reaches a node v_c such that $a_h.phase > v_c.phase$. Hence, a_h can wait for a_b at v_c. Since a_b increments $v_c.phase$ or becomes inactive at v_c, a_h waits at v_c until either $v_c.phase = x$ or $v_c.inactive = true$ holds (line 6). After a_b updates the value of either $v_c.phase$ or $v_c.inactive$, a_h resumes its behavior.

Next, consider the case that a_h passes a_b with the same phase number. In the following, we show that agents can treat this case without any additional procedure. Note that, because a_h increments its phase number after it collects two other IDs, this case happens only when a_b is a forward active agent of a_h. Let $x = a_h.phase = a_b.phase$. Let a_h, a_b, a_c, and a_d are successive agents that start phase x. Let v_h, v_b, v_c, and v_d are nodes where a_h, a_b, a_c, and a_d start phase x, respectively. Note that a_h (resp., a_b) decides whether it becomes inactive or not at v_c (resp., v_d). We consider further two cases depending on the decision of a_h at v_c. First, like the transition from the con guration of Fig. 4(a) to that of Fig. 4(b), consider the case a_h becomes inactive at v_c. In this case, since a_h does not update $v_c.id$, a_b gets $a_c.id$ at v_c and moves to v_d and then decides its behavior at v_d. Next, like the transition from the con guration of Fig. 5(a) to that of Fig. 5(b), consider the case a_h remains active at v_c. In this case, a_h increments its phase (i.e., $a_h.phase = x + 1$) and updates $v_c.phase$ and $v_c.id$. Note that, since a_h remains active, $a_h.id_2 = a_b.id$ is the smallest among the three IDs. Hence, $v_c.id$ is updated to $a_b.id$ by a_h. Then, a_h continues to move until it reaches v_d. If a_h reaches v_d before a_b reaches v_d, both $v_d.phase < a_h.phase$ and $v_d.inactive = false$ hold at v_d. Hence, a_h waits until a_b reaches v_d. On the other hand, when a_b reaches v_c, it sees $v_c.id = a_b.id$ because a_h has updated $v_c.id$. Since $a_b.id_1 = a_b.id_2$ holds,

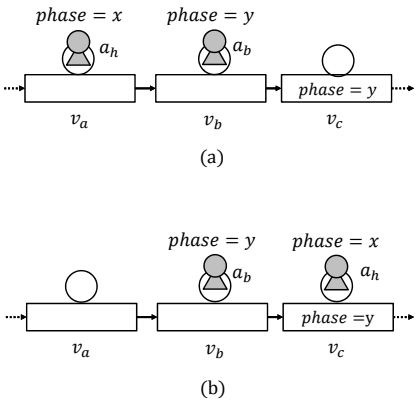


Figure 3: The rst example of an agent that passes other agents

Algorithm 2 Procedure *BasicAction*() for a_h

```
1: move to the forward node
2: while (v_j.initial = false) ∨ (v_j.inactive = true ∧ a_h.phase ≠ v_j.phase) do
3:    move to the forward node
4: end while
5: if a_h.phase > v_j.phase then
6:    wait until v_j.phase = a_h.phase or v_j.inactive = true
7:    return to step 2
8: end if
9: if (v_j.phase, v_j.id) = (a_h.phase, a_h.id_1) then
10:   change its state to a leader state
11: end if
```

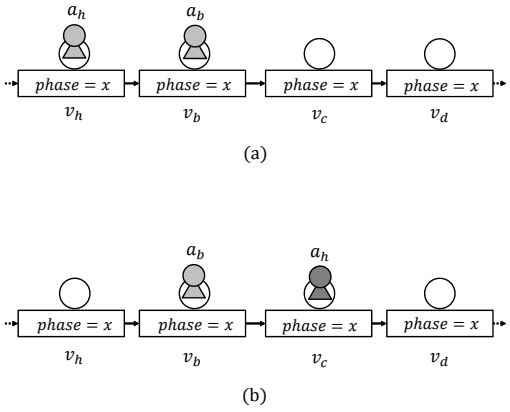


Figure 4: The second example of an agent that passes other agents

a_b becomes inactive when it reaches v_d. After that, a_h resumes the movement.

We have the following lemma about Algorithm 1 similarly to [24].

lemma 3.1 *Algorithm 1 eventually terminates, and the con guration satis es the following properties.*

- *There exists at least one leader agent.*
- *There exist at most* $\lfloor k/g \rfloor$ *leader agents.*
- *There exist at least* $g \quad 1$ *inactive agents between two leader agents.*

Proof. At rst, we show that Algorithm 1 eventually terminates. After executing $\lceil \log g \rceil$ phases, agents that have dropped out from the candidates of leaders are inactive states, and agents that remain active changes their states to leader states. Moreover, by the time executing $\lceil \log g \rceil$ phases, if there exists exactly one active agent

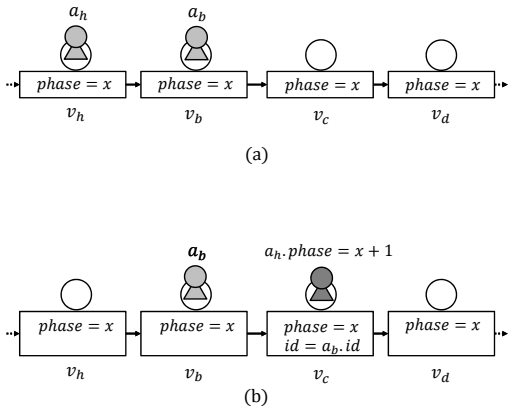


Figure 5: The third example of an agent that passes other agents

and the other agents are inactive, the active agent changes its state to a leader state. Therefore, Algorithm 1 eventually terminates. In the following, we show the above three properties.

First, we show that there exists at least one leader agent. From Algorithm 1, in each phase, if $a_h.id_2$ is strictly smaller than other two IDs, $a_h.id_1$ and $a_h.id_2$, a_h remains active. Otherwise, a_h becomes inactive. Since each agent uses unique ID, all active agents in some phase never become inactive. Hence, if there exist at least two active agents in some phase i, at least one agent remains acive after executing the phase i. Moreover, from lines 9 to 11 of Algorithm 2, if there exists exactly one candidate of leaders and the other agents remain inactive, the candidate becomes a leader. Therefore, there exists at least one leader agent.

Second, we show that there exist at most $\lfloor k/g \rfloor$ leader agents. In each phase, if an agent a_h remains as a candidate of leaders, then its forward and backward active agents drop out from candidates of leaders. Hence, in each phase, at least half active agents become inactive. Thus, after executing i phases, there exist at most $k/2^i$ active agents. Therefore, after executing $\lceil \log g \rceil$ phases, there exist at most $\lfloor k/g \rfloor$ leader agents.

Finally, we show that there exist at least g 1 inactive agents between two leader agents. At rst, we show that after executing j phases, there exist at least 2^j 1 inactive agents between two active agents. We show it by induction. For the case $j = 1$, there exists at least 2^1 $1 = 1$ inactive agents between two active agents as mentioned before. For the case $j = k$, we assume that there exist at least 2^k 1 inactive agents between two active agents. After executing $k + 1$ phases, since at least one of neighboring active agents becomes inactive, the number of inactive agents between two active agents is at least $(2^k \quad 1) + 1 + (2^k \quad 1) = 2^{k+1} \quad 1$. Hence, we can show that after executing j phases, there exist at least $2^j \quad 1$ inactive agents between two active agents. Therefore, after executing $\lceil \log g \rceil$ phases, there exist at least $g \quad 1$ inactive agents between two leader agents.

In addition, we have the following lemma similarly to [24].

lemma 3.2 *The total moves to execute Algorithm 1 is* $O(n \log g)$.

Proof. In each phase, each active agent moves until it observes two IDs of active agents. This total moves are $O(n)$ because each communication link is passed by two agents. Since agents execute this phase $\lceil \log g \rceil$ times, we have the lemma.

3.1.2 The second part: leaders' instruction and non-leaders' movement

In this section, we explain the second part, i.e., an algorithm to achieve the g-partial gathering by using leaders elected in the rst part. Let leader nodes (resp., inactive nodes) be the nodes where agents become leaders (resp., inactive agents) in the rst part. The idea of the algorithm is as follows: First each leader agent a_h writes 0 to the whiteboard on the current node. Then, a_h repeatedly moves and, whenever a_h visits an inactive node, a_h writes 0 if the number that a_h has visited inactive nodes plus one is not a multiple of g and a_h writes 1 otherwise. These numbers are used to instruct inactive agents where they should move to achieve the g-partial gathering. Note that, the number 0 means that agents do not meet at the node and the number 1 means that at least g agents meet at the node. Agent a_h continues this operation until it visits the node where 0 is already written to the whiteboard. Note that this node is a leader node. For example, consider the con guration in Fig. 6 (a). In this con guration, agents a_1 and a_2 are leader agents. First, a_1 and a_2 write 0 to their current whiteboards like Fig. 6 (b), and then they move and write numbers to whiteboards until they visit the node where 0 is written on the whiteboard. Then, the system reaches the con guration in Fig. 6 (c).

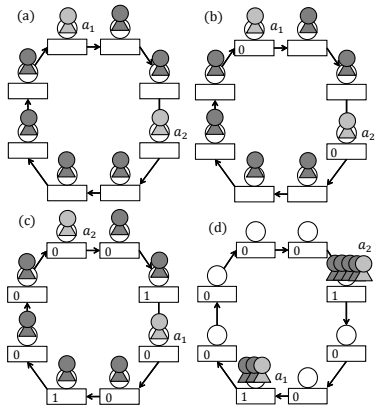


Figure 6: The realization of partial gathering($g = 3$)

Algorithm 3 Initial values needed in the second part (v_j is the current node of agent a_h.)

Variable in Agent a_h
int $a_h.count = 0$;
Variable in Node v_j
int $v_j.isGather = \perp$;

Then, each non-leader agent (i.e., inactive agent) moves based on the leader's instruction, i.e., the number written to the whiteboard. More concretely, each inactive agent moves to the rst node where 1 is written to the whiteboard. For example, after the con guration in Fig. 6 (c), each non-leader agent moves to the node where 1 is written to the whiteboard and the system reaches the con guration in Fig. 6 (d). After all, agents can solve the g-partial gathering problem.

Pseudocode. In the following, we show the pseudocode of the algorithm. In this part, states of agents are divided into the following three state

- *leader*: The agent instructs inactive agents where they should move.
- *inactive*: The agent waits for the leader's instruction.
- *moving*: The agent moves to its gathering node.

In this part, agents continue to use $v_j.initial$ and $v_j.inactive$. Remind that $v_j.initial = true$ if and only if there exists an agent at v_j initially. Algorithm 1 assures $v_j.inactive = true$ if and only if there exists an inactive agent at v_j. Note that, since each agent becomes inactive or a leader at a node such that there exists an agent initially, agents can ignore and skip every node $v_{j'}$ such that $v_{j'}.initial = false$.

At rst, the variables needed to achieve the g-partial gathering are described in Algorithm 3. Variables $a_h.count$ and $v_j.isGather$ are used so that leader agents instruct inactive agents which nodes they meet at. We explain these variables later. The initial value of $a_h.count$ is 0 and the initial value of $v_j.isGather$ is $\perp$.

The pseudocode of leader agents is described in Algorithm 4. Variable $a_h.count$ is used to count the number of inactive nodes a_h visits (The counting is done modulo g). Variable $v_j.isGahter$ is used for leader agents to instruct inactive agents. That is, when a leader agent a_h visits an inactive node v_j, a_h writes 1 to $v_j.isGather$ if $a_h.count = 0$, and a_h writes 0 to $v_j.isGather$ otherwise. Note that the number 1 means that at least g agents meet at the node and the number 0 means that agents do not meet at the node eventually. In asynchronous rings, leader agent a_h may pass agents that still execute Algorithm 1. To avoid this, a_h waits until the agents catch up with a_h. More precisely, when leader agent a_h visits the node v_j such that $v_j.initial = true$, it detects that it passes such agents if $v_j.inactive = false$ and $v_j.isGather = \perp$ hold. This is because $v_j.inactive = true$ should hold if some agent becomes inactive at v_j, and $v_j.isGather \neq \perp$ holds if some agent becomes leader at v_j. In this case, a_h waits there until either $v_j.inactive = true$ or $v_j.isGather \neq \perp$ holds (lines 7 to 9). When the leader agent updates $v_j.isGather$, an inactive agent on node v_j changes to a moving state (line 16). After a leader agent reaches the next leader node, it changes to a moving agent to move to the node where at least g agents meet (line 21). The behavior of inactive agents is given in the pseudocode of inactive agents (See Algorithm 5).

Algorithm 4 The behavior of leader agent a_h (v_j is the current node of a_h.)

1: $v_j.isGather = 0$ and $a_h.count = a_h.count + 1$
2: move to the forward node
3: **while** $v_j.isGather = \perp$ **do**
4: **while** $v_j.initial = false$ **do**
5: move to the forward node
6: **end while**
7: **if** $(v_j.inactive = false) \wedge (v_j.isGather = \perp)$ **then**
8: wait until $v_j.inactive = true$ or $v_j.isGather \neq \perp$
9: **end if**
10: **if** $v_j.inactive = true$ **then**
11: **if** $a_h.count = 0$ **then**
12: $v_j.isGather = 1$
13: **else**
14: $v_j.isGather = 0$
15: **end if**
16: // an inactive agent at v_j changes to a moving state
17: $a_h.count = (a_h.count + 1) \bmod g$
18: move to the forward node
19: **end if**
20: **end while**
21: change to a moving state

Algorithm 5 The behavior of inactive agent a_h (v_j is the current node of a_h.)

1: wait until $v_j.isGather \neq \perp$
2: change to a moving state

Algorithm 6 The behavior of moving agent a_h (v_j is the current node of a_h.)

1: **while** $v_j.isGather \neq 1$ **do**
2: move to the forward node
3: **if** $(v_j.initial = true) \wedge (v_j.isGather = \perp)$ **then**
4: wait until $v_j.isGather \neq \perp$
5: **end if**
6: **end while**

The pseudocode of moving agents is described in Algorithm 6. Moving agent a_h continues to move until it visits node v_j such that $v_j.isGather = 1$. After all agents visit such nodes, agents can solve the g-partial gathering problem. In asynchronous rings, a moving agent may pass leader agents. To avoid this, the moving agent waits until the leader agent catches up with it. More precisely, if moving agent a_h visits node v_j such that $v_j.initial = true$ and $v_j.isGather = \perp$, a_h detects that it passed a leader agent. To wait for the leader agent, a_h waits there until the value of $v_j.isGather$ is updated.

We have the following lemma about the algorithms in section 3.1.2.

lemma 3.3 *After the leader agent election, agents solve the g-partial gathering problem in $O(gn)$ total moves.*

Proof. At rst, we show the correctness of the proposed algorithm. From Algorithm 6, each moving agent moves to the nearest node v_j such that $v_j.isGather = 1$. By lemma 3.1, There exist at least g 1 moving agents between v_j and $v_{j'}$ such that $v_j.isGather = 1$ and $v_{j'}.isGather = 1$. Hence, agents can solve the g-partial gathering problem. In the following, we consider the total moves required to execute the algorithm.

First let us consider the total moves required for each leader agent to move to its next leader node. This total number of leaders' moves is obviously n. Next, let us consider the total moves required for each inactive (or moving) agent to move to node v_j such that $v_j.isGather = 1$ (For example, the total moves from Fig 6 (c) to Fig 6 (d)). Remind that there are at least g 1 inactive agents between two leader agents and each leader agent a_h writes g 1 times 0 consecutively and one time 1 to the whiteboard respectively. Hence, there are at most $2g$ 1 moving agents between v_j and $v_{j'}$ such that $v_j.isGather = 1$ and $v_{j'}.isGather = 1$. Thus, the number of this total moves is at most $O(gn)$ because each link is passed by agents at most $2g$ times. Therefore, we have the lemma.

From Lemmas 3.2 and 3.3, we have the following theorem.

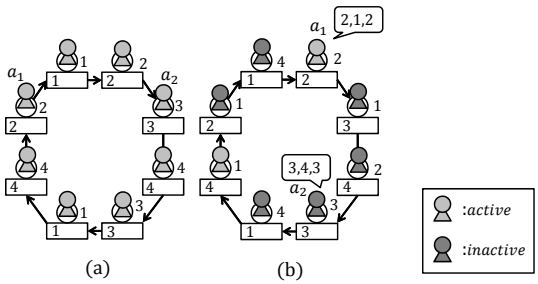


Figure 7: An example that some agent observes the same random IDs

Theorem 3.1 *When agents have distinct IDs, our deterministic algorithm solves the g-partial gathering problem in $O(gn)$ total moves.*

3.2 A Randomized Algorithm for Anonymous Agents

In this section, we propose a randomized algorithm to solve the g-partial gathering problem for the case of anonymous agents under the assumption that each agent knows the total number k of agent. The idea of the algorithm is the same as that in Section 3.1. In the rst part, agents execute the leader election partially and elect multiple leader agents. In the second part, the leader agents instruct the other agents where they move. In the previous section each agent has distinct ID, but in this section each agent is anonymous. In this section, agents solve the g-partial gathering problem by using random IDs instead of distinct IDs. We also show that agents solve the g-partial gathering problem in $O(gn)$ expected total moves.

3.2.1 The rst part: leader election

In this subsection, we explain a randomized algorithm to elect multiple leaders by using random IDs. The state of each agent is either active, inactive, leader, or semi-leader. Active, inactive, and leader agents behave similarly to Section 3.1.1, and we explain a semi-leader state later.

In the beginning of each phase, each active agent selects a random bits of $O(\log k)$ length as its own ID in the phase. After this, each agent executes the same way as Section 3.1.1, that is, each active agent moves until it observes two random IDs of active agents and compare three random IDs. If there exist no agents that observe the same random IDs, then, agents can execute the leader agent election similarly to Section 3.1.1. In this case, the total moves to execute the leader agent election are $O(n \log g)$. In the following, we explain the treatment for the case neighboring active agents have the same random IDs. Note that in this section, we assume that an agent becomes a leader at the node v_j, the agent set a *leader-flag* at v_j. We explain the treatment about a leader-flag later.

Let $a_h.id_1, a_h.id_2$, and $a_h.id_3$ be random IDs that an active agent a_h observes in some phase. If $a_h.id_1 = a_h.id_3 \neq a_h.id_2$ holds, then a_h behaves similarly to Section 3.1.1, that is, if $a_h.id_2 < a_h.id_1 = a_h.id_3$ holds, then a_h remains active and a_h becomes inactive otherwise. For example, let us con guration like Fig. 7 (a). Each active agent moves until it observes two random IDs like Fig. 7 (b). Then, agent a_1 observes three random IDs (2,1,2) and remains active because $a_1.id_2 < a_1.id_1 = a_1.id_3$ satis es. On the other hand, agent a_2 observes three random IDs (3,4,3) and becomes inactive because $a_2.id_2 > a_2.id_1 = a_2.id_3$ holds. The other agents do not observe the same random IDs and behave similarly to Section 3.1.1, that is, if their middle IDs are the smallest, they remain active and execute the next phase. If their middle IDs are not the smallest, they become inactive.

Next, we consider the case that $a_h.id_1 < a_h.id_2 = a_h.id_3$ or $a_h.id_1 = a_h.id_2 = a_h, id_3$ hold. In this case, a_h changes its own state to a *semi-leader* state. A semi-leader is an agent that has the possibility to become a leader if there exist no leader agents in the ring. The idea of behavior of each semi-leader agent is as follows: First each semi-leader moves around the ring, setting a flag at each node where there exists an agent in the initial con guration. After moving around the ring, if there exist some leader agents in the ring, each semi-leader becomes inactive. Otherwise, multiple leaders are elected among semi-leaders and the other agents become inactive. More concretely, when an active agent becomes a semi-leader, the semi-leader a_h sets a *semi-leader-flag* on its current whiteboard. This flag is used to share the same information among semi-leaders. In the following, we de ne *a semi-leader node* (resp., *a non-semi-leader node*) as the node that is set (resp., not set) a semi-leader-flag. After setting a semi-leader-flag, a_h moves around in the ring. While moving, when a_h visits a non-semi-Leder node v_j where there exists an agent in the initial con guration, that is, a non-semi-leader node v_j such that $v_j.initial = true$ holds, a_h sets the *tour-flag* on its current whiteboard. This flag is used so that each agent of any state can detect there exists a semi-leader in the ring. Moreover, when a_h visits a

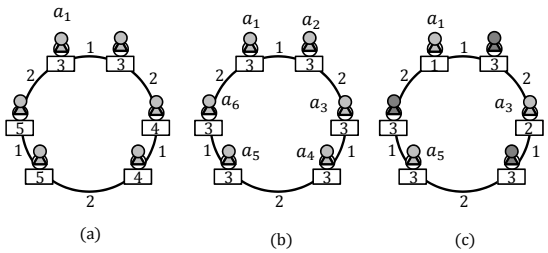


Figure 8: The behavior of semi-leaders

semi-leader node, a_h memorizes a pair of a random ID written to the current whiteboard and the number of tour-flag between two neighboring semi-leader nodes to an array $a_h.semi\text{-}leadersInfo$. This pair is used to decide if a semi-leader a_h becomes a leader or inactive after moving around the ring. We de ne $pair_i^h$ as a pair that a_h memorized for the i-th time.

After moving around the ring, a_h decides if it becomes a leader or inactive. While moving around the ring, if a_h observes a leader-flag, this means that there exist some leader agents in the ring, In this case, a_h becomes inactive. Otherwise, a_h decides if it becomes a leader or inactive by the value of $a_h.semi\text{-}leadersInfo$. Let $a_h.semi\text{-}leadersInfo = (pair_1^h, pair_2^h, \ldots, pair_t^h)$, where t implies the number of semi-leaders. Then, we de ne $info_{min}$ as the lexicographically minimum array among $\{a_h.semi\text{-}leadersInfo | a_h \text{ is a semi-leader }\}$. For array $info = (pair_1, pair, \ldots, pair_t)$, we de ne $shift(info, x) = (pair_x, pair_{1+x}, \ldots, pair_t, pair_1, \ldots, pair_{x\ 1})$. If $info = shift(info, x)$ holds for some x such that $0 < x < t$, we say $info$ is periodic. If $info$ is periodic, we de ne the period of info as $period = \min\{x > 0 | info = shift(info, x)\}$. If $a_h.semi\text{-}leadersInfo$ is not periodic, there exists exactly one semi-leader $a_{h'}$ that $a_{h'}.semi\text{-}leadersInfo = info_{min}$. Then, a_h becomes a leader and the other semi-leaders become inactive. For example, consider the con guration in Fig. 8(a). For simplicity, we omit nodes with no semi-leaders. Each number in the whiteboard represents a random ID, and each number near the link represents the numbers of tour-flags between two leader-flag. The semi-Leadedr a_1 moves around the ring and obtains $a_1.semi\text{-}leadersInfo = ((3,1),(3,2),(4,1),(4,2),(5,1),(5,2))$. Since $a_1.semi\text{-}leadersInfo = info_{min}$ holds, a_1 becomes a leader. On the other hand, each semi-leader a_i $(i \neq 1)$ becomes inactive because $a_i.semi\text{-}leadersInfo \neq info_{min}$ holds.

If $a_h.semi\text{-}leadersInfo$ is periodic, there exist several semi-leaders a_h that $a_h.semi\text{-}leadersInfo = info_{min}$ holds, and we de ne A_{semi} as the set of such agents. In this case, each semi-leader a_i that $semi\text{-}leadersInfo_i \neq info_{min}$ holds becomes inactive, and each semi-leader $a_h \in A_{semi}$ decides if a_h becomes a leader or not by the number of agents in A_{semi}. If $|A_{semi}| \quad \lfloor k/g \rfloor$ holds, a_h becomes a leader (the other agents become inactive). If $|A_{semi}| > \lfloor k/g \rfloor$ holds, then a_h selects a random ID again, writes the value to the current whiteboard, moves around the ring. Then, a_h obtains new value of $a_h.semi\text{-}leadersInfo$. Each semi-leader a_h continues such a behavior until thre exist at most $\lfloor k/g \rfloor$ semi-leader agents a_h that $a_h.semi\text{-}leadersInfo = info_{min}$ holds. For example, let us consider the condiguration like Fig. 8(b). In this gure, $k = 15$ holds. Agents a_1, a_3, and a_5 obtain $semi\text{-}leadersInfo = ((3,1),(3,2),(3,1),(3,2),(3,1),(3,2))$. On the other hand, a_2, a_4, and a_6 obtain $semi\text{-}leadersInfo = ((3,2),(3,1),(3,2),(3,1),(3,2),(3,1))$. In this case, a_2, a_4, and a_6 do not satisfy the condition and drop out from candidates. Then, $|A_{semi}| = 3$ holds and there exist four other agents between a_1, a_3, and a_5. If $g = 5$, then $|A_{semi}| \quad \lfloor k/g \rfloor = 3$ holds, and a_1, a_3, and a_5 become leaders. If $g = 6$, then a_1, a_3, and a_5 select a random ID again, write the value to the current whiteboard, and move around the ring respectively. After this, we assume that con guration is transmitted to Fig. 8(c) Then, a_1 becomes a leader since its random ID is the smallest, On the other hand, a_3 and a_5 become inactive.

Pseudocode. The pseudocode of active agents is described in Algorithm 7. An active agent a_h stores its phase number in variable $a_h.phase$. Agent a_h uses the procedure $random(l)$ to get its own random ID. This procedure returns the random bits of l length. Agent a_h uses variables $a_h.id_1, a_h.id_2$, and $a_h.id_3$ to store random IDs of three successive active agents. Note that a_h stores its own random ID on $a_h.id_1$. Each node v_j has variable $v_j.phase$ and $v_j.id$, where an active agent writes its phase number and its random ID. For any v_j, initial values of these valiable are 0. In addition, v_j has boolean variable $tour\text{-}flag$ and $leader\text{-}flag$. The inital values of these variable are $false$. Moreover, a_h use a valiable $a_h.tLeaderObserve$, which represents whether a_h observes a tour-flag or not. If a_h observes a tour-flag, it means that there exists a semi-leader in the ring. The initial value of $a_h.tLeaderObserve$ is $false$.

In each phase, each active agent decides its own random ID of $3 \log k$ bits length through $random(l)$, and a_h moves until it observes two random IDs by $BasicAction()$ in Algorithm 2, and If each active agent a_h neither observes a tour-flag nor observes random IDs that $a_h.id_1 < a_h.id_2 = a_h.id_3$ or $a_h.id_1 = a_h.id_2 = a_h.id_3$ hold, this pseudocode works similarly to Algorithm 3.1.1, and when an agent becomes a leader, the agent set a leader-flag at v_j. If an active agent a_h observes a tour-flag, then a_h moves until it observes two random IDs of active agents and becomes inactive. If an active agent a_h observes three random IDs that $a_h.id_1 > a_h.id_2 = a_h.id_3$ or

Algorithm 7 The behavior of active agent a_h (v_j is the current node of a_h)

Variables in Agent a_h
int $a_h.phase$;
int $a_h.id_1$,$a_h.id_2$,$a_h.id_3$;
boolean $a_h.semiObserve = false$
Variables in Node v_j
int $v_j.phase$;
int $v_j.id$;
boolean $v_j.inactive = false$;
boolean $tour\text{-}flag = false$;
boolean $leader\text{-}flag = false$;
Main Routine of Agent a_h
1: $a_h.phase = 1$
2: $a_h.id_1 = random(3 \log k)$
3: $v_j.phase = a_h.phase$
4: $v_j.id = a_h.id_1$
5: $BasicAction()$
6: **if** $v_j.tour = true$ **then**
7: $a_h.semiObserve = true$
8: **end if**
9: $a_h.id_2 = v_j.id$
10: $BasicAction()$
11: **if** $v_j.tour = true$ **then**
12: $a_h.semiObserve = true$
13: **end if**
14: $a_h.id_3 = v_j.id$
15: **if** $a_h.semiObserve = true$ **then**
16: change its state to an inactive state
17: **end if**
18: **if** $(a_h.id_1 > a_h.id_2 = a_h.id_3) \vee (a_h.id_1 = a_h.id_2 = a_h.id_3)$ **then**
19: change its state to a semi-leader
20: **end if**
21: **if** $a_h.id_2 \quad min(a_h.id_1, a_h.id_3)$ **then**
22: $v_j.inactive = true$ and become inactive
23: **else**
24: **if** $a_h.phase = \lceil \log g \rceil$ **then**
25: $leader\text{-}flag = true$
26: change its state to a leader state
27: **else**
28: $a_h.phase = a_h.phase + 1$
29: **end if**
30: return to step 2
31: **end if**

$a_h.id_1 = a_h.id_2 = a_h.id_3$, then a_h changes its own state to a semi-leader state.

Algorithm 8 represents variable required for the behavior of semi-leader agents. The behavior of semi-leaders until they move around the ring is dedcribed in Algorithm 9, and The behavior of tmeleaderes after they move around the ring is described in Algorithm 10. Each semi-leader agent a_h uses variable $a_h.agentCount$ to detect if a_h moves around the ring or not. a_h uses variable N_{tour} to count the number of tour-flag between two neighboring semi-leaders. a_h stores its phase number in the semi-leader state to variable $a_h.semiPhase$, and v_j stores the phase number to variable $v_j.semiPhase$. These variables are used for the case that there exist a lot of semi-leaders a_h such that $a_h.semi\text{-}leadersInfo = info_{min}$ holds. In addition, a_h use variable $a_h.leaderObserve$ to detect if there exists a leader agent in the ring or not. The initial value of $a_h.leaderObserve$ is false. Moreover, each node v_j has variable $leader\text{-}flag$, $semi\text{-}leader\text{-}frag$, and $tour\text{-}flag$. Before each semi-leader a_h begins moving in the ring, if tour-flag is set at v_j, a_h becomes inactive. This is because, otherwise, each semi-leader cannot share the same $semi\text{-}leadersInfo$.

After each semi-leader moves around the ring, let A_{semi} be a set of semi-leaders a_h that $a_h.semi\text{-}leadersInfo = info_{min}$ holds. If $|A_{semi}| > \lfloor k/g \rfloor$ holds, then there exist less than $g \quad 1$ agent between two agents in A_{semi}. In this case, each semi-leader $a_h \in A_{semi}$ updates its phase and random ID again, and moves around the ring.

Algorithm 8 Values required for the behavior of semi-leader agent a_h (v_j is the current node of a_h)

Variables in Agent a_h
int $a_h.semiPhase$;
int $a_h.agentCount$;
int $a_h.N_{tour}$;
int $a_h.x$;
array $a_h.semi\text{-}leadersInfo[\,]$;
array $info_{min}[\,]$;
boolean $a_h.leaderObserve = false$
Variables in Node v_j
int $v_j, semiPhase$;
int $v_j.id$;
boolean $leader\text{-}flag$;
boolean $semi\text{-}leader\text{-}flag$;
boolean $tour\text{-}flag$;

Then, a_h obtains new value of $a_h.semi\text{-}leadersInfo$. Each semi-leader a_h continues such a behavior until thre exist at most $\lfloor k/g \rfloor$ semi-leader agents a_h that $a_h.semi\text{-}leadersInfo = info_{min}$ holds.

We have the following lemma about Algorithm 7.

Algorithm 9 The rst half behavior of semi-leader agent a_h (v_j is the current node of a_h)

1: **if** $tour\text{-}frag = true$ **then**
2: change its state to an inactive state
3: **end if**
4: $semi\text{-}leader\text{-}flag = true$
5: $a_h.semiPhase = 1$
6: $v_j.semiPhase = a_h.semiPhase$
7: $a_h.x = 0$
8: **while** $a_h.agentCount \neq k$ **do**
9: move to the forward node
10: **while** $v_j.initial = false$ **do**
11: move to the forward node
12: **end while**
13: $a_h.agentCount = a_h.agentCount + 1$
14: **if** $leader\text{-}frag = true$ **then**
15: $a_h.leaderObserve = true$
16: **end if**
17: **if** $semi\text{-}leader\text{-}flag = true$ **then**
18: **if** $a_h.semiPhase \neq v_j.semiPhase$ **then**
19: wait until $a_h.semiPhase = v_j.semiPhase$
20: **end if**
21: $a_h.semi\text{-}leadersInfo[a_h.x] = (v_j.id, a_h.N_{tour})$
22: $a_h.N_{tour} = 0$
23: $a_h.x = a_h.x + 1$
24: **end if**
25: **if** $v_j.tour = false$ **then**
26: $v_j.tour = ture$
27: **end if**
28: $a_h.N_{tour} = a_h.N_{tour} + 1$
29: **end while**

lemma 3.4 *Algorithm 7 eventually terminates, and the con guration satis es the following properties.*

- *There exists at least one leader agent.*
- *There exist at most* $\lfloor k/g \rfloor$ *leader agents.*
- *There exist at least* $g \quad 1$ *inactive agents between two leader agents.*

Algorithm 10 The behavior of semi-leader agent a_h (v_j is the current node of a_h)

1: **if** $a_h.leaderObserve = true$ **then**
2: change its state to an inactive state
3: **end if**
4: let $info_{min}$ be a lexicographically minimum sequence among
$\{shift(a_h.semi\text{-}leaderInfo[\ ],x)|0 \quad x \quad a_h.x \quad 1\}$.
5: **if** $a_h.semi\text{-}leadersInfo \neq info_{min}$ **then**
6: change its state to an ianctive state
7: **end if**
8: let A_{semi} be the number of semi-leader agents a_h that $a_h.semi\text{-}leadersInfo_h = info_{min}$ holds
9: **if** $|A_{semi}| \quad \lfloor k/g \rfloor$ **then**
10: change its state to a leader state
11: **else**
12: $a_h.semiPhase = a_h.semiPhase + 1$
13: $a_h.agentCount = 0$
14: $v_j.ID = random(3 \log k)$
15: return to step 6
16: **end if**

Proof. The above properties are the same to Lemma 1. Thus, if there exist no agents that become semi-leaders during the algorithm, each agent behaves similarly to Section 3.1.1 and above properties are satis ed. In the following, we consider the case that at least one agent becomes a semi-leader .

First, we show that there exists at least one leader agent and there exist at most $\lfloor k/g \rfloor$ leader agents. From line 1 to 3 in Algorithm 10, if there exists a leader agent in the ring, each semi-leader becomes inactive. Otherwise, from line 5 to 16, multiple leaders are elected among A_{semi}. If $|A_{semi}| > \lfloor k/g \rfloor$ holds, then each semi-leader $a_h \in A_{semi}$ continues Algorithm 10 until $|A_{semi}| \quad \lfloor k/g \rfloor$ holds. Since there exists at least one agent in A_{semi} and it does not happen that all agents in A_{semi} become inactive, there exist one to $\lfloor k/g \rfloor$ leader agents.

Next, we show that there exist at least $g \quad 1$ inactive agents between two leaders. As mentioned above, there are at most $\lfloor k/g \rfloor$ leader agents. If there are at least two leaders, the numbers of inactive agents between two leaders are the same because $a_h.semi\text{-}leadersInfo$ is periodic. When there are at most $\lfloor k/g \rfloor$ leaders, the number between two leaders is at least $(k \quad \lfloor k/g \rfloor) \div (\lfloor k/g \rfloor) \quad g \quad 1$. Thus, there exist at least $g \quad 1$ inactive agents between two leaders.

Therefore, we have the lemma.

lemma 3.5 *The expected total moves to execute Algorithm 7 are $O(n \log g)$.*

Proof. If there do not exist neighboring active agents that have the same random IDs, Algorithms 7 works similarly to Section 3.1.1, and the total moves are $O(n \log k)$. In the following, we consider the case that neighboring active agents have the same random IDs.

Let l be the length of a random ID. Then, the probability that two active neighboring active agents have the same random ID is $(\frac{1}{2})^l$. Thus, when there exist k_i acitve agents in the i-th phase, the probability that there exist neighboring active agents that have the same random IDs is at most $k_i \times (\frac{1}{2})^l$. Since at least half active agents drop from candidates in each phase, after executing $\lceil \log g \rceil$ phases, the probability that there exist neighboring active agents that have the same random IDs is at most $k \times (\frac{1}{2})^l + \frac{k}{2} \times (\frac{1}{2})^l + \cdots + \frac{k}{2^{\lceil \log g \rceil - 1}} \times (\frac{1}{2})^l < 2k \times (\frac{1}{2})^l$. Since $l = 3 \log k$ holds, the probability is at most $\frac{2}{k^2} < \frac{1}{k}$. Moreover in this case, at most k agents become semi-leaders and move around the ring. Then, each semi-leader a_h obtains $a_h.semi\text{-}leadersInfo$. If there exist at most $\lfloor k/g \rfloor$ semi-leader agents a_h that $a_h.semi\text{-}leadersInfo = info_{min}$, then agents nish the leader agent election and the total moves are at most $O(kn)$. On the other hand, the probability that there exist more than $\lfloor k/g \rfloor$ semi-leader agents a_h that $a_h.semi\text{-}leadersInfo = info_{min}$ is at most $\frac{1}{k} \times (\frac{1}{2})^{(\lfloor k/g \rfloor + 1) \times l}$. In this case, each semi-leader a_h updates its phase and random ID again, moves around the ring, and obtains new value of $a_h.semi\text{-}leadersInfo$. Each semi-leader a_h continues such a behavior until thre exist at most $\lfloor k/g \rfloor$ semi-leader agents a_h that $a_h.semi\text{-}leadersInfo = info_{min}$ holds. We assume that $t = (\lfloor k/g \rfloor + 1) \times l$ and semi-leaders nish the leader agent election after they move around the ring for the s-th times. The probability that semi-leaders move around the ring s times is at most $\frac{1}{k} \times (\frac{1}{2})^{st}$ and clearly $\frac{1}{k} \times (\frac{1}{2})^{st} < \frac{1}{ks}$ holds. Moreover in this case, the total moves are at most $O(skn)$.

Therefore, we have the lemma.

3.2.2 The second part: leaders' instruction and non-leaders' movement

After executing the leader agent election in Section 3.2.1, the conditions shown by Lemma 3.4 is satis ed, that is, 1) At least one agent is elected as a leader, 2) at most $\lfloor k/g \rfloor$ agents are elected as leaders, and 3) there exist at least $g \quad 1$ inactive agents between two leader agents. Thus, we can execute the algorithms in Section 3.1.2 after the algorithms in Section 3.2.1. Therefore, agents can solve the g-partial gathering problem.

From Lemmas 3.4 and 3.3, we have the following theorem.

Theorem 3.2 *When agents have no IDs, our randomized algorithm solves the g-partial gathering problem in expected $O(gn)$ total moves.*

3.3 Deterministic Algorithm for Anonymous Agents

In this section, we consider a deterministic algorithm to solve the g-partial gathering problem for anonymous agents. At rst, we show that there exist unsolvable initial con gurations in this model. Later, we propose a deterministic algorithm that solves the g-partial gathering problem in $O(kn)$ total moves for any solvable initial con guration.

3.3.1 Existence of Unsolvable Initial Con gurations

To explain unsolvable initial con gurations, we de ne distance sequence of a con guration. For con guration c, we de ne distance sequence of agent a_h as $D_h(c) = (d_0^h(c), \ldots, d_{k\ 1}^h(c))$, where $d_i^h(c)$ is the distance between the i-th forward agent of a_h and the $(i+1)$-th forward agent of a_h in c. Then, we de ne distance sequence of con guration c as the lexicographically minimum sequence among $\{D_h(c) | a_h \in A\}$. We denote distance sequence of con guration c by $D(c)$. For sequence $D = (d_0, d_1, \ldots, d_{k\ 1})$, we de ne $shift(D, x) = (d_x, d_{1+x}, \ldots, d_{k\ 1}, d_0, d_1, \ldots, d_{x\ 1})$. If $D = shift(D, x)$ holds for some x such that $0 < x < k$, we say D is periodic. If D is periodic, we de ne the period of D as $period = \min\{x > 0 | D = shift(D, x)\}$.

Theorem 3.3 *Let c_0 be an initial con guration. If $D(c_0)$ is periodic and period is less than g, the g-partial gathering is not solvable.*

Proof. Let $m = k/period$. Let A_j $(0 \quad j \quad period \quad 1)$ be a set of agents a_h such that $D_h(c_0) = shift(D(c_0), j)$ holds. Then, when all agents move in the synchronous manner, all agents in A_j continue to do the same behavior and thus they cannot break the periodicity of the initial con guration. Since the number of agents in A_j is m and no two agents in A_j stay at the same node, there exist m nodes where agents stay in the nal con guration. However, since $k/m = period < g$ holds, it is impossible that at least g agents meet at the m nodes. Therefore, the g-partial gathering problem is not solvable.

3.3.2 Proposed Algorithm

In this section, for solvable initial con gurations, we propose a deterministic algorithm to solve the g-partial gathering problem in $O(kn)$ total moves. Let $D = D(c_0)$ be the distance sequence of initial con guration c_0 and $period = \min\{x > 0 | D = shift(D, x)\}$. From Theorem 3.3, the g-partial gathering problem is not solvable if $period < g$. On the other hand, our proposed algorithm solves the g-partial gathering problem if $period \quad g$ holds. In this section, we assume that each agent knows the number of agents k.

The idea of the algorithm is as follows: First each agent a_h moves around the ring and obtains the distance sequence $D_h(c_0)$. After that, a_h computes D and $period$. If $period < g$ holds, a_h terminates the algorithm because the g-partial gathering problem is not solvable. Otherwise, agent a_h identi es nodes such that agents in $\{a_\ell | D = D_\ell(c_0)\}$ initially exist. Then, a_h moves to the nearest node among them. Clearly $period(\quad g)$ agents meet at the node, and the algorithm solves the g-partial gathering problem.

Pseudocode. The pseudocode is described in Algorithm 11. The pseudocode describes the behavior of agent a_h, and v_j represents the node where agent a_h currently stays. Agent a_h uses a variable $a_h.total$ to count the number of agent nodes (i.e., nodes v_j with $v_j.initial = true$). If $a_h.total = k$ holds, agent a_h knows it moves around a ring. While agent a_h moves around a ring once, it obtains its distance sequence by variable $a_h.D$. After that a_h computes the distance sequence $D_{min} = D(c_0)$ and $period$. Then, it determines whether the g-partial gathering is solvable or not. If it is solvable, a_h moves to the node to meet other agents.

We have the following theorem about Algorithm 11.

Theorem 3.4 *If the initial con guration is solvable, our algorithm solves the g-partial gathering problem in $O(kn)$ total moves.*

Algorithm 11 The behavior of active agent a_h (v_j is the current node of a_h.)

Variables in Agent a_h
int $a_h.total$;
int $a_h.dis$;
int $a_h.x$;
array $a_h.D[\ ]$;
array $D_{min}[\ ]$;
Main Routine of Agent a_h
1: **while** $a_h.total \neq k$ **do**
2: move to the forward node
3: **while** $v_j.initial = false$ **do**
4: move to the forward node
5: $a_h.dis = a_h.dis + 1$
6: **end while**
7: $a_h.D[a_h.total] = a_h.dis$
8: $a_h.total = a_h.total + 1$
9: $a_h.dis = 0$
10: **end while**
11: let D_{min} be a lexicographically minimum sequence among $\{shift(a_h.D, x)|0 \quad x \quad k \quad 1\}$.
12: $period = \min\{x > 0|shift(D_{min}, x) = D_{min}\}$
13: **if** $(g > period)$ **then**
14: terminate the algorithm
15: // the g-partial gathering problem is not solvable
16: **end if**
17: $a_h.x = \min\{x \quad 0|shift(a_h.D, x) = D_{min}\}$
18: move to the forward node $\sum_{i=0}^{a_h.x'} {}^{1} a_h.D[i]$ times

Proof. At rst, we show the correctness of the algorithm. Each agent a_h moves around the ring, and computes the distance sequence D_{min} and its *period*. If $period < g$ holds, the g-partial gathering problem is not solvable from Theorem 3.3 and a_h terminates the algorithm. In the following, we consider the case that $period \quad g$ holds. From line 18 in Algorithm 11, each agent moves to the forward node $\sum_{i=0}^{a_h.x} {}^{1} a_h.D[i]$ times. By this behavior, each agent a_h moves to the nearest node such that agent a_ℓ with $a_\ell.D = D(c_0)$ initially exists. Since $period(\quad g)$ agents move to the node, the algorithm solves the g-partial gathering problem.

Next, we analyze the total moves required to solve the g-partial gathering problem. In Algorithm 11, all agents move around the ring. This requires $O(kn)$ total number of moves. After this, each agent moves at most n times to meet other agents. This requires $O(kn)$ total moves. Therefore, agents solve the g-partial gathering problem in $O(kn)$ total moves.

4 Partial Gathring in Tree Networks

We consider three model variants. The rst is the weak multiplicity and non-token model. The second is the strong multiplicity and non-token model. The third is the weak multiplicity and removable-token model.

4.1 Weak Multiplicity Detection and Non-Token Model

In this section, we consider the g-partial gathering problem for the weak multiplicity detection and non-token model. We have the following theorem.

Theorem 4.1 *In the weak multiplicity detection and non-token model, there exist no universal algorithms to solve the g-parital gathering problem if* $g \quad 5$ *holds.*

Proof. We show the theorem for the case that g is an odd number (we can also show the theorem for the case that g is an even number). We assume that the tree network is symmetric. In addition, we assume that $3g \quad 1$ agents are placed symmetrically in the initial con guration c_0, that is, if there exists an agent at a node v, there also exists an agent at the node v', where v and v' are symmetric. Later, we assume that each pair of nodes v_1 and $v_{1'}$, v_2 and $v_{2'}$,... is symmetric. Note that, since $2g \quad k \quad 3g \quad 1$ holds, agents are allowed to meet at one or two nodes. In the proof, we consider a waiting state of agents as follows. When an agent a is in the waiting state at node v, a never leaves v before the con guration for a changes. Concretely, there are two cases. The rst case is that when a visits the node v and enters a waiting state at v, there exist no other agents

at v. In this case, a does not leave v before another agent visits v and stay there. The second case is that when a visits v and enters a waiting state at v, there exists another waiting agent b at v. In this case, a does not leave v before b leaves v. In any algorithm, it is necessary that each agent enters a waiting state. In there exists some agent a that does not enter a waiting state, a moves in the tree network forever or terminates the algorithm at some node. If there exists an agent that does not enter a waiting state and terminates the algorithm at some node v, there also exists an agent that does not enter a waiting state and terminates the algorithm at the node v', where v and v' are symmetric. In addition, at least g agents must meet at v and v' respectively. However, if the location of agents in the initial con guration is not symmetric, it may happen that less than g agents meet at v or v' and agents do not the satisfy the condition of the g-partial gathering problem.

We consider the execution E_t that each agent moves symmetrically and when some agent a enters in a waiting state at a node v, a does not leave v even if another agent enters a waiting state at v. Each agent continues such a behavior until all agents enter in a waiting states, and we de ne c_t as the con guration that all agents' states are waiting states from c_0. In c_t, since agents are initially placed symmetrically and move symmetrically, if there exist l waiting agents at the node v_j, there also exist l waiting at the node $v_{j'}$. Let $v_1, v_2, \ldots, v_t$ $(v_{1'}, v_{2'}, \ldots v_{t'})$ be nodes where at least one agent exists in c_t. In addition, let N_l be the number of waiting agents at v_l in c_t. Note that, $N_1 + N_2 + \cdots + N_t = k/2$ holds and we assume that $N_1 \quad N_2 \quad \cdots \quad N_t$ holds. Moreover, we assume that agents $a_1^j, a_2^j, \ldots, a_{N_j}^j$ enter waiting states at v_j in this order. Then, we have the following two lemmas.

lemma 4.1 *At some node v_j with exactly one waiting agent a_1^j, a_1^j never leaves v_j before another agent enters a waiting state at v_j.*

Proof. When a_1^j enters a waiting state at v_j, there exist no other waiting agents at v_j. Thus, the con guration for a_1^j does not change unless another agent enters a waiting state at v_j.

lemma 4.2 *At some node v_j with at least three waiting agents, at least two agents never leave v_j by the end of the algorithm.*

Proof. We assume that agents a_1^j, a_2^j, a_3^j enter waiting states at v_j in this order. Since a_1^j is the rst agent that enters a waiting state at v_j, when a_2^j enters a waiting state at v_j, the con guration for a_1^j changes, and a_1^j can leave v_j. However, since we consider the weak multiplicity detection model, even if a_1^j leaves v_j, the con gurations for a_2^j and a_3^j do not change. Thus, agents a_2^j and a_3^j never leave v_j.

There are eight patterns to assign values to $N_1, N_2, \ldots, N_t$ $(N_{1'}, N_{2'}, \ldots, N_{t'})$. In the following, we show that agents cannot solve the g-partial gathering problem in any pattern.

⟨pattern 1: for the case that $N_2 \quad 3$ holds⟩
In this case, there exist at least three waiting agents at $v_1, v_2, v_{1'}$ and $v_{2'}$ respectively. Hence, from Lemma 4.2, there exist at least four nodes with agents that never leave the current nodes. However, since $k = 3g \quad 1$ holds, agents are allowed to meet at one or two nodes. This implies that agents cannot solve the g-partial gathering problem.

⟨pattern 2: for the case that $N_1 = N_2 = \cdots = N_t = 1$ holds⟩
In this case, there exist no nodes with more than one agent. Hence from Lemma 4.1, the con guration of each agent does not change and each agent never leaves the current node. This implies that agents cannot solve the g-partial gathering problem.

Before considering the pattern 3, we introduce the notion of elimination. Let c_0' be the initial con guration that there do not exist agents a_i^j $(i \neq 1)$, and the other elements (the topology and the location of agents) are the same to c_0. Then, we say that agents a_i^j are eliminated from c_0. Note that, since $k = 3g \quad 1$ and $2g \quad k$ holds, at most $g \quad 1$ agents can be eliminated. Moreover, we consider the execution E_t' similarly to E_t, that is, each agents moves symmetrically and when an agent a enters a waiting state at the node v, a never leaves v until all agents enter waiting states. We de ne c_t' as the con guration that all agents' states are waiting states form c_0'. Then, we have the following lemma.

lemma 4.3 *The location of agents in c_t' is equal to the location of agents in c_t except for agents a_i^j.*

Proof. The proof consists of two parts. The rst part is before one a_i^j enters a waiting state and the second part is after a_i^j enters a waiting state.

Before a_i^j enters a waiting state, a_i^j moves in the tree network. Hence, it does not happen that the other agents observe a_i^j because agents detect the existence of another agent only at nodes. Therefore, even if a_i^j is eliminated, the other agents behave similarly to E_t.

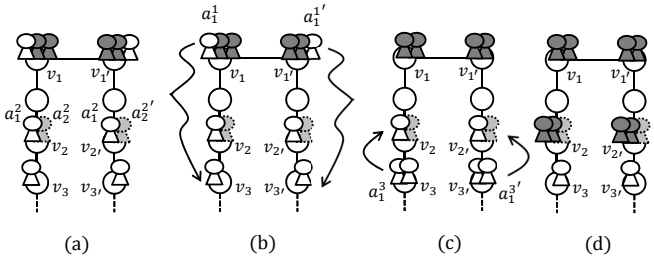


Figure 9: An example of the pattern 3

When a_i^j enters a waiting state at v_j, there already exist a waiting agent a_1^j. Since we consider the weak multiplicity detection model, when another agent a_k^j $(k > i)$ visits the node v_j, the con guration for a_k^j at v_j with agents a_1^j and a_i^j is equal to the con guration at v_j with an agent a_1^j. Thus, even if a_i^j is eliminated in c'_0, a_k^j also enters a waiting state at v_j in c'_t.

Therefore, we have the lemma.

we use these lemmas to show the unsolvability of the remaining patterns.

⟨pattern 3: for the case that N_1 3 and $N_2 = 2$ hold⟩
In this case, there exist waiting agents a_1^1, a_2^1, and a_3^1 ($a_1^{1'}, a_2^{1'}$, and $a_3^{1'}$) at v_1 ($v_{1'}$), and a_2^1 and a_3^1 ($a_2^{1'}$ and $a_3^{1'}$) never leave v_1 ($v_{1'}$) by Lemma 4.2. Since $k = 3g$ 1 holds and agents are allowed to meet at one or two node, all agents must meet at v_1 or $v_{1'}$.

Now let us consider the con guration c'_0 that agents a_2^2 and $a_2^{2'}$ are eliminated. Then from Lemma 4.3, there exists an execution E'_t from c'_0 to c'_t, where there exists exactly one waiting agent a_1^2 ($a_1^{2'}$) at v_2 ($v_{2'}$) in c'_t. An example is represented in Fig. 9 (a). In the gure, we assume that agents a_2^2 and $a_2^{2'}$ of the dotted lines are eliminated. In addition, the gray agents $a_2^1, a_3^1, a_2^{1'}, a_2^{1'}$ mean that they never leave the current nodes by the end of the algorithm. From Lemma 4.1, agents need to make the con guration c'_u that another agent a (a') enters a waiting state at v_2 ($v_{2'}$), and call such an execution E'_u. In the gure, we assume that agents a_1^1 and $a_1^{1'}$ move symmetrically and enter waiting states at v_3 and $v_{3'}$ respectively (Fig. 9 (b)), and after this, agents a_1^3 and $a_1^{3'}$ move symmetrically (Fig. 9 (c)) and enter waiting states at v_2 and $v_{2'}$ respectively (Fig. 9 (d)).

Now, let us consider the con guration c_t. In c_t, their exist two waiting agents a_1^2 and a_2^2 ($a_1^{2'}$ and $a_2^{2'}$) at v_2 ($v_{2'}$). In addition, since a_1^2 ($a_1^{2'}$) is the rst agent that enters a waiting state at v_2 ($v_{2'}$), a_1^2 ($a_1^{2'}$) can leave v_2 ($v_{2'}$). However since agents move asynchronously, there exists an execution similarly to E'_u, that is, agents a_1^2 ($a_2^{2'}$) does not leave v_2 ($v_{2'}$) until another agent a (a') enters a waiting state at v_2 ($v_{2'}$). Then, there exist three waiting agents a_1^2, a_2^2 and a ($a_1^{2'}, a_2^{2'}$ and a') at v_2 ($v_{2'}$) like Fig. 9. From Lemma 4.2, agents a_2^2 and a ($a_2^{2'}$ and a') never leave v_2 ($v_{2'}$). This means that there exist at least four nodes with agents that never leave at the current nodes and agents cannot solve the g-partial gathering problem.

From the pattern 4 to pattern 6, we consider cases that exist at least three waiting agents at v_1 and $v_{1'}$, and there exists at most one waiting agent at the other nodes.

⟨pattern 4: for the case that 3 N_1 $(g+1)/2$, and $N_2 = 1$ hold⟩
In this case, there exist several waiting agents at v_1 and $v_{1'}$, and there exist at most one waiting agents at the other nodes. Let us the consider the con guration c'_0 that agents $a_2^1, \ldots, a_{N_1}^1, a_2^{1'}, \ldots, a_{N_{1'}}^{1'}$ are eliminated. Note that, the number of eliminated agents $a_2^1, \ldots, a_{N_1}^1, a_2^{1'}, \ldots, a_{N_{1'}}^{1'}$ is $2N_1$ 2 g 1 since N_1 $(g+1)/2$ holds. Then from Lemma 4.3, there exist an execution E'_t from c'_0 to c'_t, where at most one waiting agent at each node in c'_t. This con guration is similarly to the pattern 2 and agents cannot solve the g-partial gathering problem.

⟨pattern 5: for the case that $(g+3)/2$ N_1 g, and $N_2 = 1$ hold⟩
In this case, let us the consider the initial con guration c'_0 that agents $a_2^1, \ldots, a_{N_1}^1$ are eliminated like Fig. 10 (a). Note that, the number of eliminated agents $a_2^1 \ldots, a_{N_1}^1$ are N_1 1 g 1 since N_1 g holds. Then from lemma 4.3, there exist an execution E'_t from c'_0 to c'_t, where there exist $N_{1'}$ waiting agents at $v_{1'}$ and there exist at most one waiting agent at the other nodes in c'_t. In this con guration, rstly agent $a_1^{1'}$ needs to leave $v_{1'}$ and enter a waiting state at another node $v_{j'}$. In addition, agents need to make the con guration c'_u that some agent a' enters a waiting state at v_j, and we de ne E'_u as an execution from c'_t to c'_u. In the gure, we assume that an agent $a_1^{1'}$ moves and enters a waiting state at $v_{3'}$ (Fig. 10 (b)), and after this, an agent $a_1^{3'}$ moves and enters a waiting state at v_3 (Fig. 10 (c)).

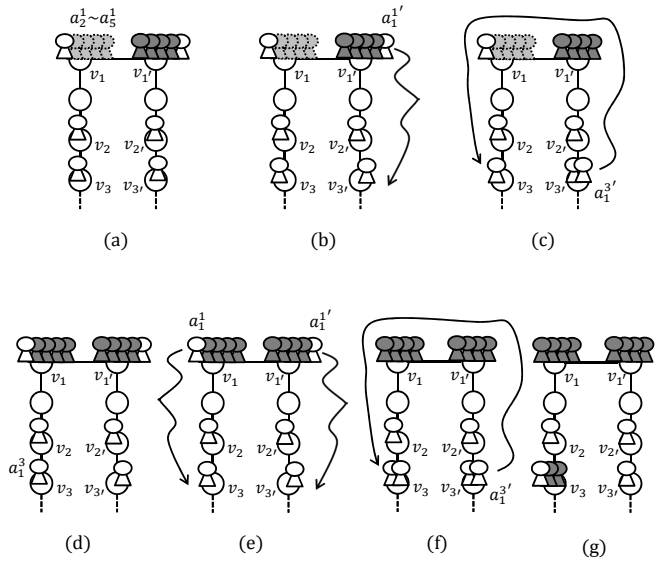


Figure 10: An example of the pattern 5

Now let us consider c_0 like Fig. 10 (d) and we assume that a_1^1 and $a_1^{1'}$ behave symmetrically until they enter waiting states at v_j and $v_{j'}$ respectively. Then, the con gurations for a_1^j and $a_1^{j'}$ change and they can leave v_j and $v_{j'}$ respectively. However, there exist an execution similarly to E'_u, that is, agent a_1^j does not leave the node v_j, agent $a_1^{j'}$ leaves $v_{j'}$, and some agent a' enters a waiting state at v_j Then, there exist three waiting agent a_1^j, a_1^1, and a' at v_j. From Lemma 4.2, agents a_1^1 and a' never leave v_j. In the gure, agents a_1^1 and $a_1^{1'}$ move and enter waiting states at v_3 and $v_{3'}$ respectively (Fig. 10 (e)), and after this, an agent $a_1^{3'}$ moves and enters a waiting state at v_3 (Fig. 10 (f)). Then in Fig. 10 (g), agents a_1^3, a_1^1, and $a_1^{3'}$ are in the waiting states. Note that, agents a_2^1, a_3^1 ($a_2^{1'}, a_3^{1'}$) also never leave v_1 ($v_{1'}$). This means there exist at least three nodes with agent that never leave the current nodes and agents cannot solve the g-partial gathering problem.

⟨pattern 6: for the case that 3 N_1 g 1, and $N_2 = 1$ hold⟩
In this case, there exist an execution E_t that agents moves symmetrically from c_0 to c_t and $(3g$ $1)/2$ agents meet at v_1 and $v_{1'}$ respectively.

Now let us consider the initial con guration c'_0 that agents $a_3^1, \ldots, a_{3+(g\ \ 1)/2}^1$ are eliminated. Then, there exist an execution similarly to E_t, that is, agents moves symmetrically and each agent meets at v_1 or $v_{1'}$. However, $(g+1)/2$ agents $a_3^1, \ldots, a_{3+(g\ \ 1)/2}$ are eliminated, the number of agents that meet at a_1 is $(3g$ $1)/2$ $(g+1)/2 = g$ 1. This does not satisfy the condition of the g-partial gathering problem.

In the pattern 7 and 8, we consider the case that there exist at most two waiting agents at each node.

⟨pattern 7: for the case that $N_1 = N_2 = 2$ and $N_3 = 1$ holds⟩
In this case, there are two agents at $v_1, v_2, v_{1'}$, and $v_{2'}$, and there exist at most one agent at the other nodes. Now, let us consider the initial con guration c'_0 that agents $a_2^1, a_2^2, a_2^{1'}$, and $a_2^{2'}$ are eliminated. Then from Lemma 4.3, there exist an execution E'_t from c'_0 to c'_t, where there exist at most one waiting agent at each node in c'_t. This con guration is similarly to the pattern 2 and agents cannot solve the g-partial gathering problem.

⟨pattern 8: for the case that $N_1 = N_2 = N_3 = 2$ holds⟩
In this case, we consider the initial con guration c'_0 that agents a_1^2 and $a_1^{2'}$, are eliminated. Then from Lemma 4.3, there exist an execution E'_t from c'_0 to c'_t, where there exists exactly one waiting agent a_1^2 ($a_1^{2'}$) at v_2 ($v_{2'}$) in c'_t like Fig. 11 (a). In addition from Lemma 4.1, agents need to make the con guration c'_u from c'_t, where some agent enters a waiting state at v_2 ($v_{2'}$) in c'_u. We assume that a_1^1 and a_1^1 leave v_1 and $v_{1'}$, behave symmetrically, and some agents a and a' enter waiting states at v_3 and $v_{3'}$ respectively. We call such an execution E''_u. In the gure, we assume that a_1^1 and $a_1^{1'}$ directly enter waiting states respectively (Fig. 11 (b)).

Now let us consider c_t like Fig.11 (e). In c_t, there also exists an execution similarly to E'_u, that is, agent a_1^2 ($a_1^{2'}$) does not leave v_2 ($v_{2'}$), agent a_1^1 ($a_1^{1'}$) leaves v_1 ($v_{1'}$), and some agent a (a') enters a waiting state at v_2 ($v_{2'}$) in nite time. Then, there exist three waiting agent a_1^2, a_2^2, and a ($a_1^{2'}, a_2^{2'}$, and a'). From Lemma 4.2,

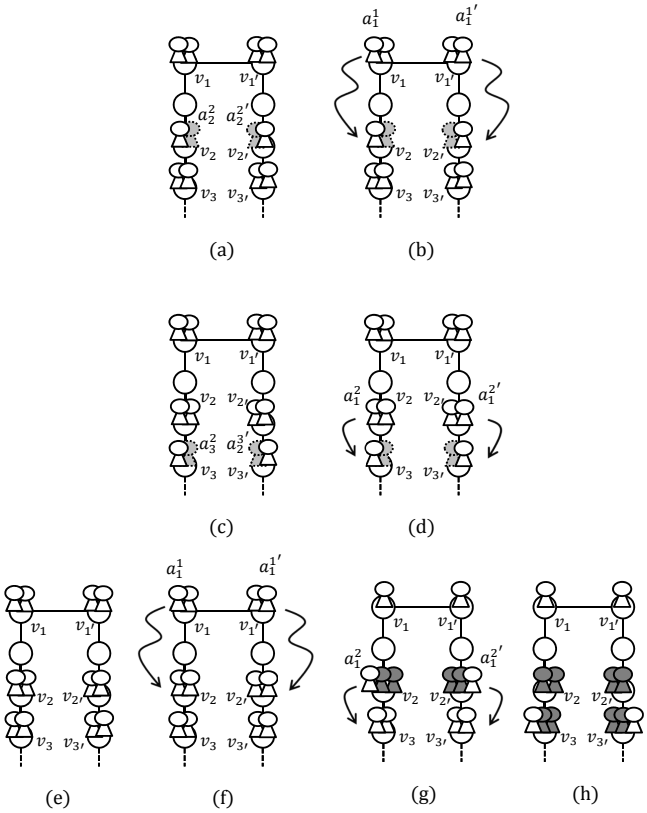


Figure 11: An example of the pattern 8

agents a_2^2 and a ($a_2^{2'}$ and a') never leave v_2 ($v_{2'}$), and we call this con guration c_u. In the gure, we assume that agents a_1^1 and $a_1^{1'}$ enter waiting states respectively (Fig. 11 (f)). Then, agents $a_2^2, a_1^1, a_2^{2'}$, and $a_1^{1'}$ never leave current nodes.

Next, let us consider another initial con guration c_0'' that agents a_2^3 and $a_2^{3'}$ are eliminated. Then from Lemma 4.3, there exists an con guration E_t'' from c_0'' to c_t'', where there exists exactly one waiting agent a_1^3 ($a_1^{3'}$) at v_3 ($v_{3'}$) in c_t'' like Fig. 11 (c). In addition from Lemma 4.1, agents need to make the con guration c_u'' from c_t'', where some agent enters a waiting state at v_3 and $v_{3'}$ in c_u''. We assume that a_1^2 and $a_1^{2'}$ leave v_2 and $v_{2'}$, behave symmetrically, and some agents b and b' enter waiting states at v_3 and $v_{3'}$ respectively. We call such an execution E_u'' In the gure, we assume that agents a_1^2 and $a_1^{2'}$ directly enter waiting state at v_3 and $v_{3'}$ respectively (Fig. 11 (d)).

Now let us consider c_u. In c_u, there also exists an execution similarly to E_u'', that is, agent a_1^2 ($a_1^{2'}$) leaves v_2 ($v_{2'}$) and agent b (b') enters a waiting state at v_3 ($v_{3'}$) in nite time. Then, there exist three waiting agent a_1^3, a_2^3, and b ($a_1^{3'}, a_2^{3'}$, and b'). From Lemma 4.2, agents a_2^3 and b ($a_2^{3'}$ and b') never leave v_3 ($v_{3'}$). In the gure, we assume that agents a_1^2 and $a_1^{2'}$ move symmetrically (Fig. 11 (g)), and enter waiting state at v_3 and $v_{3'}$ respectively (Fig. 11 (h)). Thus, there exist four nodes with agents that never leave the current node. This implies that agents cannot solve the g-partial gathering problem.

Therefore, we have the theorem.

4.2 Strong Multiplicity Detection and Non-Token Model

In this section, we consider a deterministic algorithm to solve the g-partial gathering problem for the strong multiplicity detection and non-token model. First, we have the following theorem.

Theorem 4.2 *In the strong multiplicity detection and non-token model, agents require $\Omega(kn)$ total moves to solve the g-partial gathering problem even if agents know k.*

Proof. To show the theorem by contradiction, we assume that there exists an algorithm A to solve the g-partial gathering problem in $o(kn)$ total moves. In the proof, we say agent a is in a waiting state at node v iff a never leaves v before another agent visits v. First, we claim that some agent needs to enter a waiting state at some node. If there exists no such agent, every agent can leave a node before another agent visits the node.

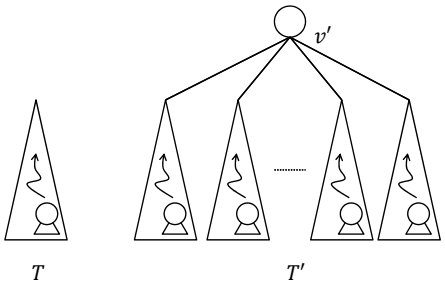


Figure 12: Figures of T and T'

This implies, since agents move asynchronously, agents never meet other agents. Consequently, such a behavior cannot solve the g-partial gathering problem. Hence, there exists an agent that enters a waiting state at some node.

Next, let us consider the initial con guration c_0 such that k agents are placed in tree T with n nodes. We claim that some agent enters a waiting state in $o(n)$ moves without meeting other agents. Consider the execution that repeats a phase in which 1) every agent not in a waiting state makes a movement, 2) visits a node, and 3) before another agent comes, it leaves the node unless it enters a waiting state. Clearly each agent does not meet other agents unless it enters a waiting state. Let a_i be the agent that rstly enters a waiting state in this execution. If a_i moves $\Omega(n)$ times before it enters a waiting state, all other agents move $\Omega(n)$ times. This implies the total moves is $\Omega(kn)$, which contradicts to the assumption of A. Hence, a_i enters a waiting state in $o(n)$ moves without meeting other agents. This implies there exists a node v_x such that a_i does not visit before it enters a waiting state. In addition, we assume that a_i is placed at the node v_w in the initial con guration c_0.

Next, we construct tree T' with $kn'+1$ nodes as follows: Let $T^1, \ldots, T^k$ be k trees with the same topology as T and v_x^j ($1 \quad j \quad k$) be the node in T^j corresponding to v_x in T. Tree T' is constructed by connecting a node v' to v_x^j for every j (Fig. 12). Let v_w^j ($1 \quad j \quad k$) be the node in T^j corresponding to v_w in T. Consider the con guration c_0' such that k agents are placed at $v_w^1, v_w^2, \ldots, v_w^k$ respectively. Since agents do not have knowledge of n, each agent does the same behavior as a_i in T (note that they do not visit v_x^j). Hence, each agent placed in T^j ($1 \quad j \quad k$) enters a waiting state without moving out of T^j. Thus, each agent enters a waiting state at different nodes and does not resume the behavior. Therefore, algorithm A cannot solve the g-partial gathering problem in T'. This is a contradiction.

Next, we propose a deterministic algorithm to solve the g-partial gathering problem in $O(kn)$ total moves for the strong multiplicity detection and non-token model for the case $g \quad k/2$. Remind that, in the strong multiplicity detection model, each agent can count the number of agents at the current node. After starting the algorithm, each agent performs *a basic walk* [7]. In the basic walk, each agent a_h leaves the initial node through the port 0. Later, when a_h visits a node v_j through the port p, a_h leaves v_j through the port $(p+1)$ mod d_{v_j}. In the basic walk, each agent traverses the tree in the DFS-traversal. Hence, when each agent visits nodes $2(n \quad 1)$ times, it visits the all nodes in the tree and returns to the initial node. Note that, we assume that agents do not know the number n of nodes. However, if an agent records the topology of the tree every time it visits nodes, it can know the time when it returns to the initial node.

The idea of the algorithm is as follows: First, each agent performs the basic walk until it obtains the whole topology of the tree. Next, each agent computes a center node of the tree and moves there to meet other agents. If the tree has exactly one center node, then each agent moves to the center node and terminates the algorithm. If the tree has two center nodes, then each agent moves to one of the center nodes so that at least g agents meet at each center node. Concretely, agent a_h rst moves to the closer center node v_j. If there exist at most g agents at v_j, including a_h, then a_h terminates the algorithm at v_j. Otherwise, a_h moves to another center node $v_{j'}$ and terminates the algorithm.

The pseudocode is described in Algorithm 12. We have the following theorem.

Theorem 4.3 *In the strong multiplicity detection and non-token model, agents solve the g-partial gathering problem in $O(kn)$ total moves.*

Proof. At rst, we show the correctness of the algorithm. From Algorithm 12, if the tree has one center node, agents go to the center node and agents solve the g-partial gathering problem obviously. Otherwise, each agent a_h rst moves to one of center nodes. If there exist at least g agents at the center node, a_h moves to another center node. Since $k \quad 2g$ holds, agents can solve the g-partial gathering problem.

Next, we analyze the total moves to solve the g-partial gathering problem. At rst, agents perform the basic walk and record the topology of the tree. This requires at most $2(n \quad 1)$ total moves for each agent. Next, each

Algorithm 12 The behavior of active agent a_h (v_j is the current node of a_h.)

Main Routine of Agent a_h

```
perform the the basic walk until it obtains the whole topology of the tree
if there exists exactly one center node then
    go to the center node via the shortest path and terminate the algorithm
else
    go to the closest center node via the shortest path
    if there exist at most g agents then
        terminate the algorithm
    else
        move to another center node
        terminate the algorithm
    end if
end if
```

agent moves to one of the center nodes, and terminates the algorithm. This requires at most $\frac{n}{2} + 1$ moves for each agent. Hence, each agent requires $O(n)$ total moves to solve the g-partial gathering problem. Therefore, agents require $O(kn)$ total moves.

4.3 Weak Multiplicity Detection and Removable-Token Model

In this section, we propose a deterministic algorithm to solve the g-partial gathering problem for the weak multiplicity detection and removable-token model. We show that our algorithm solves the g-partial gathering problem in $O(gn)$ total moves. Remind that, in the removable-token model, each agent has a token. In the initial con guration, each agent leaves a token at the initial node. We de ne *a token node* (resp., *a non-token node*) as a node that has a token (resp., does not have a token). In addition, when an agent visits a token node, the agent can remove the token.

The idea of the algorithm is similar to Section 3.1, but in Section 3.1, the network is a unidirectional ring. In this section, we make agents perform the basic walk and regard a tree network as a unidirectional ring network. Concretely, if agent a_h starts the basic walk at node v_0 and continues it until a_h visits nodes $2(n \quad 1)$ times, then each communication link is passed twice and a_h returns to v_0. Thus, when a_h visits nodes $v_1, v_2, \ldots, v_{2(n \quad 1)}$ in this order, then we consider that a_h moves in the unidirectional ring network with $2(n \quad 1)$ nodes. Later, we call this ring *the virtual ring*. In the virtual ring, we de ne the direction from v_i to v_{i+1} as a *forward* direction, and the direction from v_{i+1} to v_i as a *backward* direction. Moreover, when a_h visits a node v_j through a port p from a node $v_{j \quad 1}$ in the virtual ring, agents also use p as the port number at $(v_{j \quad 1}, v_j)$. For example, let us consider a tree in Fig. 13(a). Agent a_h performs the basic walk and visits nodes a, b, c, b, d, b in this order. Then, the virtual ring of Fig. 13(a) is represented in Fig. 13(b). Each number in Fig. 13(b) represents the port number through which a_h visits each node in the virtual ring. Next, we de ne a token node in a virtual ring as follows. First, the initial token node in the tree network is also the token node in the virtual ring. In addition, when agent a_h visits a token node v_j in the tree, we de ne that a_h visits a token node in the virtual ring if it visits v_j through the port $(d_{v_j} \quad 1)$. In Fig. 13(a), if nodes a and b are token nodes, then in Fig. 13(b), nodes a and b'' are token nodes. By this de nition, a token node in the tree network is mapped to one token node in the virtual ring. Thus, by performing the basic walk, we can assume that each agent moves in the same virtual ring. Moreover, in the virtual ring, each agent also moves in a FIFO manner, that is, when an agent a_h leaves some node v_j before another agent a_i leaves v_j, a_h takes a step before a_i does it.

The algorithm consists of two parts. In the rst part, agents execute the leader agent election partway and elect some leader agents. In the second part, leader agents instruct the other agents which node they meet at, and the other agents move to the node by the instruction. In the following section, we explain the algorithm by using a virtual ring.

4.3.1 The rst part: leader election

In the leader agent election, the states of agents are divided into the following three types:

- *active*: The agent is performing the leader agent election as a candidate of leaders.
- *inactive*: The agent has dropped out from the candidate of leaders.
- *leader*: The agent has been elected as a leader.

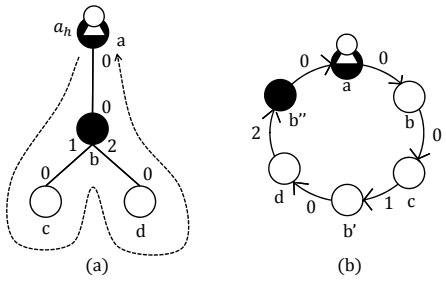


Figure 13: An example of the basic walk

The aim of the rst part is similar to Section 3.1.1, that is, to elect some leaders and satisfy the following three properties: 1) At least one agent is elected as a leader, 2) at most $\lfloor k/g \rfloor$ agents are elected as leaders, and 3) in the virtual ring, there exist at least g 1 inactive agents between two leader agents.

In Section 3.1.1, each agent is distinct and each node has whiteboard. However, in this paper, we assume that each agent is anonymous and some nodes have tokens. First, we explain the treatment about IDs. For explanation, let *active nodes* be nodes where active agents start execution of each phase. In this section, agents use *virtual IDs* in the virtual ring. Concretely, when agent a_h moves from an active node v_j to v_j's forward active node $v_{j'}$, a_h observes port sequence $p_1, p_2, \ldots p_l$, where p_m is the port number through which a_h visits the node by the m-th movement after leaving v_j. In this case, a_h uses this port sequence $p_1, p_2, \ldots p_l$ as its virtual ID. For example, in Fig. 13(b), when a_h moves from a to b'', a_h observes the port numbers $0, 0, 1, 0, 2$ in this order. Hence, a_h uses 00102 as a virtual ID from a to b''. Similarly, a_h uses 0 as a virtual ID from b'' to a. Note that, multiple agents may have the same virtual IDs, and we explain the behavior in this case later. Next, we explain the treatment about whiteboards. In Section 3.1.1, each node has a whiteboard, while in this paper, each node is allowed to have an only token. Fortunately, we can easily overcome this problem by using virtual IDs. Concretely, each active agent a_h moves until a_h visits three active nodes. Then, a_h observes its own virtual ID, the virtual ID of a_h's forward active agent a_i, and the virtual ID of a_i's forward active agent a_j. Thus, a_h can obtain three virtual IDs id_1, id_2, id_3 without using whiteboards. Therefore, agents can use the above approach [24], that is, a_h behaves as if it would be an active agent with ID id_2 in bidirectional rings. In the rest of this paragraph, we explain how agents detect active nodes. In the beginning of the algorithm, each agent starts the algorithm at a token node and all token nodes are active nodes. After each agent a_h visits three active nodes, a_h decides whether a_h remains active or drops out from the candidate of leaders at the active (token) node. If a_h remains active, then a_h starts the next phase and leaves the active node. Thus, in some phase, when some active agent a_h visits a token node v_j with no agent, a_h knows that a_h visits an active node and the other nodes are not active nodes in the phase.

After observing three virtual IDs id_1, id_2, id_3, each active agent a_h compares virtual IDs and decides whether a_h remains active (as a candidate of leaders) in the next phase or not. Different from Section 3.1.1, multiple agents may have the same IDs. To treat this case, if $id_2 < \min(id_1, id_3)$ or $id_2 = id_3 < id_1$ holds, then a_h remains active as a candidate of leaders. Otherwise, a_h becomes inactive and drops out from the candidate of leaders. For example, let us consider the initial con guration like Fig. 14(a). In the gure, black nodes are token nodes and the numbers near communication links are port numbers. The virtual ring of Fig. 14(a) is represented in Fig. 14(b). For simplicity, we omit non-token nodes in Fig. 14(b). The numbers in Fig. 14(b) are virtual IDs. Each agent a_h continues to move until a_h visits three active nodes. By the movement, a_1 observes three virtual IDs (01,01,01), a_2 observes three virtual IDs $(01, 01, 1000101010)$, a_3 observes three virtual IDs $(01, 1000101010, 01)$, and a_4 observes three virtual IDs (1000101010,01,01) respectively. Thus, a_4 remains as a candidate of leaders, and a_1, a_2, and a_3 drop out from the candidates of leaders. Note that, like Fig. 14, if an agent observes the same virtual IDs three times, it drops out from the candidate of leaders. This implies, if all active agents have the same virtual IDs, all agents become inactive. However, we can show that, when there exist at least three active agents, it does not happen that all active agents observe the same virtual IDs. Moreover, if there are only one or two active agents in some phase, then the agents notice the fact during the phase. In this case, the agents immediately become leaders. By executing $\lceil \log g \rceil$ phases, agents complete the leader agent election.

Pseudocode. The pseudocode to elect leaders is given in Algorithm 13. All agents start the algorithm with active states. The pseudocode describes the behavior of active agent a_h, and v_j represents the node where agent a_h currently stays. If agent a_h becomes an inactive state or a leader state, a_h immediately moves to the next part and executes the algorithm for an inactive state or a leader state in section 4.3.2. Agent a_h uses variables id_1, id_2, and id_3 to store three virtual IDs. Variable *phase* stores the phase number of a_h. In Algorithm 13,

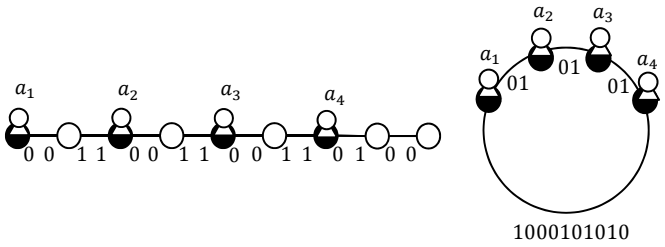


Figure 14: An example that agents observe the same port sequence

Algorithm 13 The behavior of active agent a_h (v_j is the current node of a_h.)

Variables in Agent a_h
int $phase = 0$;
int id_1, id_2, id_3;
Main Routine of Agent a_h
1: $phase = phase + 1$
2: $id_1 = NextActive()$
3: $id_2 = NextActive()$
4: $id_3 = NextActive()$
5: **if** there exist at most two active agents in the tree **then**
6: change its state to a leader state
7: **end if**
8: **if** $(id_2 < min(id_1, id_3)) \vee (id_2 = id_3 < id_1)$ **then**
9: **if** $(phase = \lceil \log g \rceil)$ **then**
10: change its state to a leader state
11: **else**
12: return to line 1
13: **end if**
14: **else**
15: change its state to an inactive state
16: **end if**

each active agent a_h moves until a_h observes three virtual IDs and decides whether a_h remains active as a candidate of leaders or not on the basis of virtual IDs. Note that, since each agent moves in a FIFO manner, it does not happen that some active agent passes another active agent in the virtual ring, and each active agent correctly observes three neighboring virtual IDs in the phase. In Algorithm 13, a_h uses procedure *NextActive*(), by which a_h moves to the next active node and returns the port sequence as a virtual ID. The pseudocode of *NextActive*() is described in Algorithm 14. Agent a_h uses variable *port* to store a virtual ID while moving, and a_h uses variable *move* to store the number of nodes it visits. Note that, if there exist only one or two active agents in some phase, then the agent moves around the virtual ring before getting three virtual IDs. In this case, the active agent knows that there exist at most two active agents in the phase and they become leaders. To do this, agents record the topology every time they visit nodes, but we omit the description of this behavior in Algorithm 13 and Algorithm 14.

First, we show the following lemma to show that at least one agent remains active or becomes a leader in each phase.

lemma 4.4 *When there exist at least three active agents, at least one agent has a virtual ID different from another agent.*

Proof. To show the lemma, we use the theorem from [5]. Let $t[1..q]$ be a port sequence that an agent observes in visiting q nodes by performing the basic walk. In our algorithm, $t[1..q]$ represents a virtual ID that the agent uses from a active node to the next active node. Moreover, we de ne $(t[1..q])^k$ as the concatenation of k copies of $t[1..q]$. In addition, the *length* of an n-node tree T is the length of its Euler tour, that is, $2(n \quad 1)$. Then, we use the following theorem.

Theorem 4.4 *Let T be a tree of length at least $q \quad 1$. Assume that $t[1..q]$ is not periodic and $t[1..kq] = (t[1..q])^k$ for some $k \quad 3$. Then one of the following three cases must hold [5].*

Algorithm 14 int *NextActive*() (v_j is the current node of a_h.)

Main Routine of Agent a_h
array *port*[ ];
int *move*;
Main Routine of Agent a_h
1: $move = 0$
2: leave v_j through the port 0
// arrive at the forward node
3: let p be the port number through which a_h visits v_j
4: $port[move] = p$
5: $move = move + 1$
6: **while** (there does not exist a token) $\vee$
($p \neq d_{v_j}$ 1) $\vee$ (there exists another agent) **do**
7: leave v_j through the port $(p + 1) \mod d_{v_j}$
// arrive at the forward node
8: let p be the port number through which a_h visits v_j
9: $port[move] = p$
10: $move = move + 1$
11: **end while**
12: return *port*[ ]

1. *The length of T is q.*
2. *The length of T is $2q$.*
3. *The length of T is greater than kq.*

We show the lemma by contradiction, that is, assume that there exist k' 3 active agents in some phase and all k' active agents have the same virtual IDs. Let x be the virtual ID. Let us consider some agent that starts the basic walk at a node r and continues until it returns to r. Then, $t[1..k'|x|] = (t[1..|x|])^{k'}$ holds and the length of the tree is $k'|x|$. However, from Theorem 4.4, the length of the tree is never $k'|x|$. This is a contradiction.

Next, we have the following lemmas about Algorithm 13.

lemma 4.5 *Algorithm 13 eventually terminates, and satis es the following three properties.*

- *There exists at least one leader agent.*
- *There exist at most $\lfloor k/g \rfloor$ leader agents.*
- *In the virtual ring, there exist at least g 1 inactive agents between two leader agents.*

Proof. We show the lemma in the virtual ring. Obviously, Algorithm 13 eventually terminates. In the following, we show the above three properties.

At rst, we show that there exists at least one leader agent. From lines 5-7 of Algorithm 13, when there exist only one or two active agents in some phase, the agents become leaders. When there are at least three active agents in some phase, if $a_h.id_2 < \min(a_h.id_1, a_h.id_3)$ or $a_h.id_2 = a_h.id_3 < a_h.id_1$ holds, agent a_h remains as a candidate of leaders, and otherwise a_h drops out from the candidate of leaders. Thus, unless all agents observe the same virtual IDs, at least one agent remains active as a candidate of leaders. From Lemma 4.4, it does not happen that all agents observe the same virtual IDs. Therefore, there exists at least one leader agent.

Next, we show that there exist at most $\lfloor k/g \rfloor$ leader agents. In each phase, if $a_h.id_2 < \min(a_h.id_1, a_h.id_3)$ or $a_h.id_2 = a_h.id_3 < a_h.id_1$ holds, a_h remains as a candidate of leaders. If the agent a_h satis es $a_h.id_2 < \min(a_h.id_1, a_h.id_3)$, then the a_h's backward and forward active agents drop out from the candidates of leaders. In the following, let us consider the case that agent a_h satis es $a_h.id_2 = a_h.id_3 < a_h.id_1$. Let $a_{h'}$ be a a_h's backward active agent and $a_{h''}$ be a a_h's forward active agent. Agent $a_{h'}$ observes three virtual IDs $a_{h'}.id_1, a_{h'}.id_2, a_{h'}.id_3$, and both $a_{h'}.id_2 = a_h.id_1$ and $a_{h'}.id_3 = a_h.id_2$ hold. Hence, $a_{h'}.id_2 > a_{h'}.id_3$ holds, and $a_{h'}$ drops out from the candidate of leaders. Next, $a_{h''}$ observes three virtual IDs $a_{h''}.id_1, a_{h''}.id_2, a_{h''}.id_3$, and both $a_{h''}.id_1 = a_h.id_2$ and $a_{h''}.id_2 = a_h.id_3$ hold. Since $a_{h''}.id_1 = a_{h''}.id_2$ holds, $a_{h''}$ does not satisfy the condition to remain as a candidate of leaders and drops out from the candidate. Thus, in each phase, at least half of active agents drop out from the candidates of leaders and become inactive. After executing i phases, there exist at most $k/2^i$ active agents. Therefore, after executing $\lceil \log g \rceil$ phases, there exist at most $\lfloor k/g \rfloor$ leader agents.

Finally, we show that there exist at least g 1 inactive agents between two leader agents in the virtual ring. At rst, we show that after executing j phases, there exist at least 2^j 1 inactive agents between two active agents. We show this by induction. For the case $j = 1$, there exists at least 2^1 $1 = 1$ inactive agent between two active agents as mentioned before. For the case $j = k$, we assume that there exist at least 2^k 1 inactive agents between two active agents. After executing $k + 1$ phases, since at least one of neighboring active agents becomes inactive, the number of inactive agents between two active agents is at least $(2^k$ $1) + 1 + (2^k$ $1) = 2^{k+1}$ 1. Hence, after executing j phases, there exist at least 2^j 1 inactive agents between two active agents. Therefore, after executing $\lceil \log g \rceil$ phases, there exist at least g 1 inactive agents between two leader agents in the virtual ring.

lemma 4.6 *Algorithm 13 requires $O(n \log g)$ total moves.*

Proof. In the virtual ring, each active agent moves until it observes three virtual IDs in each phase. This requires at most $O(n)$ total moves because each communication link of the virtual ring is passed at most three times and the length of the ring is $2(n$ 1). Since agents execute $\lceil \log g \rceil$ phases, we have the lemma.

4.3.2 The second part: leaders' instruction and agents' movement

In this section, we explain the second part, i.e., an algorithm to achieve the g-partial gathering by using leaders elected by the algorithm in Section 4.3.1. Let leader nodes (resp., inactive nodes) be the nodes where agents become leaders (resp., inactive agents). Note that all leader nodes and inactive nodes are token nodes. In this part, states of agents are divided into the following three types:

- *leader*: The agent instructs inactive agents where they should move.
- *inactive*: The agent waits for the leader's instruction.
- *moving*: The agent moves to its gathering node.

We explain the idea of the algorithm in the virtual ring. The basic movement is also similar to Section 3.1.2, that is, to divide agents into groups with at least g agents. In Section 3.1.2, each node has a whiteboard, while in this paper, each node is allowed to have an only token. In this section, agents achieve the g-partial gathering by using removable tokens. Concretely, each leader agent a_h moves to the next leader node, and while moving a_h repeats the following behavior: a_h removes tokens of inactive nodes g 1 times consecutively and then a_h does not remove a token of the next inactive node. After that, agents move to token nodes and meet at least g agents there.

First, we explain the behavior of leader agents. Whenever leader agent a_h visits an inactive node v_j, it counts the number of inactive nodes that a_h has visited. If the number plus one is not a multiple of g, a_h removes a token at v_j. Otherwise, a_h does not remove the token and continues to move. Agent a_h continues this behavior until a_h visits the next leader node $v_{j'}$. After that, a_h removes a token at $v_{j'}$. After completing this behavior, there exist at least g 1 inactive agents between two token nodes. Hence, agents solve the g-partial gathering problem by going to the nearest token node (This is done by changing their states to moving states). For example, let us consider the con guration like Fig. 15(a) ($g = 3$). We assume that a_1 and a_2 are leader agents and the other agents are inactive agents. In Fig. 15(b), a_1 visits the node v_2 and a_2 visits the node v_4 respectively. The numbers near nodes represent the number of inactive nodes that a_1 and a_2 observed respectively. Agents a_1 and a_2 remove tokens at v_1 and v_3, and do not remove tokens at v_2 and v_4 respectively. After that, a_1 and a_2 continue this behavior until they visit the next leader nodes. At the leader nodes, they remove the tokens (Fig. 15(c)).

When a token at v_j is removed, an inactive agent at v_j changes its state to a moving state and starts to move. Concretely, each moving agent moves to the nearest token node v_j. Note that, since each agent moves in a FIFO manner, it does not happen that a moving agent passes a leader agent and terminates at some token node before the leader agent removes the token. After all agents complete their own movements, the con guration changes from Fig. 15(c) to Fig. 15(d) and agents can solve the g-partial gathering problem. Note that, since each agent moves in the same virtual ring in a FIFO manner, it does not happen that an active agent executing the leader agent election passes a leader agent and that a leader agent passes an active agent.

Pseudocode. In the following, we show the pseudocode of the algorithm. The pseudocode of leader agents is described in Algorithm 15. Variable *tCount* is used to count the number of inactive nodes a_h visits. When a_h visits a token node v_j with another agent, v_j is an inactive node because an inactive agent becomes inactive at a token node and agents move in a FIFO manner. Whenever each leader agent a_h visits an inactive node, a_h increments the value of *tCount*. At inactive node v_j, a_h removes a token at v_j if $tCount \neq g$ 1 and continues to move otherwise. This means that, if a token is not removed at inactive node v_j, at least g agents meet at v_j. When a_h removes a token at v_j, an inactive agent at v_j changes its state to a moving state. When a_h visits a token node $v_{j'}$ with no agents, $v_{j'}$ is the next leader node. This is because agents at token nodes are in leader

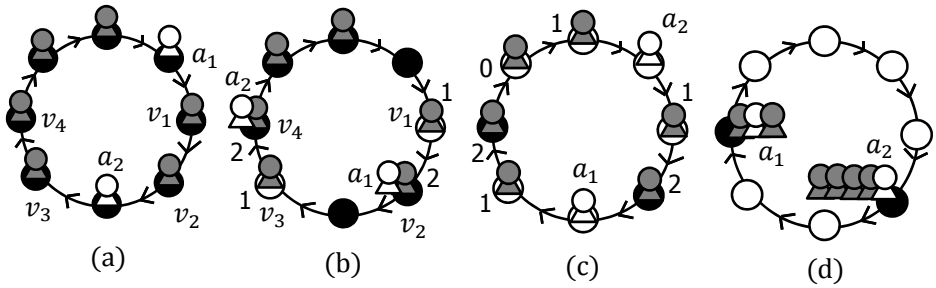


Figure 15: Partial gathering for removable-token model for the case $g = 3$ (a_1 and a_2 are leaders, and black nodes are token nodes)

Algorithm 15 The behavior of leader agent a_h (v_j is the current node of a_h)

Variable in Agent a_h
int $tCount = 0$;
Main Routine of Agent a_h
1: *NextToken*()
2: **while** there exists another agent at v_j **do**
3: //this is an inactive node
4: $tCount = (tCount + 1) \mod g$
5: **if** $tCount \neq g$ 1 **then**
6: remove a token at v_j
7: //an inactive agent at v_j changes its state to a moving state
8: **end if**
9: *NextToken*()
10: **end while**
11: remove a token at v_j
12: change its state to a moving state

or inactive states, and each inactive agent does not leave the token node until the token is removed. When leader agent a_h moves to the next leader node $v_{j'}$, a_h removes a token at $v_{j'}$ and changes its state to a moving state. In Algorithm 15, a_h uses the procedure *NextToken*(), by which a_h moves to the next token node. The pseudocode of *NextToken*() is described in Algorithm 16. In Algorithm 16, a_h performs the basic walk until a_h visits a token node v_j through the port (d_{v_j} 1).

The psedocode of inactive agents is described in Algorithm 17. Inactive agent a_h waits at v_j until either a token at v_j is removed or a_h observes another agent. If the token is removed, a_h changes its state to a moving state. If a_h observes another agent, the agent is a moving agent and terminates the algorithm at v_j. This means v_j is selected as a token node where at least g agents meet in the end of the algorithm. Hence, a_h terminates the algorithm at v_j.

The pseudocode of moving agents is described in Algorithm 18. In the virtual ring, each moving agent a_h moves to the nearest token node by using *NextToken*().

We have the following lemma about algorithms in Section 4.3.2.

lemma 4.7 *After the leader agent election, agents solve the g-partial gathering problem in $O(gn)$ total moves.*

Proof. We show the lemma in the virtual ring. At rst, we show the correctness of the proposed algorithms. Let $v_0^f, v_1^f, \ldots, v_l^f$ be inactive nodes that still have tokens after all leader agents complete their behaviors, and we call these nodes *nal nodes*. From Algorithm 15, each leader agent a_h removes the token at the inactive node g 1 times consecutively and does not remove the token at the next inactive node respectively. By this behavior and Lemma 4.5, there exist at least g 1 moving agents between v_i^f and v_{i+1}^f. Moreover, each moving agent moves to the nearest nal node. Therefore, agents solve the g-partial gathering problem.

In the following, we analyze the total moves required for the algorithms. At rst, let us consider the total moves required for each leader agent to move to the next leader node. This requires $2(n$ 1) total moves since all leader agents move around the virtual ring. Next, let us consider the total moves required for each moving (inactive) agent to move to the nearest token node (For example, the total moves form Fig. 15(c) to Fig. 15(d)). From Algorithm 18, each moving agent moves to the nearest nal node. We assume that some moving agent a_h goes to nal node v_i^f and terminates the algorithm. Then, a_h only moves between $v_{i\ 1}^f$ and v_i^f. In the

Algorithm 16 void *NextToken*() (v_j is the current node of a_h.)

Main Routine of Agent a_h

leave v_j through the port 0
let p be the port number through which a_h visits v_j
while (there dose not exist a token) $\vee$ ($p \neq d_{v_j}$ 1) **do**
  leave v_j through the port $(p+1)$ mod d_{v_j}
  let p be the port number through which a_h visits v_j
end while

Algorithm 17 The behavior of inactive agent a_h (v_j is the current node of a_h)

while (there dose not exist another agent at v_j)$\vee$(there exists a token at v_j) **do**
  wait at v_j
end while
if there exists another agent at v_j **then**
  terminate the algorithm
end if
if there does not exist a token **then**
  change its state to a moving state
end if

Algorithm 18 The behavior of moving agent a_h (v_j is the current node of a_h)

Main Routine of Agent a_h

NextToken()
terminate the algorithm

following, we show that the number of moving agents between some nal node v_i^f and its forward nal node v_{i+1}^f is at most $O(g)$. From Algorithm 15, the number of moving agents between two v_i^f and v_{i+1}^f is the sum of inactive nodes and leader nodes between v_i^f and v_{i+1}^f. Since there exists at least one nal node between two leader nodes, there exists at most one leader node between v_i^f and v_{i+1}^f. If there exist no leader node between v_i^f and v_{i+1}^f, then clearly there exist g 1 inactive nodes between v_i^f and v_{i+1}^f. If there exists one leader node v_l between v_i^f and v_{i+1}^f, there exist at most g 1 inactive nodes between v_i^f and v_l, and at most g 1 inactive nodes between v_l and v_{i+1}^f respectively. Thus, there exist at most $O(g)$ moving agents between some nal node v_i^f and v_{i+1}^f, and the total moves required for each moving (inactive) agent to move to the nearest nal node is at most $O(gn)$ since each communication link is passed by at most $O(g)$ times.

Therefore, we have the lemma.

From Lemma 4.6 and Lemma 4.7, we have the following theorem.

Theorem 4.5 *In the weak multiplicity detection and the removable-token model, our algorithm solves the g-partial gathering problem in $O(gn)$ total moves.*

5 Conclusion

In this paper, we proposed algorithms to solve the g-partial gathering problem in asynchronous unidirectional ring and asynchronous tree networks. If the network is a ring, we proposed three algorithms. First, we proposed a deterministic algorithm to solve the g-partial gathering problem for distinct agents in $O(gn)$ total moves. Second, we proposed a randomized algorithm to solve the g-partial gathering problem for anonymous agents in expected $O(gn)$ total moves. Third, we proposed a deterministic algorithm to solve the g-partial gathering problem for anonymous agents in $O(kn)$ total moves and we showed that there exist unsolvable initial con gurations in this model. In these three models, we assume that each node has a whiteboard. If the network is a tree, we considered three model variants. First, in the weak multiplicity and non-token model, we showed that there exist no algorithms to solve the g-partial gathering problem. Second, in the multi-visibility and non-token model, we showed that agents require $\Omega(kn)$ total moves to solve the g-partial gathering problem and proposed a deterministic algorithm to solve the g-partial gathering problem in $O(kn)$ total moves. Finally, in the single-visibility and removable-token model, we proposed a deterministic algorithm to solve the g-partial gathering problem in $O(gn)$ total moves.

References

[1] E. Kranakis, D. Krozanc, and E. Markou. *The mobile agent rendezvous problem in the ring*, volume 1. Morgan & Claypool Publishers, 2010.

[2] S. Dobrev, P. Flocchini, G. Prencipe, and N. Santoro. Mobile search for a black hole in an anonymous ring. *Algorithmica*, 48(1):67–90, 2007.

[3] T. Suzuki, T. Izumi, F. Ooshita, H. Kakugawa, and T. Masuzawa. Move-optimal gossiping among mobile agents. *Theoretical Computer Science*, 393(1):90–101, 2008.

[4] J. Chalopin, S. Das, and A. Kosowski. Constructing a map of an anonymous graph: Applications of universal sequences. *Proc. of OPODIS*, pages 119–134, 2010.

[5] L. Gasieniec, A. Pelc, T. Radzik, and X. Zhang. Tree exploration with logarithmic memory. *Proc. of SODA*, pages 585–594, 2007.

[6] S. Kawai, F. Ooshita, H. Kakugawa, and T. Masuzawa. Randomized rendezvous of mobile agents in anonymous unidirectional ring networks. *Proc. of SIROCCO*, pages 303–314, 2012.

[7] S. Elouasbi and A. Pelc. Time of anonymous rendezvous in trees: Determinism vs. randomization. *Proc. of SIROCCO*, pages 291–302, 2012.

[8] D. Baba, T. Izumi, H. Ooshita, H. Kakugawa, and T. Masuzawa. Linear time and space gathering of anonymous mobile agents in asynchronous trees. *Theoretical Computer Science*, pages 118–126, 2013.

[9] J. Czyzowicz, A. Kosowski, and A. Pelc. Time vs. space trade-offs for rendezvous in trees. *Proc. of SPAA*, pages 1–10, 2012.

[10] S. Dobrev, P. Flocchini, G. Prencipe, and N. Santoro. Multiple agents rendezvous in a ring in spite of a black hole. *Proc. of OPODIS*, pages 34–46, 2004.

[11] L. Barriere, P. Flocchini, P. Fraigniaud, and N. Santoro. Rendezvous and election of mobile agents: impact of sense of direction. *Theory of Computing Systems*, 40(2):143–162, 2007.

[12] G. D. Marco, L. Gargano, E. Kranakis, D. Krizanc, A. Pelc, and U. Vaccaro. Asynchronous deterministic rendezvous in graphs. *Proc. of MFCS*, pages 271–282, 2005.

[13] S. Guilbault and A. Pelc. Gathering asynchronous oblivious agents with restricted vision in an in nite line. *Proc. of SSS*, pages 296–310, 2013.

[14] A. Collins, J. Czyzowicz, L. Gasieniec, A. Kosowski, and R. Martin. Synchronous rendezvous for location-aware agents. *Proc. of DISC*, pages 447–459, 2011.

[15] E. Kranakis, D. Krizanc, and E. Markou. Mobile agent rendezvous in a synchronous torus. *Proc. the 2nd Interna- tional Conference on Trustworthy Global Computing*, pages 653–664, 2006.

[16] S. Guilbault and A. Pelc. Asynchronous rendezvous of anonymous agents in arbitrary graphs. *Proc. of OPODIS*, pages 421–434, 2011.

[17] Y. Dieudonne, A. Pelc, and D. Peleg. Gathering despite mischief. *Proc. of SODA*, pages 527–540, 2012.

[18] P. Flocchini, E. Kranakis, D. Krizanc, F. L. Luccio, N. Santoro, and C. Sawchuk. Mobile agents rendezvous when tokens fail. *Proc. of SIROCCO*, pages 161–172, 2004.

[19] L. Gasieniec, E. Kranakis, D. Krizanc, and X. Zhang. Optimal memory rendezvous of anonymous mobile agents in a unidirectional ring. *SOFSEM*, pages 282–292, 2006.

[20] E. Kranakis, N. Santoro, C. Sawchuk, and D. Krizanc. Mobile agent rendezvous in a ring. *Proc. of ICDCS*, pages 592–599, 2003.

[21] P. Flocchini, E. Kranakis, D. Krizanc, N. Santoro, and C. Sawchuk. Multiple mobile agent rendezvous in a ring. *Proc. of LATIN*, pages 599–608, 2004.

[22] E. Korach, D. Rotem, and N. Santoro. Distributed algorithms for nding centers and medians in networks. *TOPLAS*, 6(3):380–401, 1984.

[23] P Fraigniaud and A Pelc. Deterministic rendezvous in trees with little memory. *Proc. of DISC*, pages 242–256, 2008.

[24] G. L. Peterson. An $O(n \log n)$ unidirectional algorithm for the circular extrema problem. *TOPLAS*, 4(4):758–762, 1982.

Move-efficient algorithms for group gossiping of mobile agents

Jun Ri, Masahiro Shibata, Fukuhito Ooshita, Hirotsugu Kakugawa and Toshimitsu Masuzawa
Graduate School of Information Science and Technology
Osaka University
Osaka, Japan
Email: {jun-ri, m-sibata, f-oosita, kakugawa, masuzawa}@ist.osaka-u.ac.jp

***Abstract*—We introduce a concept of agent groups and formulate the group gossiping problem. An (agent) group is a set of agents that work together to attain a single objective. For example, agents created by a single application form one group. Since multiple applications are executed in a single mobile agent system, there exist multiple groups in such a system. In this case, it is useful to support cooperation among agents in each group.**

From this motivation, we formulate the group gossiping problem. The group gossiping problem requires each agent to collect information of all agents in its group. Since information to be collected is private and precious, no agents want to expose the information to other groups. For this reason, agents can exchange only control information (e.g., counter values or IDs) with other groups. In this setting, we aim to minimize the total number of moves to solve the group gossiping problem. As a result, we show the asymptotically tight upper and lower bounds for several network topologies.

***Keywords*-Mobile agent, Group gossiping, Move complexity, Distributed algorithm**

I. Introduction

A. Background and Motivation

As a design paradigm for distributed systems, *mobile agents* have received a lot of attention [3], [14], [15]. A (mobile) agent is an autonomous piece of software that moves in a network carrying its state information. There are many reasons for using agents [15]; For example, if remote hosts store a large amount of data for processing, agents can reduce the network traffic by moving to the remote hosts and processing the data locally. Besides adaptability and flexibility of agents simplify design of distributed systems.

In most mobile agent systems, multiple agents work cooperatively to improve system performance. For example, in a network management system, multiple agents traverse the network to collect load information of nodes and links, share the collected information, and inform each node about the information. To support the cooperation of mobile agents, many algorithms for fundamental operations such as gossiping and gathering have been developed [20], [25]. Gossiping requires all agents to share information of all agents, and gathering requires all agents to meet at a single node.

In this paper, we introduce a concept of *agent groups*. Intuitively, agents in a single agent group work to attain the same objective. For example, if an application creates multiple agents to attain some objective, the agents form a single agent group. Previous algorithms for gossiping and gathering assume all agents belong to a single group in the sense that all agents aim to achieve the same goal (i.e., share the information of all agents or meet all agents).

On the other hand, mobile agents from *multiple* agent groups often share a single mobile agent system. For example, if multiple applications share a single mobile agent system, agents created by each application make up one agent group. Another example is a mobile agent system used by multiple organizations. In such a system, agents created by each organization make up one agent group. For both cases, agents are not interested in agents that belong to other agent groups. Consequently agents want to share the information or to gather among agents in their agent groups. This motivates us to consider new variants of operations, gossiping or gathering in each agent group.

B. Our contributions

From the above motivation, we formulate the group gossiping problem. In this problem, agents are divided into groups, and the goal is to make each agent collect information of all agents in its group. We aim to minimize the total number of agent moves (over all groups) to solve the group gossiping problem. This is because, since all agents share a single mobile agent system, it is important to optimize the performance of the whole system.

Since algorithms for gossiping among agents in a single agent group are proposed [25], there are two trivial solutions to solve the group gossiping problem. One is to execute the algorithm in [25] by assuming that all agents belong to a single group. However, to execute this algorithm, agents must expose its information to agents in other groups. Such a behavior is usually not allowed due to security reasons since the information may be private and precious. Another solution is to execute the algorithm in [25] independently for each group. This solution avoids the above security problem. However, since agents do not cooperate with agents in other groups, this requires a large total number of moves. For this reason, we aim to reduce the total number of moves by

Table I
THE TOTAL NUMBER OF MOVES TO SOLVE THE GROUP GOSSIPING PROBLEM

Graph	System model	Sense of direction	A trivial solution	Our results
Ring	synchronous	without	$O(gN)$	$\Theta(gN)$
	asynchronous	without	$O(gN \log k')$	$\Theta(N \log k' + gN)$
Tree	asynchronous	without	$O(gN)$	$\Theta(gN)$
Complete	asynchronous	without	$O(gN \log k')$	$\Theta(N \log k)$
		with	$O(gN)$	$\Theta(N)$
Arbitrary	asynchronous	without	$O(gN \log k' + gM)$	$\Theta(N \log k' + M + gN)$

N, M and k are the numbers of nodes, links and agents respectively.
g is the numbers of groups, and k' is the maximum number of agents in a group.

exchanging some control data (e.g., counter values or IDs) among agents in different groups.

Table I summarizes the contribution of this paper for the group gossiping problem. The column of a trivial solution shows the total number of moves required to execute the algorithm in [25] independently for each group. For synchronous rings and asynchronous trees, we show that the trivial algorithm is asymptotically optimal in terms of the total number of moves. For asynchronous ring networks, asynchronous complete networks, and asynchronous arbitrary networks, we propose algorithms that can reduce the total number of moves compared to the trivial solution. This means cooperation of agents in different groups reduces the total number of moves to solve the group gossiping problem. We also show that the proposed algorithms are asymptotically optimal in terms of the total number of moves.

C. Related Works

To improve the performance of mobile agent systems, many fundamental problems for cooperation of agents have been studied. The most investigated problem is the gathering problem [14], [20], which requires all agents to meet at a single node[1]. The gathering problem has been considered in several settings [2], [4], [6], [7], [8], [10], [18]. Dessmark et al. [6] give algorithms for synchronous agents in trees, rings, and arbitrary graphs. Dieudonné and Pelc [7] propose deterministic algorithms for multiple synchronous agents in arbitrary networks. Dieudonné et al. [8] show that two asynchronous agents can meet in arbitrary networks at polynomial number of moves.

The gossiping problem is considered in [17], [25]. Note that algorithms for the gathering problem are also solutions for the gossiping problem because all agents can exchange their information by meeting at a single node. However, the gossiping problem can be solved without gathering of all agents. Suzuki et al. [25] show that the gossiping problem requires a smaller total number of moves than the gathering problem. In [17], self-stabilizing algorithms for the gossiping problem are studied.

[1]The gathering problem for two agents is sometimes called the rendezvous problem.

Another fundamental problem is the exploration problem, which requires agents to visit every node. Exploration algorithms for a single agent [19], [22] and for multiple agents [5], [9] have been proposed. The important notion for the exploration problem is a universal exploration sequence (UXS) [13]. By moving based on a UXS, agents can visit every node. Reingold [22] gives a log-space constructible UXS and shows that an agent solves the exploration problem with $O(\log n)$ bits of memory.

While all the above researches do not consider agent groups, Shibata et al. [23], [24] consider some groups and introduce the partial gathering problem. The partial gathering problem requires, for given input g, that each agent makes a group composed of at least g agents and meet them at a single node. The concept looks similar to groups in this paper, however that is different because groups are initially given in the group gossiping problem.

II. PRELIMINARIES

In this section, we define the system model and the group gossiping problem. Note that we use the same system model as [25] except for a concept of groups.

A. Mobile Agent Systems

A network is modeled as an undirected graph $G = (V, E)$, where V and E are respectively the node set and the link set in G. The numbers of nodes and links are denoted by N (i.e., $N = |V|$) and M (i.e., $M = |E|$), respectively. A link in E connects two distinct nodes in V. The link between nodes u and v is denoted by e_{uv} or e_{vu}. On each node, each incident link is locally labeled by port numbers. Let $\lambda_u(e)$ be the port number of link e on a node u ($\lambda_u(e) \in \{1, 2, \ldots, deg_u\}$, where deg_u is the number of u's incident links). There exist k agents in the network. An agent a_i ($0 \leq i < k$) is an autonomous state machine that can migrate from one node to another in the network. Agents on a node $u \in V$ can migrate to a node $v \in V$ only when link e_{uv} is contained in E. We assume that each agent has a distinct identifier (ID) [2]. Let id_i be the ID of agent a_i (use id_i and a_i interchangeably in this paper). Each agent does not initially know IDs of

[2]Node IDs are not required in this paper, however this is not essential. This is because distinct agents can assign distinct IDs to nodes when they visit the nodes.

other agents. Agents are divided into g *groups*. Each agent a_i knows its group ID gid_i, and agents with the same group ID form a single group. For each agent a_i, we define $\mathcal{G}(a_i)$ as the set of agents that belong to the same group as a_i (i.e., have the same group ID as a_i). Note that $a_i \in \mathcal{G}(a_i)$ holds for any i. We assume each group consists of at least two agents. Let k' be the maximum number of agents that belong to a single group (i.e., $k' = \max_{0 \leq i < k}\{|\mathcal{G}(a_i)|\}$). We assume that each agent has prior knowledge of neither G, N, M, k, g nor k'. Each agent is initially located on any node in G, and more than one agent is not located on the same node. A node on which an agent is initially located is called the *home node* of the agent. When a node v is the home node of agent a_i, we also say a_i is the *home agent* of v.

Each node v is provided with a *whiteboard*, i.e., local storage where agents on v can write and read data. When multiple agents on a node execute their operations, the operations are sequentially executed in an arbitrary order with the following assumption: at the beginning of the algorithm, each agent executes operations at its home node before other agents access a whiteboard on the node.

An agent a_i on each node v performs a sequence of the following operations;

- *read*(v) : agent a_i reads data written on node v's whiteboard, and executes local computation.
- *write*(v, d) : agent a_i writes data d on node v's whiteboard.
- *move*($v, \lambda_v(e)$) : agent a_i moves to one of v's neighbor nodes through the link labeled by port number $\lambda_v(e)$. If $\lambda_v(e)$ is zero, a_i stays on v.

We assume that these three operations are executed atomically. Agents are said to be *asynchronous* if migration time and local processing time of agents are unpredictable but finite. In contrast, agents are *synchronous* if all agents execute every rounds synchronously; in each round, every agent arrives at a node, accesses the whiteboard and executes local computation on the node, and stays on the node or starts migration to one of the neighboring nodes. The agents arrive at the destination nodes at the beginning of the next round.

A state of an agent is represented by a set of variables the agent has and a set of information the agent has collected. A state of a node is represented by the state of its whiteboard. A system configuration C is represented by the states of all nodes, the states of all agents, and the locations of all agents. A system configuration is changed by events of agents (e.g., read and write on a whiteboard, and migration). Let C_0 be an initial configuration of a system and Ev_i be a set of events that occur simultaneously at the configuration C_i. An execution of a mobile agent system is an alternate sequence of configurations and sets of events $EX = C_0, Ev_0, C_1, Ev_1, C_2, \cdots$, such that occurrence of events Ev_{i-1} changes the configuration from C_{i-1} to C_i. We say that an agent a_j terminates in a configuration C_i iff a_j never executes any operation after C_i.

B. Group Gossiping Problem

In this paper, we define the *group gossiping problem*. In an initial configuration, each agent a_j has only its own information I_j. The goal of this problem is that every agent collects information of all agents in its group. Hereinafter information means information each agent has to collect. The group gossiping problem is defined as follows.

Let $S_j(C_i)$ be a set of the information an agent a_j has in configuration C_i. In initial configuration C_0, each agent a_j has only its own information I_j;

$$S_j(C_0) = \{I_j\}.$$

The group gossiping problem is solved in configuration C_i iff all k agents terminate and the following condition is satisfied;

$$\forall j (0 \leq j < k) : \ S_j(C_i) = \bigcup_{a_\ell \in \mathcal{G}(a_j)} \{I_\ell\}$$

Agents can write only the control data on a whiteboard, e.g., some number of identifiers and counter values. We disallow each agent a_j to leave any information I_j on a whiteboard due to security reason. It is insecure to write precious information on a whiteboard because every agent in different groups can access the whiteboard. Instead, we allow agents to exchange the set of information if agents in the same group stay on the same node. When a set P of agents is located on the same node in configuration C_i, then the subsequent configuration C_{i+1} satisfies;

$$\forall a_j \in P : \ S_j(C_{i+1}) = \bigcup_{a_\ell \in P \cap \mathcal{G}(a_j)} S_\ell(C_i).$$

We define a move as a migration of an agent from one node to its neighbor node. The complexity of the group gossiping problem is measured by the total number of moves until all agents terminate in the worst case.

III. A LEADER-ELECTION-BASED ALGORITHM

In this section, we show a leader-election-based algorithm to solve the group gossiping problem for several networks. To elect a leader among agents, we use an algorithm for the node leader election problem in message-passing systems. For this reason, we first give the definition of the node leader election problem.

Definition 1 In message-passing systems, an algorithm is said to solve the node leader election problem for network G iff it satisfies the following conditions: 1) exactly one of nodes is elected as a leader and all nodes in G know the ID of the leader node, and 2) once a node decides to be a leader, the node never changes its decision.

If we have a node leader election algorithm for asynchronous message-passing systems, we can use it to elect a leader agent in mobile agent systems from the following theorem in [25].

Theorem 1 [25] Suppose the total number of exchanged messages to execute a node leader election algorithm with k initiators in asynchronous message-passing systems is at most m_{mp}. Then, the (agent) leader election with k agents is solved with at most $2 \cdot m_{mp}$ moves.

In the rest of this section, we explain an algorithm to solve the group gossiping problem by using an algorithm for the agent leader election. First, we explain the idea of algorithms for the gossiping problem in [25] to extend it to the group gossiping problem. Remind that the gossiping problem requires agents to collect information of all agents. In the algorithm, agents first execute a leader election algorithm and elect a single leader. Then, the leader completes gossiping by traversing the network twice: The leader collects information of all agents in the first traversal, and distribute the information to every agent in the second traversal. Since the leader election usually requires a larger number of moves, the leader election dominates the total number of moves throughout the algorithm.

For the group gossiping problem, we can use the same idea. That is, each group elects a single group leader, and the group leader completes gossiping in the group by traversing the network twice. However, if each agent group executes a leader election independently, this requires a lot of moves. To avoid this, we elect a single leader among all agents at the beginning of the algorithm. Then, the leader elects a group leader for each group by traversing the network once. Concretely, when the leader meets an agent of group gid, it nominates the agent as a group leader if it has not yet met another agent of group gid.

From the above algorithm, we have the following lemma.

Lemma 1 Suppose that the leader election with k agents is solved with at most m_l moves and the number of moves required for an agent to traverse the whole network is at most m_t. Then, the group gossiping problem with k agents and g groups is solved with $m_l + (2g + 1)m_t$ moves.

Proof: Agents elect a single leader and then the leader elects a group leader for each group by traversing the network once. This requires $m_l + m_t$ moves. After that, each group leader completes gossiping in its group by traversing the network twice. This requires $2gm_t$ moves. ■

IV. Upper and lower bounds

In this section, we present the upper and lower bounds on the total number of moves for the group gossiping problem in several networks.

A. Synchronous ring networks

In this subsection, we consider synchronous ring networks without sense of direction. For ring networks, the node leader election problem in synchronous message-passing systems can be solved with $O(N)$ messages [11]. Since this algorithm works in only synchronous systems, Theorem 1 cannot be applied. However, fortunately this result can be easily applied to agent systems, and it is proved that gossiping among k agents can be solved with $O(N)$ moves in any synchronous ring network [25]. By executing this algorithm independently for each group, we have the following theorem.

Theorem 2 The group gossiping problem is solved with $O(gN)$ moves in synchronous ring networks.

For the lower bound, we have the following theorem.

Theorem 3 Any algorithm for the group gossiping problem requires $\Omega(gN)$ moves in synchronous ring networks.

Proof: For any given N, k, g and k', we show that there exists an initial configuration that requires $\Omega(gN)$ moves to solve the group gossiping problem. Assume that nodes $v_0, v_1, \ldots, v_{N-1}$ are located in this order. Let $\mathcal{G}_i$ $(0 \leq i < g)$ be a set of agents in group i. Since $|\mathcal{G}_i| \geq 2$ holds, we can deploy agents as follows: First deploy two agents in $\mathcal{G}_i$ on v_i and $v_{i+N/2}$ for each i $(0 \leq i < g)$, and then deploy remaining agents on remaining nodes arbitrarily. Since the two agents in each group must exchange their information, this requires at least $N/2$ moves for each group. Therefore, the total number of moves is at least $gN/2$. ■

B. Asynchronous ring networks

In this subsection, we consider asynchronous ring networks without sense of direction. For ring networks, the node leader election problem in asynchronous message-passing systems can be solved with $O(N \log N)$ messages [21]. This is the message complexity for an arbitrary number of initiator nodes in the worst case. It can easily be shown that the message complexity for k initiator nodes is $O(N \log k)$ by using an algorithm similar to [21]. Therefore, from Theorem 1, leader election among k agents can be solved with $O(N \log k)$ moves.

An agent can travel the whole network with N moves. Thus, from Lemma 1 and $k \leq gk'$, the following theorem is obtained.

Theorem 4 The group gossiping problem is solved with $O(N \log k' + gN)$ moves in asynchronous ring networks.

In the following, we show the lower bound. Similarly to Theorem 3, we have the following lemma.

Lemma 2 Any algorithm for the group gossiping problem requires $\Omega(gN)$ moves in asynchronous ring networks.

In addition, we prove the following lemma from the lower bound of the node leader election problem in message-passing systems.

Lemma 3 Any algorithm for the group gossiping problem requires $\Omega(N \log k')$ moves in asynchronous ring networks.

Proof: We show the lemma holds even if the number of group g is given to each node. For contradiction, assume that the group gossiping problem is solved with $o(N \log k')$ moves in any asynchronous ring network. Then, we show that the node leader election problem with k' initiator nodes in message-passing systems is solved with $o(N \log k')$ messages.

Consider a N-node ring network $G = (V, E)$ in message-passing systems. Assume that nodes $v_0, v_1, \ldots, v_{N-1}$ are located in this order. Let V_I be a set of k' initiator nodes. To solve the node leader election in G with the initiator node set V_I, we let nodes in G simulate (agent) leader election in a virtual ring network $G' = (V', E')$ constructed as follows: For each $v_i \in V_I$, replace v_i in G by a path $P_i = (v_i^0, v_i^1, \ldots, v_i^{g-1})$, and add links $\{v_{i-1}, v_i^0\}$ and $\{v_i^{g-1}, v_{i+1}\}$ to G. Then, the length of ring G' is $N' = N + (g-1)k' \leq 2N$. To solve the node leader election in G, each initiator node v_i simulates all of nodes in P_i of the virtual ring network G'.

Each node can easily simulate the agent algorithm by sending a state of an agent in a message. Then, the number of total messages is equal to the number of total moves of agents. First, each initiator node v_i puts g agents $a_i^0, a_i^1, \ldots, a_i^{g-1}$ on every node in path P_i. At that time, each agent a_i^j has an ID (id_i, j) and a group ID j, where id_i is the ID of node v_i. In the whole virtual ring network, there exist $k = gk'$ agents and each group consists of k' agents. Each agent uses its ID as information to be collected. In this setting, nodes simulate an algorithm to solve the group gossiping problem. This is done with $o(N' \log k') = o(N \log k')$ messages. After completing the group gossiping, each initiator node knows IDs of all initiator nodes. Thus, the node with the minimum ID can become a leader. After that, the leader broadcasts its ID with $O(N)$ messages and then the node leader election problem is solved. Throughout the algorithm, nodes exchange $o(N \log k')$ messages to solve the node leader election problem.

However, it is proved in [**?**] that the lower bound on messages for the node leader election problem with k' initiator nodes is $\Omega(N \log k')$. This is a contradiction. ■

From Lemmas 2 and 3, we have the following theorem.

Theorem 5 Any algorithm for the group gossiping problem requires $\Omega(N \log k' + gN)$ moves in asynchronous ring networks.

C. Asynchronous non-rooted tree networks

In this subsection, we consider asynchronous non-rooted tree networks without sense of direction. For asynchronous tree networks, it is proved in [25] that gossiping among k agents can be solved with $O(N)$ moves. By executing this algorithm independently for each group, we have the following theorem.

Theorem 6 The group gossiping problem is solved with $O(gN)$ moves in asynchronous tree networks.

For the lower bound, we have the following theorem.

Theorem 7 Any algorithm for the group gossiping problem requires $\Omega(gN)$ moves in asynchronous tree networks.

Proof: We consider a N-node line network as a tree network. Then, we can prove the theorem similarly to Theorem 3. ■

D. Asynchronous complete networks without sense of direction

In this subsection, we consider asynchronous complete networks without sense of direction. It is proved in [1] that the message complexity of an algorithm for the node leader election problem in message-passing systems is $O(N \log N)$. This is the message complexity for an arbitrary number of initiator nodes in the worst case. It can easily be shown that the message complexity for k initiator nodes is $O(N \log k)$ by using an algorithm similar to [1]. Thus, from Theorem 1, k agents can elect a leader in $O(N \log k)$ moves. Since an agent can travel the whole network in $O(N)$ moves, agents can solve the group gossiping problem in $O(N \log k + gN)$ moves from Lemma 1.

In the following, we reduce the total number of moves from $O(N \log k + gN)$ to $O(N \log k)$. This is done by reducing the total number of moves after one leader is elected. First, we explain the overview of the whole algorithm. In the first phase, agents execute leader election and then move back to their home nodes. In the second phase, an elected leader a_ℓ traverses the whole network, and recognizes, for each group, the links incident at a_ℓ's home node that connect to home nodes of the agents in the group. Agent a_ℓ writes this information on the whiteboard of its home node to guide group leaders later. In the third phase, a_ℓ traverses the whole network to elect one group leader from each group. In the fourth phase, each group leader visits every agent in its group twice. To do this, each group leader first moves to a_ℓ's home node and then visits every agent in its group by using the information written in the second phase. Note that, since the sets of nodes visited by group leaders are disjoint except for a_ℓ's home node, the fourth phase requires $O(N)$ total moves.

We give the details of the algorithm. First, we explain some important variables on an agent or a whiteboard.

For each agent a_i, we denote the ID and the group ID by $a_i.id$ and $a_i.gid$, respectively. Agent a_i has a variable $a_i.role$ which indicates the role of a_i. The role of a_i can be a candidate ($a_i.role = candidate$), a leader ($a_i.role = leader$), a non-leader ($a_i.role = non\_leader$), or a group leader ($a_i.role = group\_leader$). At the beginning of the algorithm, every agent is a candidate. After the first phase, one agent becomes a leader and others become non-leaders. After the third phase, one agent in each group becomes a group leader. Each node v has the following variables in its whiteboard: $v.id$, $v.gid$, $v.port\_label[0, 1, \ldots, N-2]$, and $v.p_{leader}$. If v is the home node of some agent a_i, a_i first assigns its ID and group ID to $v.id$ and $v.gid$ respectively. Variables $v.port\_label[0, 1, \ldots, N-2]$ and $v.p_{leader}$ store the information to guide group leaders. We assume every variable on the whiteboard has $\perp$ as its initial value.

In the first phase, each agent assigns its ID and group ID to $v.id$ and $v.gid$, and then executes a leader election algorithm described in the first paragraph of this subsection. After completing the leader election, each agent goes back to its home node. This behavior is easily realized by memorizing its trajectory (i.e., a sequence of port numbers) and moving based on the reverse of the trajectory. At the end of the first phase, one agent a_ℓ is elected as a leader (i.e., $a_\ell.role = leader$) and others are non-leaders (i.e., $a_i.role = non\_leader$ for $i \neq \ell$).

In the second and third phases, only a leader agent a_ℓ moves. The pseudocode of the second and third phases is given in Algorithm 1. In the second phase, for each port p, a_ℓ leaves its home node v_ℓ through port p, visits a node v, records v's group ID (if v is a home node of an agent), returns to v_ℓ, and then writes v's group ID to $v_\ell.port\_label[p]$. During the second phase, after a_ℓ moves from v_ℓ to some node v, a_ℓ writes on $v.p_{leader}$ the port number connecting to v_ℓ. In the third phase, for each group except for a_ℓ's group, a_ℓ moves to a home node of one agent and nominates it as a group leader. At the end of the third phase, a_ℓ also becomes a group leader. In the pseudocode, a_ℓ uses some additional variables on agents. In line 17, a leader nominates an agent as a group leader by exchanging some messages with the agent. The message exchange is easily realized by using the whiteboard.

In the fourth phase, each group leader a_g visits home nodes of agents in its group twice. The pseudocode of the fourth phases is given in Algorithm 2. On the first visit a_g collects information of each agent, and on the second visit a_g delivers collected information. Each group leader visits only home nodes of agents in its group by moving based on $v_\ell.port\_label[0 \ldots N-2]$. Each group leader terminates after it completes delivery of information. Each non-leader agent terminates after it receives information of all agents in the same group.

From the above algorithms, the following theorem is obtained.

Algorithm 1 The second and third phases: the behavior of leader a_ℓ.

```
1:  // The second phase
2:  for p := 0 to N − 2 do
3:     leave through port p and visit v through port q
4:     v.p_leader := q
5:     v_gid := v.gid // v.gid =⊥ if v has no home agent
6:     leave through port q and go back to v_ℓ
7:     v_ℓ.port_label[p] := v_gid
8:  end for
9:
10: // The third phase
11: selected := {⊥, a_ℓ.gid}
12: for p := 0 to N − 2 do
13:    if v_ℓ.port_label[p] ∈ selected then
14:       continue
15:    else
16:       leave through port p and visit v
17:       wait for a home agent of v and nominate it as a
          group leader
18:       selected := selected ∪ {v.gid}
19:       leave through port v.p_leader and go back to v_ℓ
20:    end if
21:    become a group leader
22: end for
```

Algorithm 2 The fourth phase: the behavior of group leader a_g.

```
1:  // v_g is the current node of a_g.
2:  if v_g.p_leader ≠⊥ then
3:     // v_g = v_ℓ if v_g.p_leader =⊥ holds.
4:     leave through port v_g.p_leader and visit v_ℓ
5:  end if
6:  for p := 0 to N − 2 do
7:     if v_ℓ.port_label[p] = a_g.gid then
8:        leave through port p and visit v
9:        wait for the home agent of v and collect informa-
          tion
10:       leave through port v.p_leader and visit v_ℓ
11:    end if
12: end for
13: for p := 0 to N − 2 do
14:    if v_ℓ.port_label[p] = a_g.gid then
15:       leave through port p and visit v
16:       deliver information to the home agent of v
17:       leave through port v.p_leader and visit v_ℓ
18:    end if
19: end for
```

Theorem 8 The group gossiping problem is solved with $O(N \log k)$ moves in asynchronous complete networks without sense of direction.

Proof: Clearly the above algorithm solves the group gossiping problem. In the following, we consider the total number of moves. In the first phase, agents elect a leader with $O(N \log k)$ moves from the algorithm proposed in [1] and Theorem 1. They return to their home nodes with the same number of moves. Consequently, the first phase requires $O(N \log k)$ moves. In the second and third phases, a leader visits all nodes twice with $O(N)$ moves. In the fourth phase, each group leader visits home nodes of agents in its group twice. Since the sets of nodes visited by group leaders are disjoint except for v_ℓ, the fourth phase requires $O(N)$ total moves. Therefore, the toal number of moves is $O(N \log k)$ throughout the algorithm. ■

In the following, we show the lower bound. Similarly to [1], it can be proved that the message complexity of the node leader election problem with k initiator nodes is $\Omega(N \log k)$. By the proof similar to Lemma 3, we can show the lower bound is $\Omega(N \log k')$. However, since this is not tight, we prove the tight lower bound by another approach.

Theorem 9 Any algorithm for the group gossiping problem requires $\Omega(N \log k)$ moves in asynchronous complete networks without sense of direction.

Proof: For simplicity, we assume $k = 2^p$ and $N = qk$ hold for some positive integers p, q. Without loss of generality, we assume a_0 and a_{k-1} belong to the same group. To prove the lower bound on the total number of moves, we consider agents that have a communication capability stronger than that assumed in Section II. To be concrete, we assume that, when agent a_x visits some node v after agent a_y visits v, agents a_x and a_y can continue to obtain both of a_x's state and a_y's state after that. We say, in this case, a_x and a_y share the states, and denote the relation by $a_x \simeq a_y$. Clearly, the relation $\simeq$ satisfies reflexive law and symmetrical law. In addition, we assume $\simeq$ satisfies transitive law, that is, $\forall a_x, a_y, a_z : a_x \simeq a_y \wedge a_y \simeq a_z \Rightarrow a_x \simeq a_z$ holds. Then, $\simeq$ satisfies equivalence law and consequently a set of agents can be divided into equivalence classes. We define $[a_x]$ as the equivalence class of a_x under $\simeq$, that is, $[a_x] = \{a_y | a_y \simeq a_x\}$. We define the territory $T(a_x)$ of a_x as the set of nodes visited by an agent in $[a_x]$. From the definition, when agent a_y visits a node in $T(a_x)$, equivalence classes $[a_x]$ and $[a_y]$ are merged. To solve the group gossiping problem, all agents in the same group must belong to the same equivalence class.

To prove the lower bound, we consider an (adversary) scheduler. The scheduler decides the timing of agents' movements. In addition, we assume the scheduler decides the port number of each link during execution of an algorithm. That is, the scheduler assigns a port number to a link on a node when the port number on the node is first used. When an agent visits node v through link e, the port number of e on v is also assigned if it is first used.

Fix an algorithm A that solves the group gossiping problem. We construct the behavior of the scheduler that makes agents move $\Omega(N \log k)$ times to solve the group gossiping by A.

The behavior of the scheduler is divided into multiple rounds. The scheduler activates agents independently but synchronizes them at the end of each round. First, we consider the 0-th round. During the 0-th round, the scheduler makes every agent visit N/k nodes on the condition that no two agents visit the same node. That is, the scheduler assigns a port number to each link so that no two agents visit the same node. Then, for every agent a_x, $[a_x] = \{a_x\}$ and $|T(a_x)| = N/k$ hold. At the end of the 0-th round, the scheduler makes a_{2m} and a_{2m+1} ($m = 0, 1, \ldots, k/2 - 1$) visit a node in $T(a_{2m+1})$ and $T(a_{2m})$, respectively. Since equivalence classes $[a_{2m}]$ and $[a_{2m+1}]$ are merged, $[a_{2m}] = \{a_{2m}, a_{2m+1}\}$ and $|T(a_{2m})| = 2N/k$ hold. The total number of moves during the 0-th round is N because every agent moves N/k times.

After that, the first round starts. Let $[a_x]_1$ and $T_1(a_x)$ be the equivalence class and territory of a_x at the beginning of the first round, respectively. Then, $[a_{2m}]_1 = \{a_{2m}, a_{2m+1}\}$ and $|T_1(a_{2m})| = 2N/k$ hold for $m = 0, 1, \ldots, k/2 - 1$. During the first round, the scheduler makes agents in $[a_{2m}]_1$ move in their territory $T_1(a_{2m})$ as many times as possible. That is, the scheduler, if possible, chooses an agent $a_x \in [a_{2m}]_1$ and assigns a port number so that a_x moves to a node in $T_1(a_{2m})$. If such a behavior is impossible, the scheduler makes agents move so that an agent in $[a_{4m}]_1$ and $[a_{4m+2}]_1$ ($m = 0, 1, \ldots, k/4 - 1$) moves to a node in $T_1(a_{4m+2})$ and $T_1(a_{4m})$, respectively. Then, the first round ends. At that time, $[a_{4m}] = \{a_{4m}, a_{4m+1}, a_{4m+2}, a_{4m+3}\}$ and $|T(a_{4m})| = 4N/k$ hold for $m = 0, 1, \ldots, k/4 - 1$. We consider the total number of moves during the first round. Assume that, at the end of the first round, agent $a_x \in [a_{4m}]_1$ (resp., $a_x \in [a_{4m+2}]_1$) moves from node $u \in T_1(a_{4m})$ (resp., $u \in T_1(a_{4m+2})$) to node $v \in T_1(a_{4m+2})$ (resp., $v \in T_1(a_{4m})$). This behavior happens because every unused link around u connects to a node not in $T_1(a_x)$. In other words, every link connecting to a node in $T_1(a_x)$ is used. From $|T_1(a_x)| = 2N/k$, $2N/k - 1$ links around u is used. Note that, before the beginning of the first phase, the number of used links around u is at most N/k because $|T(a_y) = N/k|$ holds for each a_y at the end of the 0-th round. This implies at least $N/k - 1$ links around u are used during the first round. That is, during the first round, agents in $[a_x]_1$ move at least N/k times in total including the movement from u to v. Since there exist $k/2$ equivalence classes, the total number of moves during the first round is at least $N/2$.

We can repeat the same discussion for the subsequent rounds. Consider the i-th round. Let $[a_x]_i$ and $T_i(a_x)$ be the equivalence class and territory of a_x at the beginning of the i-th round, respectively. Then, $[a_{2^i m}]_i =$

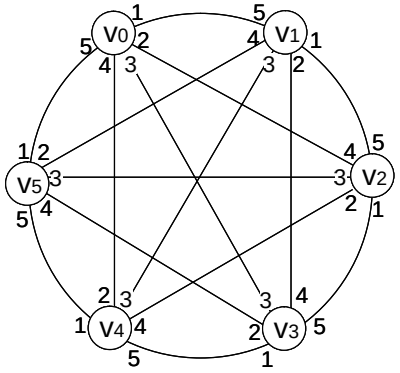


Figure 1. A complete network with sense of direction of six nodes.

$\{a_{2^i m}, a_{2^i m+1}, \ldots, a_{2^i(m+1)-1}\}$ and $|T_i(a_{2^i m})| = 2^i N/k$ hold for $m = 0, 1, \ldots, k/2^i - 1$. Similarly to the first round, the scheduler makes agents in $[a_{2^i m}]_i$ move in their territory $T_i(a_{2^i m})$ as many times as possible. At the end of the round, the scheduler makes agents move so that an agent in $[a_{2^{i+1} m}]_i$ and $[a_{2^{i+1} m+2^i}]_i$ ($m = 0, 1, \ldots, k/2^i - 1$) moves to a node in $T_i(a_{2^{i+1} m+2^i})$ and $T_i(a_{2^{i+1} m})$, respectively. Then, the i-th round ends. We consider the total number of moves during the i-th round. Assume that, at the end of the i-th round, agent $a_x \in [a_{2^{i+1} m}]_i$ (resp., $a_x \in [a_{2^{i+1} m+2^i}]_i$) moves from node $u \in T_i(a_{2^{i+1} m})$ (resp., $u \in T_1(a_{2^{i+1} m+2^i})$) to node $v \in T_i(a_{2^{i+1} m+2^i})$ (resp., $v \in T_i(a_{2^{i+1} m})$). Since $2^i N/k - 1$ links around u are used at that time and at most $2^{i-1} N/k$ links are used before the i-th round, $2^{i-1} N/k - 1$ links are used during the i-th round. Consequently, agents in each equivalence class move at least $2^{i-1} N/k$ times in total during the i-th round. Since there exist $k/2^i$ equivalence classes, the total number of moves during the i-th round is at least $N/2$.

Lastly, we consider how many rounds are required to solve the group gossiping problem. Since a_0 and a_{k-1} belong to the same group, a_0 and a_{k-1} must belong to the same equivalence class. However, this occurs at the end of the $(\log k - 1)$-th round. This implies $\Omega(\log k)$ rounds are required. Since agents move $\Omega(N)$ times in total during each round, agents move $\Omega(N \log k)$ times in total to complete the group gossiping problem. ■

E. Asynchronous complete networks with sense of direction

In this subsection, we consider asynchronous complete networks with sense of direction. The sense of direction is given at each node as follows; nodes are denoted by $v_0, v_1, \ldots, v_{N-1}$, numbered clockwise in a ring, and for every i, j ($0 \le i, j \le N-1, i \ne j$), the port number of link $e_{v_i v_j}$ on v_i is $(j - i) \bmod N$. Figure 1 shows a complete network with sense of direction of six nodes.

For this network, we can apply the algorithm proposed in the previous subsection. In addition, it is proved in [16] that the message complexity of an algorithm for the node leader election in message-passing system is $O(N)$. This implies the first phase of the algorithm requires only $O(N)$ total moves. Therefore we have the following theorem.

Theorem 10 The group gossiping problem is solved with $O(N)$ moves in asynchronous complete networks with sense of direction.

The following lower bound clearly holds since every node should be visited by at least one agent.

Theorem 11 Any algorithm for the group gossiping problem requires $\Omega(N)$ moves in asynchronous complete networks with sense of direction.

F. Asynchronous arbitrary networks

In this subsection, we consider asynchronous arbitrary networks without sense of direction. For a network G, a leader election in asynchronous message-passing systems can be solved by Gallager's algorithm for constructing a minimum spanning tree (MST) in G [12]. It is proved in [12] that the message complexity of the algorithm is $O(N \log N + M)$, where M is the number of links. In [25], it is proved that, in a network with k initiator nodes, the message complexity for constructing a MST is $O(N \log k + M)$ by using Gallager's algorithm. Therefore, from Theorem 1, leader election among k agents can be solved with $O(N \log k + M)$ moves.

An agent can travel the whole network with $O(N)$ moves by traversing the constructed MST. Thus, from Lemma 1 and $k \le gk'$, the following theorem is obtained.

Theorem 12 The group gossiping problem is solved with $O(N \log k' + M + gN)$ moves in asynchronous arbitrary networks.

In the following, we consider the lower bound. Clearly, since every link should be traveled by at least one agent, any algorithm requires $\Omega(M)$ moves. In addition, since arbitrary networks include ring networks, any algorithm requires $\Omega(N \log k' + gN)$ moves from Theorem 5. Therefore, we have the following theorem.

Theorem 13 Any algorithm for the group gossiping problem requires $\Omega(N \log k' + M + gN)$ moves in asynchronous arbitrary networks.

V. Conclusion

In this paper, we introduced a concept of agent groups and formulated the group gossiping problem. We also showed the upper and lower bounds on the total number of moves to solve the group gossiping problem for various networks. As a future work, we would like to study some problems related to agent groups. For example, it is interesting to consider the

group gathering problem, which requires agents in the same group to meet at a single node.

REFERENCES

[1] Y. Afek and E. Gafni. Time and message bounds for election in synchronous and asynchronous complete networks. In *Proc. of the 4th Annual ACM Symposium on Principles of Distributed Computing*, pages 186–195, 1985.

[2] D. Baba, T. Izumi, F. Ooshita, H. Kakugawa, and T. Masuzawa. Linear time and space gathering of anonymous mobile agents in asynchronous trees. *Theoretical Computer Science*, 478:118–126, 2013.

[3] J. Cao and S. K. Das. *Mobile agents in networking and distributed computing*. Wiley-Interscience, 2012.

[4] J. Czyzowicz, D. Kowalski, and A. Pelc. Time versus space trade-offs for rendezvous in trees. *Distributed Computing*, 27:95–109, 2014.

[5] S. Das, P. Flocchini, A. Nayak, S. Kutten, and N. Santoro. Map construction of unknown graphs by multiple agents. *Theoretical Computer Science*, 385:34–48, 2007.

[6] A. Dessmark, P. Fraigniaud, D. Kowalski, and A. Pelc. Deterministic rendezvous in graphs. *Algorithmica*, 46:69–96, 2006.

[7] Y. Dieudonné and A. Pelc. Anonymous meeting in networks. In *Proc. of 24th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2013.

[8] Y. Dieudonné, A. Pelc, and V. Villain. How to meet asynchronously at polynomial cost. In *Proc. of 2013 ACM Symp. on Principles of Distributed Computing*, pages 92–99, 2013.

[9] P. Fraigniaud, L. Gasieniec, D. Kowalski, and A. Pelc. Collective tree exploration. *Networks*, 48:166–177, 2006.

[10] P. Fraigniaud and A. Pelc. Delays induce an exponential memory gap for rendezvous in trees. *ACM Transactions on Algorithms*, 9, 2013.

[11] G. N. Frederickson and N. A. Lynch. Electing a leader in a synchronous ring. *Journal of ACM*, 34(1):98–115, 1987.

[12] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems*, 5(1):66–77, 1983.

[13] M. Koucký. Universal traversal sequences with backtracking. *Journal of Computer and System Sciences*, 65:717–726, 2002.

[14] E. Kranakis, D. Krizanc, and E. Markou. *The mobile agent rendezvous problem in the ring*. Synthesis Lectures on Distributed Computing Theory, Lecture # 1. Morgan & Claypool Publishers, 2010.

[15] D. B. Lange and M. Oshima. Seven good reasons for mobile agents. *Communications of the ACM*, 42(3):88–89, 1999.

[16] M. C. Loui, T. A. Matsushita, and D. B. West. Election in a complete network with a sense of direction. *Information Processing Letters*, 22(4):185–187, 1986.

[17] T. Masuzawa and S. Tixeuil. Quiescence of self-stabilizing gossiping among mobile agents in graphs. *Theoretical Computer Science*, 411:1567–1582, 2010.

[18] F. Ooshita, S. Kawai, H. Kakugawa, and T. Masuzawa. Randomized gathering of mobile agents in anonymous unidirectional ring networks. *IEEE Transactions on Parallel and Distributed Systems*, 25:1289–1296, 2014.

[19] P. Panaite and A. Pelc. Exploring unknown undirected graphs. *Journal of Algorithms*, 33:281–295, 1999.

[20] A. Pelc. Deterministic rendezvous in networks: A comprehensive survey. *Networks*, 59:331–347, 2012.

[21] G. L. Peterson. An $o(n \log n)$ unidirectional algorithm for the circular extrema problem. *ACM Transactions on Programming Languages and Systems*, 4(4):758–762, 1982.

[22] O. Reingold. Undirected connectivity in log-space. *Journal of ACM*, 55, 2008.

[23] M. Shibata, S. Kawai, F. Ooshita, H. Kakugawa, and T. Masuzawa. Algorithms for partial gathering of mobile agents in asynchronous rings. In *Proc. of 16th Int'l Conf. on Principles of Distributed Systems*, pages 254–268, 2012.

[24] M. Shibata, F. Ooshita, H. Kakugawa, and T. Masuzawa. Move-optimal partial gathering of mobile agents in asynchronous trees. In *Proc. of 21st Int'l Colloquium on Structural Information and Communication Complexity*, 2014.

[25] T. Suzuki, T. Izumi, F. Ooshita, H. Kakugawa, and T. Masuzawa. Move-optimal gossiping among mobile agents. *Theoretical Computer Science*, 393:90–101, 2008.

st-ordering問題を利用した極大DAG構成自己安定アルゴリズムに関する研究

大野 陽香[1]　片山 喜章[1]　増澤 利光[2]

1. はじめに

自己安定アルゴリズムとは, 任意の初期ネットワーク状況から実行を開始しても, 目的のシステム状況に到達することができる分散アルゴリズムである [1]. この性質から, 自己安定アルゴリズムとは, プロセスの一時的な故障によって分散システムがどのようなネットワーク状況に陥っても, 故障したプロセスが復旧すれば自動的に目的の状況 (正当な状況) に戻る. つまり, 自己安定アルゴリズムは一時故障に対して耐性のある分散アルゴリズムであり, 長期に渡ってネットワーク状況を安定に保ち, 一時故障に柔軟に対応することの求められる分散システムを動作させる場合に適している. 自己安定アルゴリズムは, どのような状況からでも有限時間内に安定するため, アルゴリズムを階層化することで, 複数のアルゴリズムを合成することができる. この手法を公平な合成という. 下位のアルゴリズムが安定すると, それを初期状況として上位のアルゴリズムが動作するため, 上位のアルゴリズムもやがて安定し, 目的の状況へと到達することができる. 公平な合成を用いることで, 任意のネットワークで問題を解くことも可能となる.

DAG(Directed Acyclic Graph) とは, 閉路のない有向グラフであり, 内向辺しか持たないプロセスをシンク (sink), 外向辺しか持たないプロセスをソース (source) と呼ぶ. DAG は分散アルゴリズムを動作させる初期グラフとしてよく利用される. また, 単一プロセス s,t が指定された 2 連結無向グラフ G において, s をソース, t をシンクとし, G 上の全ての辺が方向付けられているような DAG を Transport net といい, Transport net を構築する問題を Transport net 問題と呼ぶ.

st-ordering(st-numbering) 問題とは, 単一プロセス s,t が指定されたプロセス数 n の 2 連結無向グラフ G において, s に 1, t に n, s,t を除く各プロセスに 2 以上 $n-1$ 以下の整数 (st-order, st-number) を割り当てる問題である. ただし, s,t を除くプロセスには, 自分よりも大きい st-order を持つプロセスと, 自分よりも小さい st-order を持つプロセスの両方が隣接するように st-order を割り当てなければならない. st-ordering を行ったグラフ G の全ての辺に対して, 小さい st-order を持つプロセスから大きい st-order を持つノードへ向かうように方向づけを行うことで, Transport net を形成することができる. つまり, st-ordering 問題を解くことで Transport net 問題も解決することができる.

Transport net 問題, st-ordering 問題を解く自己安定アルゴリズムに関する既存研究について説明する.

Karaata ら [2] は, 二つの幅優先木を利用して Transport net を構築する自己安定アルゴリズムを提案した. Chaudhuri ら [3] は, 深さ優先木を利用して, st-ordering を行う自己安定アルゴリズムを提案した. 文献 [2], [3] で提案されているアルゴリズムは, いずれも相異なる識別子を持つプロセスで構成される連結度 2 以上の無向グラフでしか動作しない. また, 連結無向グラフの連結度が 1 である場合に, できるだけ多くの辺を方向づけるような DAG を構成するアルゴリズムは提案されていない. 本稿では, 連結度が 1 である無向連結グラフ上でも, できるだけ多くの辺を方向づけるような極大 DAG を構成するアルゴリズムを提案する. また, 文献 [2], [3] では, グラフ上の全てのプロセスが相異なる識別子を持っているが, 本稿では単一プロセス s,t と, 匿名プロセスで構成されるグラフを考える.

[1] 名古屋工業大学大学院工学研究科情報工学専攻
Graduate School of Computer Science and Engineering, Nagoya Institute of Technology
[2] 大阪大学大学院情報科学研究科コンピュータサイエンス専攻
Graduate School of Information Science and Technology, Osaka University

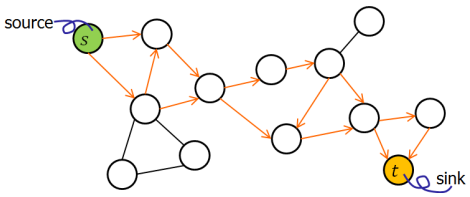


図 1　極大 DAG の例

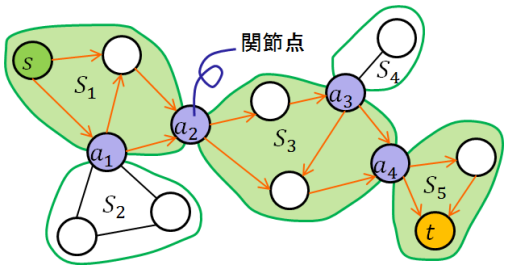


図 2　関節点で分解される連結成分

2. 極大 DAG 構成問題の定義

極大 DAG を次のように定義する.

定義 2.1.（極大 DAG） 単一プロセス s,t を持つ無向連結グラフ $G=(V,E)$ において, 次の条件を満たすように構成された DAG を極大 DAG という.

(1) s をソース, t をシンクとする

(2) s,t を除くプロセス $P_i \in V\backslash\{s,t\}$ は, シンクにもソースにもならない

(3) できるだけ方向付けられる辺を方向付ける

□

定義 2.1 のような DAG を構成する問題を, 極大 DAG 構成問題と定義する.

図 1 では, これよりも多く辺に方向づけを行うと, s,t 以外のプロセスがシンクやソースとなったり, 有向閉路が形成されたりするため, 極大 DAG の定義を満たさない. これ以上辺に方向付けることはできないため, "極大" であるといえる. また, 連結度 2 以上の無向グラフ上で極大 DAG を構成すると, 構成された極大 DAG は Transport net となる.

3. 提案アルゴリズム

本稿の提案アルゴリズムは関節点を利用する. 関節点とは, グラフ上から取り除くとそのグラフが非連結となるようなプロセスである. 図 2 より, 極大 DAG 上の方向を付けない辺は, 関節点に接続していることがわかる. 関節点は, 連結グラフを極大で連結な部分グラフである連結成分に分解することができる. 図 2 では, 関節点 $a_1, a_2, \cdots, a_4$ により連結成分 $S_1, S_2, \ldots, S_5$ に分解できる. 分割された連結成分は, 辺が全て有向辺で構成される連結成分と, 全て無向辺で構成される連結成分に分けられる. 辺が全て有向辺である各連結成分では, Transport net が構成されており, そのような連結成分内のプロセスと有向辺は全て, s から t への単純経路上にある. 図 2 では, 連結成分 S_1, S_3, S_5 でそれぞれ Transport net が構成されている. また, Transport net を構成している連結成分を挟むような関節点は, それぞれの Transport net のシンク, ソースとなっている. 図 2 では, 関節点 a_1 が S_1 のシンク, また S_3 のソースとなっており, a_4 は S_3 のシンク, また S_5 のソースとなっている. 一方, Transport net を構成する連結成分を挟まないような関節点 a_1, a_3 はそれぞれの Transport net 内のシンクやソースではない.

このことから, 関節点でグラフを連結成分に分解し, s から t への単純経路を含む各連結成分上で Transport net を構築することで極大 DAG を構成することができる. ただし, 極大 DAG を構成するには, Transport net を構築する連結成分に属するプロセスと構築しない連結成分に属するプロセス, つまり, 辺を方向付けるプロセスと, そうでないプロセスに分類する必要がある. そのため, 極大 DAG 上のプロセスを次の 3 種類に分類する.

st-node： それぞれの連結成分上に構成される Transport net のソース, またはシンクとなるプロセス

normal： Transport net を構成する連結成分上における st-node 以外のプロセス

NULL： Transport net が構成されない連結成分上に存在するプロセス

この分類に基づき, 分割した連結成分上で Transport net 構築を行う. st-node, normal に分類されたプロセスは, 極大 DAG において有向辺が接続するプロセスであり, また NULL に分類されたプロセスに接続する辺はすべて無向辺である.

Transport net を構築するために, st-ordering を行う. st-ordering を実行し, 割り当てられた st-order の大小によって辺を方向づけることによって各連結成分上に Transport net が構成される.

3.1 提案アルゴリズムの概略

本稿の提案アルゴリズムの概略を次に示す.

(1) 深さ優先木による関節点を求めるアルゴリズム [4] を用いて, グラフの関節点を求める

(2) 深さ優先木を利用して, s から t への経路を探索する

(3) Transport net を構築する連結成分上のプロセスと, Transport net を構築しない連結成分上のプロセスを分類する

(4) プロセスの分類に基づき, st-ordering 問題を解くアルゴリズム [3] を用いて, Transport net を構築するそれぞれの連結成分上で st-ordering を実行する

(5) st-ordering を行った連結成分上のプロセスの st-order の大小によって辺を方向づけることにより, 各連結成分で Transport net を構築し, 極大 DAG を構成する

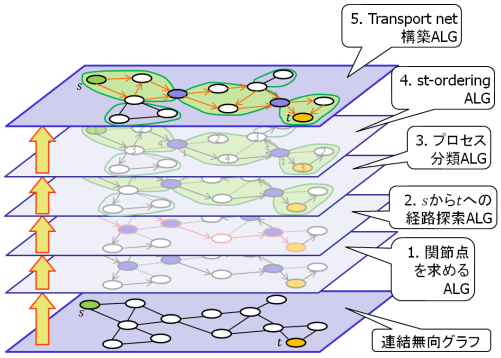


図 3　アルゴリズムの合成

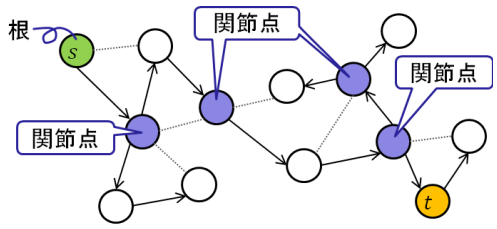


図 4　1. グラフの関節点を求める

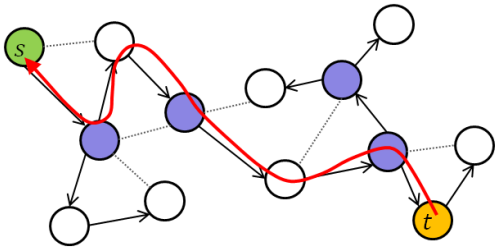


図 5　2. s から t への経路探索

これらを実現するために, 公平な合成を利用し, それぞれのアルゴリズムを合成する（図 3）.

それぞれのアルゴリズムについて, 説明する.

1. グラフの関節点を求める

[4] の提案アルゴリズムを用い, 深さ優先木を利用してグラフの関節点を求める. このとき, s を深さ優先木の根とする（図 4）.

2. s から t への単純経路の探索

先ほどの深さ優先木の木辺を利用して, t から根 s に向かって探索する（図 5）. これにより, Transport net を構築する連結成分と, Transport net を構築しない連結成分を区別することができる.

3. プロセスの分類

各プロセスを st-node, normal, NULL に分類する. プロセスを分類するために, 深さ優先木と関節点を用いて, 次のようなプロセスを定義する.

P_c : 深さ優先木において, ある関節点を取り除くと, s を含むグラフと分離してしまうような部分木の根プロセス

深さ優先木における P_c の親は, 必ず関節点である（図

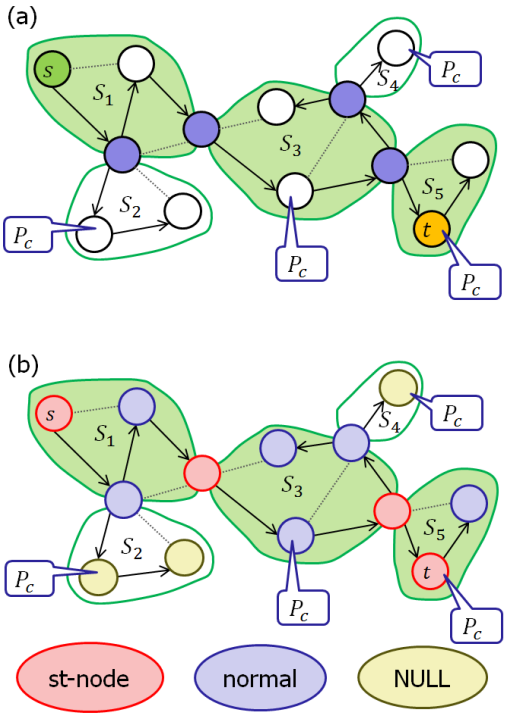


図 6　3. プロセスの分類

6（a））.

P_c は親である関節点を経由しないと, 根 s に到達できないプロセスであるため, P_c が s から t への経路上に存在しないならば, P_c の子孫には t が存在しない. つまり, P_c を根とする部分木上に Transport net が構築されることはなく, P_c とその子孫は NULL に分類される. 一方, P_c が s から t への経路上に存在するならば, P_c の親である関節点は s から t への任意の経路に必ず含まれる. よって, P_c は normal に分類され, P_c の親の関節点は st-node に分類される. P_c 以外のプロセスと st-node と判断されなかった関節点は, 親のプロセスの分類によって normal か, NULL かを判断する. 親が NULL であれば自分も NULL であり, そうでなかったら normal に分類される. また s, t は st-node に分類される（図 6（b））.

4. st-ordering

プロセスの分類に基づき, st-ordering を実行するアルゴリズム [3] を用いて, それぞれの連結成分で st-ordering を行う. このとき, NULL と分類されたプロセスは st-order を割り当てられない. st-node である関節点は, ソース s' としての変数と, シンク t' としての変数の両方を用意する. st-node である関節点に隣接する s に近い側のプロセスは関節点を t', t に近い側のプロセスは関節点を s' として扱う（図 7）.

5. Transport net を構築する各連結成分上で割り当てた st-order に基づき, Transport net をそれぞれ構築する. 小さい st-order を持つプロセスから, 大きい st-order

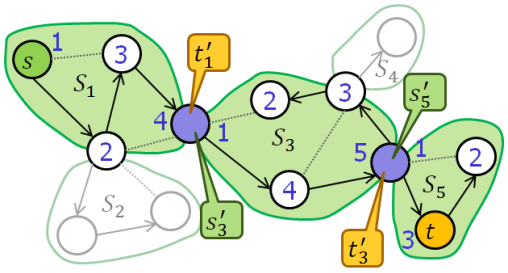


図 7　4. st-ordering

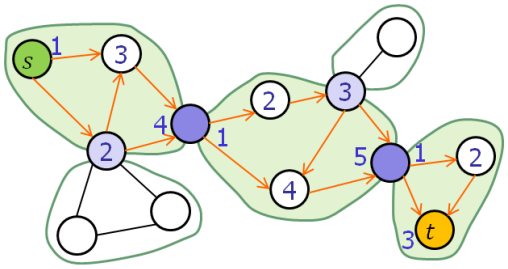


図 8　5. Transport net 構築

を持つプロセスへ向かうような方向付けを行うことで, Transport net が構築され, 極大 DAG が構成される（図 8）.

4. まとめと今後の課題

本稿では, 指定された単一プロセス s, t を持つ連結無向グラフ上に極大 DAG を構成するアルゴリズムを提案した.

今後の課題として, 正当性の証明と, アルゴリズム改良による時間計算量の削減が考えられる.

参考文献

[1] E. W. Dijkstra: *Self-stabilizing systems in spite of distributed control*, Comm. ACM 17 (11) pp.103-117 (1974).

[2] M. H. Karaata, P. Chaudhuri: *A Dynamic Self-Stabilizing Algorithm for Constructing a Transport Net*, Computing 68, pp. 143-161 (2002).

[3] Pranay Chaudhuri, Hussein Thompson: *A self-stabilizing algorithm for st-order problem*, The International Journal of Prallel, Emergent and Distributed Systems Vol. 23 (3), pp. 219-235 (2008).

[4] 大野陽香, 片山喜章: 深さ優先探索木によるグラフの関節点を求めるメッセージサイズ $\mathbf{O}(log_2 n)$ の自己安定アルゴリズムについて, 平成 26 年度東海支部連合大会 (2014).

Filling the Bandwidth Gap in Distributed Complexity for Global Problems*

Hiroaki Ookawa[1] and Taisuke Izumi[1]

Graduate School of Engineering, Nagoya Institute of Technology
cht15031@nitech.jp, t-izumi@nitech.ac.jp

Abstract. Communication complexity theory is a powerful tool to bound time complexity lower bounds of distributed algorithms for global problems such as minimum spanning tree (MST) and shortest path. While it often leads the nearly-tight lower bounds for many problems, polylogarithmic complexity gaps still lies between the currently best upper and lower bounds. In this paper, we propose a new approach for filling the gaps. Using this approach, we achieve tighter deterministic lower bounds for MST and shortest path. Specifically, for those problems, we show the deterministice $\Omega(\sqrt{n})$-round lower bound for graphs with $O(n^\epsilon)$ hop-count diameter, and the deterministice $\Omega(\sqrt{n/\log n})$ lower bound for graphs with $O(\log n)$ hop-count diameter. The main idea of our approach is to introduce a new function we call *permutation identity* and utilize its two-party communication complexity lower bound.

1 Introduction

In distributed computing theory, many graph problems are naturally treated as problems in networks, where each vertex represents a computing entity and each edge does a communication link between two nodes. The theory of *distributed graph algorithms* has been developed so far for efficient in-network computation of graph problems. A crucial factor of distributed graph algorithms is *locality*. Local algorithms require each node to compute its output only by the interaction to the nodes within a bounded distance smaller than the diameter of the network. In other words, local algorithms must terminate within $o(D)$ rounds, where D is the hop-count diameter of the network. There are a number of problems allowing local solutions: Maximal matchings, colorings, independent sets, and so on. On the other hand, some of other graph problems (e.g., minimum spanning tree shortest path, minimum cut) are known to have no local solution. They are called *global problems*. By the definition, the (worst-case) run of any algorithm for global problems inherently takes $\Omega(D)$ rounds.

For both local and global problems, the time complexity analysis for distributed algorithms (i.e., *distributed complexity theory*) are one of the important topics in distributed algorithms. In this paper, we focus on the distributed complexity of two well-known global problems: Minimum spanning tree (MST) and

* This work is supported in part by ——

shortest s-t path. As we stated above, these problems have trivial $\Omega(D)$-round lower bounds. If the communication bandwidth of each link is not bounded, every global problem has an optimal-time algorithm with $O(D)$ rounds: A process aggregates all the information of the network, and computes the result locally. However the assumption of so rich bandwidth is far from real systems, and thus the challenge of global problems is to solve them in the environment with limited bandwidth. Theoretically, such environments are called as the *CONGEST model*, where processes work under the round-based synchrony, and each link can transfer $O(\log n)$-bit messages per one round.

A seminal results about the lower bounds for global problems is the one by Das Sarma et al. [1], which exhibits that many problems, including MST and shortest s-t path, are more expensive tasks. Precisely, it shows that $\Omega(\sqrt{n}/\log n + D)$-round lower bounds hold for many global problems even D is small (i.e., $D = O(\log n)$). The core of this result is a general framework to obtain the lower bounds based on the reduction from two-party communication complexity by Yao [14]. Two-party communication complexity is a theory to reveal the amount of communication to compute a global function whose inputs are distributed among two players. The reduction framework in [1] induces the hardness of MST and shortest s-t path from the two-party communication complexity of *set-disjointness* function. While the framework is a powerful tool to bound the time compelxity of global problems, all the bounds led by that approach have a form of $\Omega(f(n)/(m \log n))$, where $f(n)$ is the amount of information inherently exchanged among the networks to solve the target problem, and m is the number of links where the information must be transferred, and $\log n$ factor is the bandwidth of each link (that is, $m \log n$ is the amount of information transmittable within a round). On the other hand, these lower bounds does not strictly match the known corresponding upper bounds, which typically has the form of $O(f(n)\mathit{polylog}(n)/m)$. That is, for many global problems, the currently best bounds still have (poly)logarithmic gaps.

The primary objective of this paper is to fill those gaps. For that goal, we propose a new two-party function whose deterministic communication complexity is slightly more expensive than set-disjointness, called *permutation identity*, and new reductions using it on the top of the framework by Das Sarma et al. [1]. Our contribution is to give tighter deterministic lower bounds for MST and shortest s-t path. Specifically, for those problems, we show the deterministice $\Omega(\sqrt{n})$-round lower bound for graphs with $O(n^\epsilon)$ hop-count diameter, and the deterministice $\Omega(\sqrt{n/\log n})$ lower bound for graphs with $O(\log n)$ hop-count diameter. The comparision with the prior work are shown in Table 1. As far as we consider the complexity of *deterministic* and *exact* computation, our bound beats the currently best ones. It also should be noted that the MST problem is almost closing the gap because the currently best upper bound is $O(\sqrt{n}\log^* n + D)$ rounds [3].

paper	bound	problem	comments
Garay et al. [3]	$O(\sqrt{n}\log^* n + D)$	MST	deterministic
Nanongkai [10]	$O(\sqrt{n}D^{1/4} + D)$	SP	$(1+o(1))$-approximation single-source SP
Das Sarma et al.[1]	$\Omega(\sqrt{\frac{n}{\log n}})$	SP,MST	randomized $\alpha(n)$-approximation $D = O(n^\epsilon)$ $(\epsilon < 1/2)$
Das Sarma et al.[1]	$\Omega(\frac{\sqrt{n}}{\log n})$	SP, MST	randomized $\alpha(n)$-approximation $(D = \Theta(\log n))$
This paper	$\Omega(\sqrt{n})$	SP, MST	deterministic $D = O(n^\epsilon)$ $(\epsilon < 1/2)$
This paper	$\Omega(\sqrt{\frac{n}{\log n}})$	SP, MST	deterministic $D = O(\log n))$

Table 1: Comparison with the prior work. SP (resp. MST) means shortest s-t path (res. minimum spanning tree).

2 Related Work

The paper by Das Sarma et al. [1] is the first one explicitly considering the distributed verification problem, which has given a general framework to lead lower bounds and approximation hardness for a vast class of problems. It is used in several following papers to obtain the complexity for a number of graph problems: Weighted/unweighted diameter and all-pair shortest paths [5, 7, 8, 12], minimum cuts [4, 10], distance sketches [7], weighted single-source shortest paths [7, 10], fast random walks [11], and so on.

While the framework by Das Sarma et al. [1] pointed out a general relationship interconnecting the communication complexity theory and distributed complexity theory, the construction of worst-case instances used in the framework is much inspired by the earlier papers leading the time lower bound for the distributed MST construction [2, 9, 13].

3 Preliminaries

3.1 Round-Based Distributed Systems

A distributed system consists of n nodes interconnected with communication links. We model it by a weighted graph $G = (V, E, w)$, where is the set of nodes, $E \subseteq V \times V$ is the set of links (edges), and $w : E \to \mathbb{R}$ is a weight function. The hop-count diameter of G (i.e., the diameter of the unweighted graph (V, E)) is

denoted by D. Executions of the system proceed with a sequence of consecutive rounds. In each round, each process sends a (possibly different) message to each neighbor, and within the round, all messages are received. After receiving the messages, the process performs local computation. Throughout this paper, we restrict the number of bits transmittable through any communication link per one round to $O(\log n)$ bits. This is known as the CONGEST model.

3.2 Distributed MST and Single-source shortest paths

In this paper we consider two popular graph problems: Minimum spanning tree (MST) and shortest s-t path. The distributed minimum spanning tree problem requires the system to find the MST of the (weighted) network. After the computation by distributed MST algorithms, each node must identify the incident edges constituting the MST. In the shortest s-t path problem, the algorithm takes two input nodes s and t, and computes a shortest path between them. After the computation, each node on the computed path must identify the incident edge toward s and the distance from s.

3.3 Two-Party Communication Complexity

Communication complexity, which is first introduced by Yao [14], reveals the amount of communication to compute a global function whose inputs are distributed in the network. The most successful scenario in communication complexity is *two-party* communication complexity, where two players, called Alice and Bob, respectively have their inputs $x, y \in U$ (where U is the domain of inputs), and compute a global function $f : U \times U \to \{0, 1\}$. The communication complexity of a two-party protocol is the number of one-bit messages exchanged by the protocol for the worst case input (if the protocol is randomized, it is defined as the expected number of bits exchanged for the worst-case input). One of the most popular functions in two-party communication complexity is *set-disjointness*, which is the function over two k-bit 0-1 vectors $x, y \in 0, 1^k$ and return value one if and only if there exists a common position $i \in [0, k-1]$ such that i-th bits of x and y are one.

While the known best lower bounds for MST and shortest s-t path is obtained by using the communication complexity of set-disjointness, it does not suffice to have a stronger bound we will prove. Thus in this paper, we introduce a new function called *permutation identity*, which is defined as follows:

Definition 1. *Let $\pi_A, \pi_B : [1, N] \to [1, N]$ be permutations over $[1, N]$. the permutation identity function $ident_N$ is defined as follows:*

$$ident_N(\pi_A, \pi_B) = \begin{cases} 1 \text{ if } \forall i \in [1, N] : \pi_A \circ \pi_B(i) = i, \\ 0 \text{ otherwise}, \end{cases}$$

where $\pi_A \circ \pi_B$ means the composition of π_A and π_B, that is, $\pi_A \circ \pi_B(i) = \pi_A(\pi_B(i))$.

Theorem 1. *The deterministic communication complexity of two-party permutation identity over* $[1, N]$ *is* $\Omega(n \log N)$ *bits.*

We also show a fundamental lemma for the permutation identity function, which is used in the following sections.

Lemma 1. *Let* π_A *and* π_B *be permutations over* $[1, N]$*. If* $\pi_A \circ \pi_B$ *is not identical, there exists* $i \in [1, N]$ *such that* $\pi_A \circ \pi_B(i) < i$ *holds.*

For lack of space, the proof for the theorem and lemma above are presented in the appendix.

4 General Framework for the Reduction

The proof of our lower bounds basically follows the framework by Das Sarma et al. [1]. The core of this framework is the reduction from two-party computation via a hard instance for distributed computation. In this section, we introduce the framework which is sligtly modified for our proof.

4.1 Graph Construction

The graph we construct is denoted by $G(N, M)$, where N and M are design parameters of the graph. For simplicity of the argument, throughout the paper, we assume that $M + 1$ is a power of 2, i.e., $M = 2^p - 1$ for some nonnegative integer p. Note that the assumption is not essential and it is not difficult to remove it. The graph is built by the following steps:

1. Prepare N paths of length M, each of which is denoted by P_i $(1 \leq i \leq N)$. The nodes constituting P_i are identified by $v_i^0, v_i^1, \cdots, v_i^M$ from left to right.
2. Add edges (v_i^0, v_j^1) and $(v_i^{(M-1)}, v_j^M)$ for any $i, j \in [1, N]$.
3. Add edges $(v_i^0, v_{(i+1)}^0)$ and $(v_i^M, v_{(i+1)}^M)$ for any $i \in [1, N-1]$.
4. Construct a complete binary tree $T(M)$ with $M + 1$ leaf. where each leave is labeled by $u^0, u^1, \cdots, u^M$ from left to right.
5. Add edges (u^i, v_j^i) for any $i \in [0, M]$ and $j \in [1, N]$.

The weight of each edge depends on concrete reductions, which is determined later. Note that the number n of nodes in $G(N, M)$ is $\Theta(NM)$, and its diameter is $D = O(\log n)$. We also define the sets of nodes $A = \{u^0\} \cup \{v_i^0, v_i^1 | i \in [1, N]\}$ and $B = \{u^M\} \cup \{v_i^{(M-1)}, v_i^M | i \in [1, N]\}$. The whole construction is illustrated in Figure 1. For this graph, we can show the following theorem.

Theorem 2 (Das Sarma et al. [1]). *Let* $\mathcal{A}$ *be any algorithm running on the graph* $G(N, M)$ *with an arbitrary edge-weight function. Then there exists a two-party protocol satisfying the following three properties:*

- *At the beginning of the protocol, Alice (resp. Bob) knows the whole topological information of* $G(N, M)$ *except for the subgraph induced by* B *(resp.* A*),*

- *after the run of the protocol, Alice and Bob output the internal states of the processes in A and B at round $(M-3)/2$ in the execution of $\mathcal{A}$ on $G(N,M)$, respectively, and*
- *the protocol consumes at most $O(M(\log MN)^2)$-bit communication.*

While the graph used in this paper is a slightly modified version of the original construction in [1], the theorem above is proved In the almost same way. So we just quote it without the proof.

4.2 Networked Two-Party Computation

To obtain the lower bounds for distributed algorithms, we uses a variation of the two-party computation problem in distributed settings. We assume that Alice and Bob are placed at two nodes in a network of n nodes, and have inputs $x \in U$ and $y \in U$ for two-party function $f : U \times U \to \{0,1\}$, respectively. It is also assumed that each node in the network (including ones other than Alice and Bob) knows everything (i.e., the complete knowledge of the network topology) except for the inputs held by Alice and Bob. Then all nodes must work cooperatively for outputting the value of $f(x,y)$ as fast as possible. In what follows, we call this problem setting the *networked two-party computation* (and the networked permutation identity problem if $f = ident_N$). Note that the measurement of the networked two-party computation is not the amount of communication, but the number of rounds.

Obviously the time complexity of networked two-party computation problems relies on the target function f and the topology of the network. An useful consequence from Theorem 2 is that we can transform the communication lower bound for any two-party computation into the time lower bound for its networked version. In the original version by Das Sarma et al. [1], the transformation from two-party set-disjointness is considered. Here we problem the similar fact from two-party permutation identify function (the proof are in the appendix):

Theorem 3. *Let $M = N/\log N$. For any deterministic algorithm $\mathcal{A}$ solving the networked permutation identity over $[1,N]$ in $G(N,M)$, its worst-case running time is $\Omega(\sqrt{n/\log n})$ rounds.*

4.3 Lower bound for MST

We show the reduction from the networked permutation identity to MST. In this reduction we construct an instance of the MST problem by virtually assigning some weight to each edge in $G(N,M)$ for $M = N/\log N$ to encode an instance (π_A, π_B) of permutation identity over $[1,N]$. After the construction of the MST, Alice and Bob can determine the identity of $\pi_A \circ \pi_B$ from the computed MST. Let $L(\pi_A, \pi_B)$ be the instance of the MST problem corresponding to the permutation identity instance (π_A, π_B), which is constructed by defining edge-weight function w as follows:

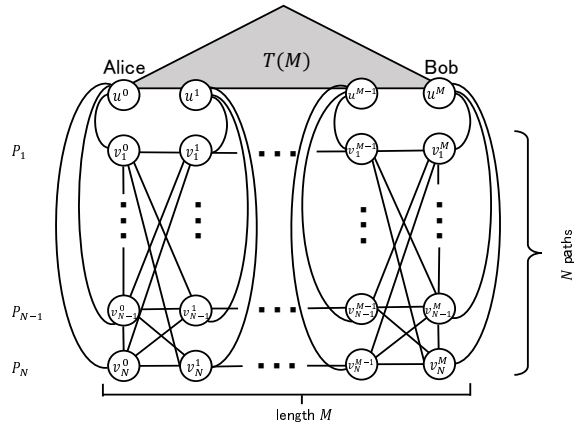


Fig. 1: Construction of $G(N, M)$

1. For any $i \in [1, N]$ and $j \in [1, M-1]$, $w(u^j, v_i^j) = 100NM$.
2. For any $i \in [1, N-1]$, $w(v_i^0, v_{i+1}^0) = 100NM$ and $w(v_i^M, v_{i+1}^M) = 100NM$.
3. For any $i \in [1, N]$, $w(u^0, v_i^0) = 2i$ and $w(u^M, v_i^M) = 2i - 1$.
4. For any $i, j \in [1, N]$, $w(v_i^0, v_j^1) = 1$ if $\pi_A(j) = i$. Otherwise $w(v_i^0, v_j^1) = 100NM$. Similarly, For any $i, j \in [1, N]$, $w(v_i^{M-1}, v_j^M) = 1$ if $\pi_B(j) = i$. Otherwise $w(v_i^{M-1}, v_j^M) = 100NM$.
5. All other edges have weight one.

The construction of $L(\pi_A, \pi_B)$ is illustrated in Figure 2. Let $E_A = \{(u^0, v_i^0) | i \in [1, N]\}$ and $E_B = \{(u^M, v_i^M) | i \in [1, N]\}$. The following lemma is the core of the reduction.

Lemma 2. *The MST of $L(\pi_A, \pi_B)$ contains no edge in E_A if and only if $\pi_A \circ \pi_B$ is identical.*

Proof. Let P'_i be the path consisting of the nodes $v^0_{\pi_A(\pi_B(i))}, v^1_{\pi_B(i)}, v^2_{\pi_B(i)}, \cdots, v^{M-1}_{\pi_B(i)}, v^M_i$. Following the standard greedy algorithm for constructing the MST, every edge with weight one is contained in the MST. Thus, the components $P'_1, P'_2, \cdots, P'_N$ and $T(M)$ are MST fragments. A component P'_i is merged with $T(M)$ by choosing either $(u^0, v^0_{\pi_A(\pi_B(i))})$ or (u^M, v^M_i) (all other edges merging them are too heavy (i.e., $100NM$) and never chosen as a MST edge). If $\pi_A \circ \pi_B$ is identical, $\pi_A(\pi_B(i)) = i$ holds. Thus we have $w(u^0, v^0_{\pi_A(\pi_B(i))}) = 2i$ and $w(u^M, v^M_i) = 2i - 1$ for any $i \in [1, N]$. This implies that P'_i is merged with $T(M)$ by edge $(u^0, v^0_{\pi_A(\pi_B(i))})$ (Figure 3). On the other hand, if $\pi_A \circ \pi_B$ is not identical, from Lemma 1, there exists at least one i satisfying $\pi_A \circ \pi_B(i) < i$. Then for such i we have $w(u^0, v^0_{\pi_A(\pi_B(i))}) \leq 2(i-1)$ and $w(u^M, v^M_i) = 2i - 1$. Thus P'_i and $T(N)$ is merged with edge $w(u^0, v^0_{\pi_A(\pi_B(i))}) \in E_A$ (Figure 4). The lemma is proved □

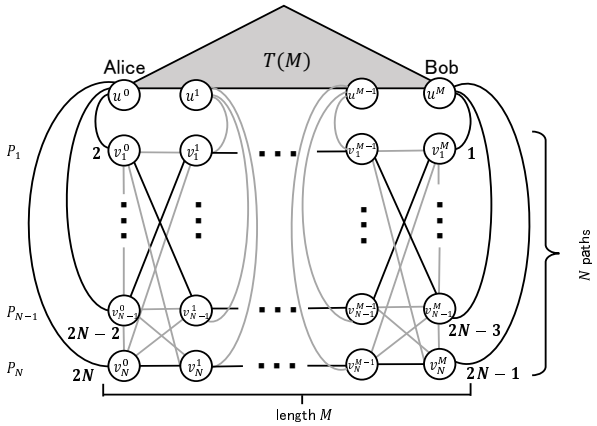


Fig. 2: An example of $L(\pi_A, \pi_B)$. Every unlabeled edge has weight one. All the edges with weight $100NM$ are grayed out.

Lemma 3. *If an algorithm $\mathcal{A}$ solves the MST problem in $L(\pi_A, \pi_B)$ within r rounds, there exists an algorithm solving the networked permutation identity over $[1, N]$ in $G(N, M)$ within $O(r)$ rounds.*

Proof. At the round one and two, each node sets up the instance $L(\pi_A, \pi_B)$ of the MST problem according to the input (π_A, π_B). Then the system runs the MST algorithm $\mathcal{A}$. From lemma 2, no edge in E_A is not included in the constructed MST if $\pi_A \circ \pi_B$ is identical. Then, after the construction of the MST, each node v_i^0 $(i \in [1, N])$ sends to u^0 the information that no incident edge is contained in the MST. By this information, u^0 can determine whether $\pi_A \circ \pi_B$ is identical or not. That is, the networked permutation identity is solved in $G(N, M)$ within $O(r)$ rounds. □

Combining Theorem 3 and Lemma 3, we have the main theorem below.

Theorem 4. *Any deterministic algorithm solving the MST problem, its worst-case running time is $\Omega(\sqrt{n/\log n})$ rounds.*

4.4 Lower Bound for Shortest *s*-*t* Path

The argument in this section is almost the same as Section 4. We construct a graph $L'(\pi_A, \pi_B)$ by fixing a weight function w for the network $G(N, N \log N)$. The weight function w is defined as follows:

1. For any $i \in [1, N]$ and $j \in [0, M]$, $w(u^j, v_i^j) = 100NM$.
2. For any $i \in [1, N-1]$, $w(v_i^0, v_{i+1}^0) = 1$ and $w(v_i^M, v_{i+1}^M) = 1$.

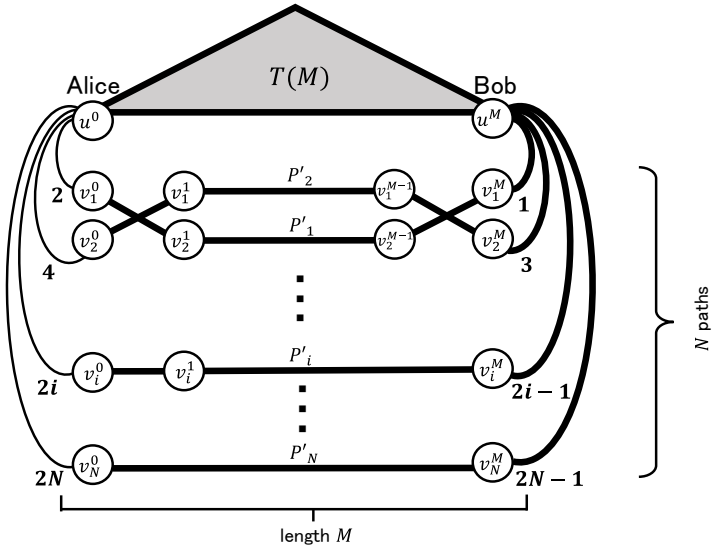


Fig. 3: Graph $L(\pi_A, \pi_B)$ when $\pi_A \circ \pi_B$ is identical.

3. For any $i, j \in [1, N]$, $w(v_i^0, v_j^1) = 100NM$ if $\pi_A(j) = i$. Otherwise $w(v_i^0, v_j^1) = 100NM$. Similarly, For any $i, j \in [1, N]$, $w(v_i^{M-1}, v_j^M) = 1$ if $\pi_B(j) = i$. Otherwise $w(v_i^{M-1}, v_j^M) = 100NM$.
4. For any $i \in [1, N]$ and $j \in [1, M-1]$, $w(v_i^j, v_i^{j+1}) = 1$.
5. Every edge in $T(M)$ has weight $100NM$.

We also define $s = v_1^0$ and $t = v_N^M$. Then, we have the following lemma:

Lemma 4. *In graph $L'(\pi_A, \pi_B)$, the length of the shortest s-t path is $N+M-1$ if and only if $\pi_A \circ \pi_B$ is identical.*

Proof. The path $v_1^0, v_2^0, \cdots v_N^0, v_N^1, v_N^2, \cdots V_N^{M-1}, v_N^M$ is the s-t path of length $N + M - 1$. We first show that this is the shortest path if $\pi_A \circ \pi_B$ is identical. Since the length of the shortest path between s and t is at most $N + M - 1$, it contains no edge with weight $100NM$. Thus we omit those edges. Then, if $\pi_A \circ \pi_B$ is identical, v_i^0 and v_i^M are connected by a path of length M. Thus, the graph (where all isolated nodes in $T(M)$ are removed) becomes a subdivision of a ladder graph. It is not difficult to see that the shortest path between s and t is $N + M - 1$ (Figure 5).

We next consider the case where $\pi_A \circ \pi_B$ is not identical. Then, from Lemma 1, there exists i satisfying $\pi_A \circ \pi_B(i) < i$. Then, we have an s-t path $v_1^0, v_2^0, \cdots v_{\pi_A(\pi_B(i))}^0, v_{\pi_B(i))}^1, v_{\pi_B(i))}^2, \cdots, v_{\pi_B(i))}^{M-1}, v_i^M, v_{i+1}^M, \cdots, v_N^M$ of length less than $N + M - 1$ (Figure 6). The lemma is proved □

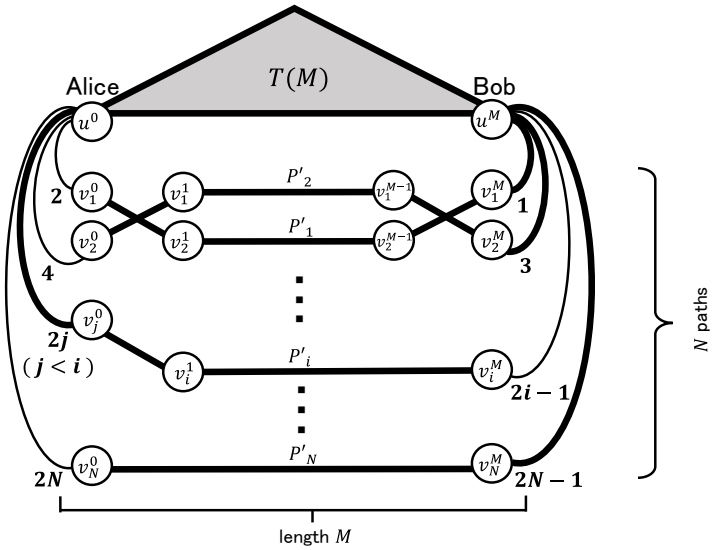


Fig. 4: Graph $L(\pi_A, \pi_B)$ when $\pi_A \circ \pi_B$ is not identical.

Lemma 5. *If an algorithm $\mathcal{A}$ solves the shortest s-t path problem in $L'(\pi_A, \pi_B)$ within r rounds, there exists an algorithm solving the networked permutation identity over $[1, N]$ within $O(r)$ rounds.*

The proof is almost the same as that for Lemma 3, and thus we omit it. Finally we obtain the following theorem.

Theorem 5. *Any deterministic algorithm solving the shortest s-t path problem, its worst-case running time is $\Omega(\sqrt{n/\log n})$ rounds.*

5 Lower bound for the graphs with $O(n^\epsilon)$ hop-count diameter

For the case of larger diameter graphs, we obtain stronger bounds by slightly modifying the framework graph $G(N, M)$. Since the fundamental idea has been proposed in the prior work [1], we state only the results in this paper. The Theorem 4 and 5 are extended as follows:

Theorem 6. *Any deterministic algorithm solving the MST problem or the shortest s-t path problem, its worst-case running time is $\Omega(\sqrt{n/\log n})$ rounds for graphs with diamter $O(\log n)$. In addition, for graphs with diameter $O(n^\epsilon)$ $(0 < \epsilon < 1/2)$, the worst-case running time is $\Omega(\sqrt{n/\log n})$ rounds.*

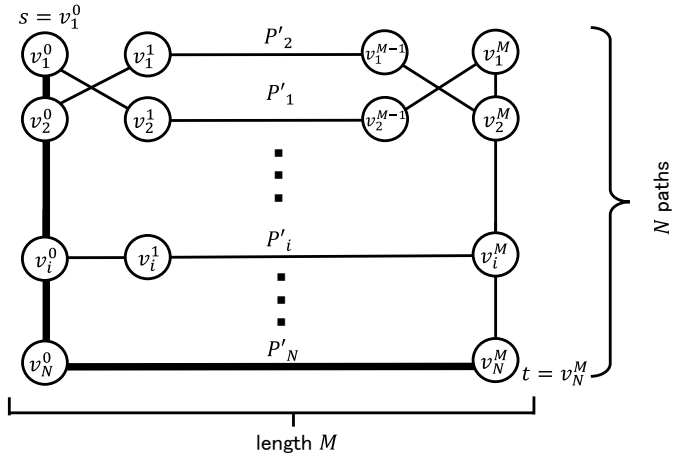


Fig. 5: Example of shortest path (marked with big edges) when $\pi_A \circ \pi_B$ is identical

6 Concluding Remarks

In this paper, we inroduced a new function called *permutation identity*. By using the seminal reduction framework by Das Sarma et al.[1], we show the deterministic $\Omega(\sqrt{\frac{n}{\log n}})$-round lower bounds for MST and shortest s-t path. Furtuermore, for graph with for graphs with $O(n^\epsilon)$ hop-count diameter, we obtained $\Omega(\sqrt{n})$ lower bound, For the MST problem, this lower bound is almost closing the logarithmic gap because the best upper bound is $O(\sqrt{n}\log^* n + D)$.

References

1. Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed verification and hardness of distributed approximation. In *Proc. of the 43rd Annual ACM Symposium on Theory of Computing*, pages 363–372, 2011.
2. Michael Elkin. An unconditional lower bound on the hardness of approximation of distributed minimum spanning tree problem. In *Proc the 30th ACM Symposium on Theory of Computing(STOC)*, pages 331 – 340, 2004.
3. Juan A. Garay, Shay Kutten, and David Peleg. A sublinear time distributed algorithm for minimum-weight spanning trees. *SIAM Journal on Computing*, 27(1):302–316, 1998.
4. Mohsen Ghaffari and Fabian Kuhn. Distributed minimum cut approximation. In *Proc. of 27th International Symposium on Distributed Computing (DISC)*, pages 1 – 15, 2013.
5. Stephan Holzer and Roger Wattenhofer. Optimal distributed all pairs shortest paths and applications. In *Proc. of the 2012 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 355–364, 2012.

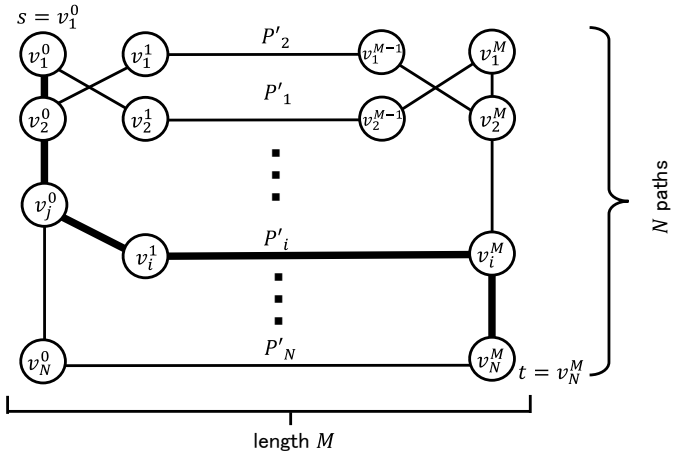


Fig. 6: Example of shortest path (marked with big edges) when $\pi_A \circ \pi_B$ is not identical ($\pi_A \circ \pi_B(i) = j$)

6. E. Kushilevitz and N. Nisan. *Communication Complexity*. Cambridge University Press, 1997.
7. Christoph Lenzen and Boaz Patt-Shamir. Fast routing table construction using small messages: Extended abstract. In *Proc. of the 45th Annual ACM Symposium on Symposium on Theory of Computing (STOC)*, pages 381–390, 2013.
8. Christoph Lenzen and David Peleg. Efficient distributed source detection with limited bandwidth. In *Proc. of the 2013 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 375–382, 2013.
9. Zvi Lotker, Boaz Patt-Shamir, and David Peleg. Distributed mst for constant diameter graphs. *Distributed Computing*, 18(6):453–460, 2006.
10. Danupon Nanongkai. Distributed approximation algorithms for weighted shortest paths. In *Proc. of the 46th ACM Symposium on Theory of Computing (STOC)*, 2014.
11. Danupon Nanongkai, Atish Das Sarma, and Gopal Pandurangan. A tight unconditional lower bound on distributed random walk computation. In *Proc. of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 257–266, 2011.
12. David Peleg, Liam Roditty, and Elad Tal. Distributed algorithms for network diameter and girth. In *Proc. of the 39th International Colloquium Conference on Automata, Languages, and Programming (ICALP)*, pages 660–672, 2012.
13. David Peleg and Vitaly Rubinovich. A near-tight lower bound on the time complexity of distributed minimum-weight spanning tree construction. *SIAM Journal on Computing*, 30(5):1427–1442, 2000.
14. Andrew Chi-Chih Yao. Some complexity questions related to distributive computing(preliminary report). In *Proc. of the 11th Annual ACM Symposium on Theory of Computing (STOC)*, pages 209–213, 1979.

A Omitted Proofs

A.1 Proof of Theorem 1

Proof. The proof of this theorem is almost the same as the well-known result for the two-party equality function. More precisely, the proof follows the *fooling-set* argument. Let $f : U \times U \to \{0, 1\}$ be a two-party function over input domain U. A subset $S \subset U \times U$ is called a *fooling set* of function f if the following conditions are satisfied for some $z \in \{0, 1\}$: (1) For any $(x, y) \in U \times U$, $f(x, y) = z$, and (2) for any distinct inputs $(x_1, y_1), (x_2, y_2) \in S$, either $f(x_1, y_2) \neq z$ or $f(x_2, y_1) \neq z$. It is well-known that the deterministic communication complexity of function f is bounded by $\Omega(\log |S|)$ (the detailed argument is found in the standard textbook of communication complexity theory [6]).

Thus it suffices to show that function $ident_N$ has a fooling set S of size $2^{\Omega(N \log N)}$. We constitute S by including all pairs (π_A, π_B) such that $\pi_A \circ \pi_B$ becomes the identical mapping. Then for any $(\pi_A, \pi_B) in S$, there is no other mapping π_C such that $\pi_A \circ \pi_C$ or $\pi_C \circ \pi_B$ becomes identical. Thus the set S clearly satisfies the conditions of fooling sets. Since the cardinarity of S is $N!$, we have the communication complexity lower bound of $\Omega(\log N!) = \Omega(N \log N)$ bits. The theorem is proved. □

A.2 Proof of Lemma 1

Proof. Suppose for contradiction that $\pi_A \circ \pi_B$ is not identical but $\pi_A \circ \pi_B(i) \geq i$ holds for any $i \in [1, N]$. Then, clearly we have $\pi_A \circ \pi_B(N) = N$, and thus we have $\pi_A \circ \pi_B(N - 1) = N - 1$, $\pi_A \circ \pi_B(N - 2) = N - 2$, $\cdots$, $\pi_A \circ \pi_B(0) = 0$. Consequently $\pi_A \circ \pi_B$ becomes identical. It is a contradiction. □

A.3 Proof of Theorem 3

Proof. Since $M = N/\log N$, we have $n = \Theta(N \cdot N/\log N) = \Theta(N^2/\log N)$. Thus we also have $\Theta(\log N) = \Theta(\log n)$ and thus, $n \log n = \Theta(N^2)$ holds. It implies $N = \sqrt{n \log n}$ and $M = \sqrt{n/\log n}$. To prove the lemma, it suffices to show that the running time of $\mathcal{A}$ in $G(N, M)$ is $\Omega(M)$ rounds. Suppose for contradiction that $\mathcal{A}$ terminates within $o(M)$ rounds. Consider the network $G(N, M)$ where Alice and Bob are respectively placed at u_0 and u_M. Then, following Theorem 2, we can construct a two-party permutation identity protocol over $[1, N]$ by simulating the execution of $\mathcal{A}$ in $G(N, M)$. That is, Alice and Bob (in the two-party computation) first set up the initial configuration of $\mathcal{A}$ by installing their own inputs x and y and run the simulation. After the simulation, they output the computation result $ident_N(x, y)$ as the result of the two-party computation. This two-party protocol consumes $o(M(\log N)^2) = o(N \log N)$ bits. It contradicts Theorem 1. □

非負行列分解の絶対値誤差最小化について

橋村 勇志　　山内 由紀子　　来嶋 秀治　　山下 雅史
九州大学 工学部 電気情報工学科

1　はじめに

この論文では，非負値行列因子分解 (NMF) に対する絶対値誤差最小化について述べる．ここでは，行列の全ての要素が 0 以上である非負行列についての議論をする．

2　準備

非負値行列因子分解では，$m \times n$ 行列 V が与えられたとき，２つの行列 W, H を用いて $V \fallingdotseq WH$ と近似することを考える．このとき，a は与えられるパラメータとし，W は $m \times a$ 行列，H は $a \times n$ 行列とする．ただし、$a < m$，$a < n$ とする．また，便宜のため $X = WH$ と表す.

本稿では，W, H を求めるアルゴリズムを以下に提案する．

3　アルゴリズム

まず，$X = WH$ で求められる行列 X と，元の行列 V の誤差について議論する．誤差の評価関数 $f(W, H)$ を，

$$f(W,H) = \sum_{i=1}^{m}\sum_{j=1}^{n} |v_{ij} - x_{ij}|$$

と定義する．すなわち，絶対値誤差である．

本稿では，絶対値誤差 $f(W, H)$ の値を最小化するアルゴリズムを提案する．アルゴリズムは、劣勾配法に基づく．

(1)W,H の初期値を定める． $n = 1$ とする．

(2)W の劣勾配をもとに W を更新する．

$$W^{(n+1)} = W^{(n)} - s^{(n)}\frac{\partial f(W,H)}{\partial W} \quad \text{ただし，} s^{(n)} = \frac{1}{n}$$

(3)H の劣勾配をもとに H を更新する．

$$H^{(n+1)} = H^{(n)} - s^{(n)}\frac{\partial f(W,H)}{\partial H} \text{ ただし，} s^{(n)} = \frac{1}{n}$$

(4) $n = n + 1$ として，(2),(3) を繰り返す.
(5) 定められた回数繰り返し出力する.

また，劣微分については，ここでは，$y = |x|$ の劣微分を

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \begin{cases} -1 & (x < 0) \\ 0 & (x = 0) \\ 1 & (x > 0) \end{cases}$$

とする．これに対応して，説明のために符号関数 $\mathrm{sgn}(x)$ を，

$$\mathrm{sgn}(x) = \begin{cases} -1 & (x < 0) \\ 0 & (x = 0) \\ 1 & (x > 0) \end{cases}$$

と定める.

このとき，劣勾配ベクトルは各 W, H 行列の要素ごとに計算できる．行列 Y を $Y = |V - X|$ と定義し，

$$\frac{\partial f(W,H)}{\partial w_{ij}} = \sum_{k=1}^{n} -h_{jk}\mathrm{sgn}(y_{ik})$$

$$\frac{\partial f(W,H)}{\partial h_{ij}} = \sum_{k=1}^{m} -w_{kj}\mathrm{sgn}(y_{ki})$$

これをもとに，要素ごとに更新を行う.

定理 1
$f(W, H) = 0$ のとき，W, H の値は，前回の繰り返しでの値と同じである.

証明
$f(W, H) = 0$ のとき，任意の i, j について，$\mathrm{sgn}(y_{ij})$ は 0 である．これと，

$$\frac{\partial f(W,H)}{\partial w_{ij}} = \sum_{k=1}^{n} -h_{jk}\mathrm{sgn}(y_{ik})$$

$$\frac{\partial f(W,H)}{\partial h_{ij}} = \sum_{k=1}^{m} -w_{kj}\mathrm{sgn}(y_{ki})$$

より，

$$\frac{\partial f(W,H)}{\partial w_{ij}} = 0, \frac{\partial f(W,H)}{\partial h_{ij}} = 0$$

が成り立つ．よって，

$$W^{(n+1)} = W^{(n)} - s^{(n)} \frac{\partial f(W,H)}{\partial W}, H^{(n+1)} = H^{(n)} - s^{(n)} \frac{\partial f(W,H)}{\partial H}$$

より，

$$W^{(n+1)} = W^{(n)}, H^{(n+1)} = H^{(n)}$$

が成り立つ．よって繰り返し後の値が元の値と同じになっている．

(証明終)

4　計算機実験

次の計算機環境で実験を行った．

OS　Windows7

言語　C++

マシンスペック　CORE i5

コンピュータにプログラムを実装し，シミュレーションを行ったところ，繰り返し回数は，$n = 10$ 万回だと，最適解が得られないことが 50%ほどあった．しかし，$n = 100$ 万回だとほとんどの場合に最適解が得られた．よって，$n = 100$ 万とした．

ただ，一部の場合において，n の回数を増やしても，劣勾配ベクトルが 0 になってしまい，局所的解にはまり最適解が得られないことがあった．

5　まとめ

絶対値誤差を最小化するアルゴリズムを提案し，証明を行い，計算機実験で検証を行った．検証は概ね成功したが，一部において不十分な点が発見された．それについては，今後の課題とする．

参考文献

[1]　D.D.Lee and H.S.Seung, Algorithms for non-negative matrix factorization, Proc. NIPS2000, 556–562.

エネルギー制限つきエージェントによるデータ配送問題

三重野琢也 †， 山内由紀子 ††， 来嶋秀治 ††， 山下雅史 ††
† 九州大学 理学部 物理学科 †† 九州大学大学院 システム情報科学研究院

1 はじめに

バッテリー駆動のロボットやエージェントにアルゴリズムを与えて自律的に動かそうとするとき，いかにエネルギーを節約して効率よくエージェントを動かすかが重要な問題となる．エネルギー制限付きエージェントによる数直線上やグラフ上での情報収集やデータ配送の問題は，これまでに研究されている [1]．本稿ではエネルギー制限のあるエージェントによる直線上でのデータ配送問題を定義し，エージェントの最適なスケジュールを与える．配送するメッセージが 1 つでも，各エージェントの初期座標が任意であれば NP 完全であることが既に示されている [2]．本稿では全エージェントも初期座標が同一である特殊な場合を考え，最適なスケジュールを与える．また，各エージェントが初期状況で同じ座標にいる場合に 1 つのデータを配送できる最大の距離を求める $O(n \log n)$ 時間アルゴリズムを提案する．

2 問題定義

$\mathrm{A} = \{a_1, a_2, ..., a_n\}$ を，数直線上に置かれた n 台のエージェント $(n \geq 1)$ の集合とし，各 a_i に対して初期の座標 l_i とバッテリー量 q_i が与えられているものとする．バッテリー量 q_i は，そのエージェント a_i が移動可能な距離に等しい．L はデータ配送のゴールの座標である．k は L まで運ばなければならないメッセージの数であり，初期状況では全てのメッセージの座標は原点であるとする．また，エージェントはメッセージが存在する座標でしかメッセージの受け渡しができない．

定義 1．データ配送問題
入力：A, L, k
出力：k 個のメッセージをゴールまで運べるなら *YES,* そうでなければ *NO* を返す

データ配送問題においてはメッセージを運ぶ座標の最大値は L であり，L より大きい座標まで運ぶことが出来る場合も L までしか考慮しないものとする．
さらに，データを配送することのできる最大の座標を求める問題を以下のように定義する．最大配送距離問題では，L より大きい座標まで運ぶことが出来る場合も考慮する．

定義 2．最大配送距離問題
入力：A, k
出力：k 個のメッセージ全てを運ぶことのできる最大の座標を返す

本稿では簡単のため，いずれの問題も $k = 1$ の場合のみを考える．

3 $k = 1$ で全エージェントが同じ座標にいる場合の考察

3.1 データ配送問題（全エージェントが L にいるとき）

$k = 1$ の場合，各エージェントが無駄な動きをしないと仮定すると，各エージェントはメッセージを受け取るために 0 以上の距離をメッセージのある方向へ向かって動き，その後はバッテリーが無くなるかゴールに到達するまでゴール方向

へ向かって動くものとすることができる．こうすると，エージェントの動く順番を定めるとそれらのエージェントの動きは一意に定まる．ゆえに $k=1$ の場合のデータ配送のスケジュール X を，動く順番にソートされた n 台のエージェントの列（$X=(a_{X_1},a_{X_2},...,a_{X_n})$）で表すことにする．また一般に，$n$ 台のエージェント全員がメッセージの運搬に寄与するとは限らない．つまり，データ配送が途切れてしまう場合や n 台未満でゴールできる場合もある．「データ配送が途切れる」とは，メッセージが L まで到達せずメッセージが存在する座標にどのエージェントも到達できないことを言う．

補題 1. $k=1,\forall i(1\leq i\leq n),l_i=L$ のとき，スケジュール $(a_1,a_2,...,a_n)$ でデータ配送が途切れることがないとすると，n 人のエージェントでメッセージを運ぶことのできる座標 S は，$S=2^n\sum_{i=1}^{n}2^{-i}q_i-(2^n-1)L$.

Proof. データ配送が途切れないと仮定すると，n 台が動いた後のメッセージの座標 S_n について，以下の漸化式が成り立つ．

$$S_0=0,\qquad S_n=S_{n-1}+\{q_n-(L-S_{n-1})\}=2S_{n-1}+q_n-L\qquad(n\geq 1).$$

これを用いて，一般項 S_n が，$S_n=2^n\sum_{i=1}^{n}2^{-i}q_i-(2^n-1)L$ であることを帰納法で示す．$n=1$ のとき，明らかに成立する．$n=l(l\geq 1)$ で成立すると仮定すると

$$S_l=2^l\sum_{i=1}^{l}2^{-i}q_i-(2^l-1)L.$$

よって，漸化式を用いると

$$S_{l+1}=2S_l+q_{l+1}-L=2^{l+1}\sum_{i=1}^{l}2^{-i}q_i-(2^{l+1}-2)L+q_{l+1}-L=2^{l+1}\sum_{i=1}^{l}2^{-i}q_i-(2^{l+1}-1)L.$$

ゆえに、任意の n で成立する．従って，$S=S_n=2^n\sum_{i=1}^{n}2^{-i}q_i-(2^n-1)L$. □

以下，スケジュール X に対する補題 1 の S を S_X と書くことにする．

定理 1. $k=1,\forall i(1\leq i\leq n)l_i=L$ とする．メッセージを運ぶことができる距離が最大となるスケジュールは，エージェントのバッテリー量が非増加順となるスケジュールである．

Proof. まず，データ配送が途中で途切れない場合を背理法で示す．バッテリー量の非増加順で動くときがメッセージを運べる距離が最大ではない，つまり，あるスケジュール $X=(a_{X_1},a_{X_2},\cdots,a_{X_i},\cdots,a_{X_j},\cdots,a_{X_n})$ (ただし、$q_{X_i}<q_{X_j}$) が存在し，S_X がメッセージを運ぶことができる距離の最大であると仮定する．このとき、スケジュール中の a_{X_i} と a_{X_j} の位置を入れ替えたスケジュールを X' とすると，補題 1 より，

$$S_{X'}-S_X=2^n(2^{-X_i}q_{X_j}+2^{-X_j}q_{X_i}-2^{-X_i}q_{X_i}-2^{-X_j}q_{X_j})=2^n(2^{-X_i}-2^{-X_j})(q_{X_i}-q_{X_j})>0$$

となり，S_X が最大であることに矛盾する．したがって，バッテリー量の非増加順で動くときが最大である．また，スケジュール X でデータ配送が途中で途切れる場合は，途切れる直前までのエージェントについて上と同じ議論を行うことで示すことができる． □

以下では，a_i は q_i について非増加順にソートされている (つまり，$q_1\geq q_2\geq ...\geq q_n$ となっている) ものとする．また，補題 1 の式に $L=S=S_n$ を代入すると，このときの S_n は，データ配送が途切れない時にちょうどメッセージを持って戻って来ることができるスタートの座標を表し，$S_n=\sum_{i=1}^{n}2^{-i}q_i$ となる．従って，データ配送が成功するときは必ず S_n は L 以上となっている．

定理 2. $k=1,\forall i(1\leq i\leq n)l_i=L,q_1\geq q_2\geq ...\geq q_n$ とする．このときデータ配送が成功するための必要十分条件は，$L\leq\sum_{i=1}^{n}2^{-i}q_i$ である．

Proof. データ配送が成功するとき，$L\leq\sum_{i=1}^{n}2^{-i}q_i$ であることは上の議論から明らか．次に，$L\leq\sum_{i=1}^{n}2^{-i}q_i$ のときにデータ配送が成功することを示す．上の議論より，データ配送が途切れないことが示せれば証明は完了である．$n\geq 2$

のとき，$L \leq \sum_{i=1}^{n} 2^{-i} q_i$ ならばデータ配送は途切れないことを，帰納法で示す ($n = 1$ のときの成立は明らか)．証明の為，ダミーのエージェント a_0 ($q_0 = 0, l_0 = 0$) が存在し，初期状況でメッセージを持っているものとする．最初に，a_1 が必ず a_0 からメッセージを受信できることを示す．

$$\sum_{i=1}^{n} \frac{q_1}{2^i} \geq \sum_{i=1}^{n} \frac{q_i}{2^i} \geq L \quad \text{より}, \quad q_1 \sum_{i=1}^{n} \frac{1}{2^i} \geq L. \quad \text{ゆえに}, \quad q_1 \geq \frac{2^n}{2^n - 1} L \geq L.$$

よって，a_1 は必ず a_0 からメッセージを受信できる．次に，a_{l-1} が a_{l-2} からメッセージを受信できると仮定したとき，a_l は a_{l-1} からメッセージを受信できることを示す ($l \geq 2$)．

$$\sum_{i=1}^{l-1} \frac{q_i}{2^i} + \sum_{i=l}^{n} \frac{q_l}{2^i} \geq \sum_{i=1}^{n} \frac{q_i}{2^i} \geq L \quad \text{より}, \quad q_l \sum_{i=l}^{n} \frac{1}{2^i} \geq L - \sum_{i=1}^{l-1} \frac{q_i}{2^i}.$$

即ち，

$$q_l \geq \frac{2^{n-l+1}}{2^{n-l+1} - 1} \left(L - \sum_{i=1}^{l-1} \frac{q_i}{2^i} \right) \geq L - \sum_{i=1}^{l-1} \frac{q_i}{2^i}.$$

ここで，$\sum_{i=1}^{l-1} 2^{-i} q_i$ は、$l-1$ 番目までのエージェントがメッセージを運んだ座標であり，$L - \sum_{i=1}^{l-1} 2^{-i} q_i$ は a_{l-1} が動き終わった時点でのメッセージとゴールの間の距離を表す．つまり，a_l は必ず a_{l-1} からメッセージを受信することができる．従って，データ配送が途切れることは無い． □

3.2 最大配送距離問題（全エージェントが s にいるとき）

次に，各 l_i が同じ値 s の場合の最大配送距離問題について考える．ここでも a_i は q_i について非増加順にソートされているものとする．この問題は以下の 3 つに場合分けができる．

(i) $s = 0$
(ii) $s \neq 0$ かつ 最適な動きでも s より遠くには運べない
(iii) $s \neq 0$ かつ s より遠くに持っていくことができる

(i) の場合は明らかに最大のバッテリーを持つエージェントが 1 人で運ぶときが最適であり，(ii) の場合は補題 1 より、バッテリー量の非増加順に動かすときが最適である．よって以下では、(iii) の場合についてのみ考察する．

ここで，座標 s 以降でメッセージの運搬に寄与できるエージェントは高々 1 台であるので，座標 s に 2 台以上を残す場合は考えなくて良い．結局，1 台のエージェントを s に残しておき残りの $n-1$ 台でメッセージを s まで (できるだけ s の近くまで) 運ぶという戦略を考えればよいことになる．定理 1 の証明の考え方を用いると，以下の補題 2 が得られる．この補題は，後に示す **Algorithm 1** の正当性を保証する．

補題 2. $k = 1, \forall i (1 \leq i \leq n) l_i = s, q_1 \geq q_2 \geq ... \geq q_n$ とする．$A \backslash \{a_{m-1}\}$ はメッセージを s まで運べず，$A \backslash \{a_m\}$ はメッセージを s まで運べるような $m (\geq 2)$ が存在すると仮定すると，以下の 4 つの命題は真である．

1. $A \backslash \{a_i\}$ がメッセージを s まで運べないとき，$j \leq i$ なる任意の j に対して，$A \backslash \{a_j\}$ はメッセージを s まで運べない．
2. $A \backslash \{a_i\}$ がメッセージを s まで運べるとき，$j \geq i$ なる任意の j に対して，$A \backslash \{a_j\}$ はメッセージを s まで運べる．
3. $i \leq m-1$ なる任意の i に対して，a_i が最後に動くようなスケジュールでメッセージを運べる距離は a_{m-1} が最後に動くようなスケジュールでメッセージを運べる距離以下である．
4. $i \geq m$ なる任意の i に対して，a_i が最後に動くようなスケジュールでメッセージを運べる距離は a_m が最後に動くようなスケジュールでメッセージを運べる距離以下である．

Proof. a_i 以外の $n-1$ 台のエージェントをバッテリー量非増加順で動かした時にメッセージを運べる距離の最大値を L_i と書くことにする．
(1. の証明) $q_i \leq q_j$ であるので，合計バッテリー量は $A\backslash\{a_j\}$ の方が大きくなることはない．よって，$L_j \leq L_i$ となる．
(2. の証明) $q_i \geq q_j$ であるので，合計バッテリー量は $A\backslash\{a_j\}$ の方が小さくなることはない．よって，$L_j \geq L_i$ となる．
(3. の証明) $2 \leq i \leq m-1$ なる任意の i について，a_{i-1} を最後に動かすスケジュールを X，a_i を最後に動かすスケジュールを Y としたとき，$S_X \leq S_Y$ であることを示す．補題 1 より，

$$S_Y - S_X = 2^n(2^{-i+1}q_{i-1} + 2^{-n}q_i) - 2^n(2^{-i+1}q_i + 2^{-n}q_{i-1}) = (2^{n-i+1}-1)(q_{i-1}-q_i) \geq 0.$$

従って，$2 \leq i \leq m-1$ においては，$i = m-1$ のスケジュール Y' のときに運べる距離は最大となる．
(4. の証明) a_i が最後に動くようなスケジュールで n 台でメッセージを運べる距離を $L_{\max}(i)$ とすると，

$$L_{\max}(m) = \max\{s + q_m, L_m\}$$

である．よって，どちらが最大になるかによって場合分けをして考える．
$s + q_m \geq L_m$ の場合，$q_m \geq q_i$ より，

$$L_{\max}(m) = s + q_m \geq s + q_i.$$

また，2. の証明より $L_m \geq L_i$ であるので，

$$L_{\max}(m) = s + q_m \geq L_m \geq L_i.$$

よって，$L_{\max}(m) \geq L_{\max}(i)$.
$L_m \geq s + q_m$ の場合，

$$L_{\max}(m) = L_m \geq s + q_m \geq s + q_i \quad \text{かつ} \quad L_{\max}(m) = L_m \geq L_i.$$

よって，$L_{\max}(m) \geq L_{\max}(i)$. □

補題 2 から，(iii) の場合において最も遠くへ運ぶことができるスケジュールは，a_m が最後に動くようなスケジューールか、a_{m-1} が最後に動くようなスケジュールかのどちらかであることがわかる．これを用いて，最大配送距離問題を $O(n \log n)$ 時間で計算するアルゴリズムを提案する．

3.3 最大配送距離問題（全員が s にいるとき）を計算する $O(n \log n)$ 時間アルゴリズム

全員が同じ座標にいるときにどこまで遠くに運べるかを $O(n \log n)$ 時間で計算できるアルゴリズムを提案する．まず，エージェントのバッテリー量非増加順のソートは $O(n \log n)$ 時間でできる．また補題 2 より，もし $A\backslash\{a_i\}$ がメッセージを s まで運べるならば，i より大きな各 j に対して a_j を最後に動かすスケジュールで運べる距離は必ず a_i を最後に動かすスケジュールで運べる距離以下であることが言える．また，もし $A\backslash\{a_i\}$ がメッセージを s まで運べないならば，i 未満の各 j に対して a_j を最後に動かすスケジュールで運べる距離は必ず a_i を最後に動かすスケジュールで運べる距離以下であることが言える．つまり s まで戻れるか戻れないかを一回比較する度に探索範囲を半分にできるため，二分探索を行える．そうすると最後の添字は必ず $m-1$ か m のどちらかになっているため，それらの比較を行えばよい．また，$q_0 = q_{n+1} = 0$ とする．

以下のアルゴリズムは，$O(n \log n)$ 時間で配送距離の最大値を求められる．よって，以下の定理が言える．

定理 3. $k = 1$ で全てのエージェントの初期座標が同じとき，メッセージを運ぶことができる最大の座標を $O(n \log n)$ 時間で求めることができる．

Algorithm 1 最大配送距離問題を解くアルゴリズム：$DataDel2(A, s)$

$b \leftarrow 1$
$e \leftarrow n$
while $b \leq e$ **do**
  $m \leftarrow \lfloor \frac{b+e}{2} \rfloor$
  if $f(A \backslash \{a_m\}, s) =$ "YES" **then**
    $e \leftarrow m - 1$
  else
    $b \leftarrow m + 1$
  end if
end while
return $\max\{g(A \backslash \{a_{m-1}\}, s, q_{m-1}), g(A \backslash \{a_m\}, s, q_m), g(A \backslash \{a_{m+1}\}, s, q_{m+1})\}$

Algorithm 2 A が s までメッセージを運べるかを判定するアルゴリズム：$f(A, s)$

$n \leftarrow |A|$
$S \leftarrow \sum_{i=1}^{n} 2^{-k} q_k$
if $S \geq s$ **then**
  return "YES"
else
  return "NO"
end if

Algorithm 3 バッテリー q を持つエージェントが座標 s に残るスケジュールで運べる距離を求めるアルゴリズム：$g(A, s, q)$

$n \leftarrow |A|$
$S_0 = 0$
for $j \leftarrow 1$ **to** n **do**
  $S_j = 2S_{j-1} + q_j - s$
  if $S_j < S_{j-1}$ **then**
    $u \leftarrow 2S_{j-1} + q - s$
    return $\max\{u, S_{j-1}\}$
  else if $S_j \geq s$ **then**
    return $\max\{S_j, s + q\}$
  end if
end for
$u \leftarrow 2S_{n-1} + q - s$
return $\max\{u, S_{n-1}\}$

4 まとめ

$k = 1$ で全てのエージェントがゴールにいるときの最適スケジュールと，$k = 1$ で全てのエージェントが同じ座標にいるときにどこまで遠くに運べるかを $O(n \log n)$ で判定するアルゴリズムを示した．今後は $k = 2$，$k = 3$ および一般の k の場合での最適なスケジュールの計算を課題としたい．

参考文献

[1] Julian Anaya, Jérémie Chalopin, Jurek Czyzowicz, Arnaud Labourel, Andrzej, and Yann Vaxès. "Collecting Information by Power-Aware Mobile Agents," In Proceedings of the 26th International Symposium on Distributed Computing 2012, pp.46–60, 2012.

[2] Jérémie Chalopin, Riko Jacob, Matúš Mihalák, and Peter Widmayer. "Data Delivery by Energy-Constrained Mobile Agents on a Line," In Proceedings of the 41st International Colloquium on Automata, Languages, and Programming, pp. 423–434 2014.

Pebble Coveringの数え上げ高速化

玉谷賢一† 山内由紀子‡ 来嶋秀治‡ 山下雅史‡
†九州大学工学部電気情報工学科
‡九州大学院システム情報科学研究院

1 はじめに

平面内で剛なグラフのうち辺数が極小のグラフのことをLamanグラフという．グラフがLamanであることの必要充分条件は辺数が$2n-3$かつ辺集合が独立であることである[1, 2]．Pebble Gameというアルゴリズムによってグラフの辺集合が独立であるかどうかを判定することが出来る[1].

Pebble Gameでは，あるグラフについてPebble Coveringが存在するかを判定する．独立な辺集合に新たな辺を追加する際，その辺を4重辺にしたときのPebble Coveringが存在するならばその辺はもとの辺集合で独立である．これをグラフの全ての頂点で調べることでグラフの辺集合が独立であるかどうかを判定出来る[1].

本稿ではPebble Coveringの数え上げの高速化を示す.

2 Pebble Coveringの数え上げ

2.1 Pebble Covering

$G=(V,E)$は無向グラフとする．初期状態では各頂点が2つのpebbleをもっているものとして，以下の条件を満たすpebbleの割当をPebble Coveringという[1].

- 各頂点は現在持っているpebbleを接続している辺に置くことが出来る.
- 全ての辺の上にpebbleがある.

この条件はGの辺Eに各頂点の出次数が高々2となる向き付けに等しい．一般にグラフGに対して複数のPebble Coveringが存在する．また，$|E|<2|V|$の場合はpebbleが全ては使われない（Pebble Coveringの状態で頂点にpebbleが残る）場合がある[1].

本稿では以下，頂点数n，辺数$2n$のPebble Coveringの全通りの数え上げについて議論する．すなわち辺のpebbleを置いている頂点から反対側への向き付けを考えると，この数え上げは出次数2の単純有向グラフの数え上げである．頂点にはラベルがあり，区別されるものとする.

2.2 数え上げアルゴリズム

2.2.1 素朴なアルゴリズム（虱潰し）

Algorithm 1 素朴なアルゴリズム

1: 頂点数nの完全グラフの辺数は$\binom{n}{2}=n(n-1)/2$通り．このうち使われる辺は$2n$本であるから辺の選び方は$\binom{n(n-1)/2}{2n}$通り.
2: この各選び方について辺の向き付けは2^{2n}通り.
3: 各向き付けについて出次数が2であるならば1数え上げる.

2.2.2 分枝限定法

素朴なアルゴリズムは 2^{2n} 通りの向き付け全てを試すことや，Pebble Covering が存在しない辺の選び方についても同様に全通りの向き付けを試すため効率がよくない．

素朴なアルゴリズムでは辺数が $2n$ であるという条件に注目していたが，分枝限定法では各頂点の出次数が 2 であるという条件に注目する．

Algorithm 2 分枝限定法

1: (初期条件) 全ての頂点を孤立点とする．
2: 任意の順に頂点を選ぶ．
3: 選んだ頂点から有向辺を引ける頂点（この頂点とまだつながっていない頂点）の個数を a とする．
4: $a \leq 1$ のとき，この頂点の次数を 2 にすることができないため失敗．
5: $a \geq 2$ のとき，外向辺の引き方は $\binom{a}{2}$ 通りあり，このそれぞれに分岐し，次の頂点について 3 から繰り返す．
6: 最後の頂点の外向辺を引いたら 1 数え上げる．

2.3 分枝限定法の高速化

後の表に示す通り，この分枝限定アルゴリズムは素朴なアルゴリズムに比べて若干の改善にはなっているが，$n \geq 10$ ではどちらの手法でも現実的な時間では解けなかった．そこで，分枝限定法の高速化について考える．アイデアは分岐の際にそれ以降の分岐の形が同じになるものはまとめて数え上げするというものである．

2.3.1 高速化のアイデア (1)

分枝限定法のステップ 3 で孤立点が二つ以上ある場合，次のような選び方はまとめて数え上げできる．

1. 孤立点を 1 つ，孤立点でない頂点から 1 つ選ぶ場合．

2. 孤立点を 2 つ選ぶ場合．

1 のとき孤立点以外の頂点の方を固定すると，どの孤立点を選んでもグラフが同型になる．2 も同様にどの 2 頂点を選んできてもグラフが同型になる．

同型のグラフはそれ以降の分岐の形が同じになるような順で頂点を選ぶことができるため，まとめて数え上げができる．

2.3.2 高速化のアイデア (2)

分枝限定法のステップ 3 で出次数 2 の頂点が 2 つ以上ある場合，次のような選び方はまとめて数え上げできる．

1. 出次数 2 の頂点を 1 つ，出次数 0 の頂点から 1 つ選ぶ場合

2. 出次数 2 の頂点を 2 つ選ぶ場合

1 のとき出次数 0 の頂点の方を固定すると，どの出次数 2 の頂点を選んでもこれ以降この辺を参照する操作は出てこない（分枝限定法では出次数 2 の頂点の間の辺を参照するような操作はない）ため，これ以降の分岐の形が同じになりまとめて数え上げができる．2 のときも同様である．

2.3.3 高速化アルゴリズム

Algorithm 3 高速化アルゴリズム

1: (初期条件) 全ての頂点を孤立点とする.
2: 任意の順に頂点を選ぶ.
3: 選んだ頂点から有向辺を引ける頂点（この頂点とまだつながっていない頂点）のうち孤立点を a 個、出次数 2 の頂点を b 個、それ以外の頂点を c 個とする.
4: $a+b+c \leq 1$ のとき，この頂点の次数を 2 にすることができないため失敗.
5: $a+b+c \geq 2$ のとき，外向辺の引き方は次のように場合分けできる.

- a の頂点から 2 つ選ぶ場合:$\binom{a}{2}$ 通りの選び方のうち 1 通りを計算し，出てきた Pebble Covering の個数を $\binom{a}{2}$ 倍する.
- b の頂点から 2 つ選ぶ場合:$\binom{b}{2}$ 通りの選び方のうち 1 通りを計算し，出てきた Pebble Covering の個数を $\binom{b}{2}$ 倍する.
- c の頂点から 2 つ選ぶ場合:まとめての数え上げはできないので $\binom{c}{2}$ 通り全てに分岐する.
- a の頂点から 1 つ，b の頂点から 1 つ選ぶ場合:ab 通りの選び方のうち 1 通りを計算し，出てきた Pebble Covering の個数を ab 倍する.
- a の頂点から 1 つ，c の頂点から 1 つ選ぶ場合:c の頂点それぞれについて分岐し，a の頂点は a 通りの選び方のうち 1 通りを計算し，出てきた Pebble Covering の個数を a 倍する.
- b の頂点から 1 つ，c の頂点から 1 つ選ぶ場合:c の頂点それぞれについて分岐し，b の頂点は b 通りの選び方のうち 1 通りを計算し，出てきた Pebble Covering の個数を b 倍する.

6: 最後の頂点の外向辺を引いたら 1 数え上げる.

3 計算機実験

実行環境　CPU:2.4GHz Intel Core i5/RAM:8 GB(DDR3)/OS:OS X 10.9.4
実行時間は三回の平均値（ただし*は 1 度の実行の値）．計測は $clock()$ 関数を使用．有効桁数 2.

n	Pebble Covering	1	2	3
5	24	0.000006[sec]	0.000026[sec]	0.000013[sec]
6	14490	0.0034[sec]	0.0015[sec]	0.000064[sec]
7	4590360	1.7[sec]	0.21[sec]	0.00042[sec]
8	1436893710	800[sec]	45[sec]	0.0028[sec]
9	502451412120		13000[sec]*	0.021[sec]
10	203034577143636			0.16[sec]
11	95515499986847640			1.4[sec]
12	52206557635168560360			12[sec]
13	32969338228517009014080			110[sec]
14	23894519022720589913502690			1100[sec]
15	19736460638800406069301676944			12000[sec]*

4　まとめ

実際に高速化によって素朴なアルゴリズムでは現実的な時間では計算できなかったところまで数え上げをすることができた．今後の課題は，同様の改良を用いて Laman グラフの数え上げの高速化を実現することである．

5　参考文献

参考文献

[1] D. J. Jacobs and B. Hendrickson, An algorithm for two-dimensional rigidity percolation: the pebble game, Jounal of Computational Physics, 137(1997), 346–365.

[2] 加藤直樹 三角形分割とラーマングラフ:建築への応用, RAMP シンポジウム 2006, 135–152.

Random Address Permute-Shift Technique for the Shared Memory on GPUs

Koji Nakano*, Susumu Matsumae†, and Yasuaki Ito*

*Department of Information Engineering
Hiroshima University
Kagamiyama 1-4-1, Higashi Hiroshima, 739-8527 Japan

†Department of Information Science
Saga University
Honjo 1, Saga, 840-8502 Japan

***Abstract*—The Discrete Memory Machine (DMM) is a theoretical parallel computing model that captures the essence of memory access to the shared memory of a streaming multiprocessor on CUDA-enabled GPUs. The DMM has w memory banks that constitute a shared memory, and w threads in a warp try to access them at the same time. However, memory access requests destined for the same memory bank are processed sequentially. Hence, it is very important for developing efficient algorithms to reduce the memory access congestion, the maximum number of memory access requests destined for the same bank. The main contribution of this paper is to present a novel algorithmic technique called the random address permute-shift (RAP) technique that reduces the memory access congestion. We show that the RAP reduces the memory access congestion to $\tilde{O}(\frac{\log w}{\log\log w})$ for any memory access requests including malicious ones by a warp of w threads. Also, we can guarantee that the congestion is 1 both for contiguous access and for stride access. The simulation results for $w = 32$ show that the expected congestion for any memory access is only 3.53. Since the malicious memory access requests destined for the same bank take congestion 32, our RAP technique substantially reduces the memory access congestion. We have also applied the RAP technique to matrix transpose algorithms. The experimental results on GeForce GTX TITAN show that the RAP technique is practical and can accelerate a direct matrix transpose algorithm by a factor of 10.**

***Keywords*-GPU, CUDA, memory bank conflicts, memory access congestion, randomized technique**

I. Introduction

The GPU (Graphics Processing Unit), is a specialized circuit designed to accelerate computation for building and manipulating images [1], [2], [3]. Latest GPUs are designed for general purpose computing and can perform computation in applications traditionally handled by the CPU. Hence, GPUs have recently attracted the attention of many application developers. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [4], the computing engine for NVIDIA GPUs. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in NVIDIA GPUs.

NVIDIA GPUs have streaming multiprocessors (SMs) each of which executes multiple threads in parallel. CUDA uses two types of memories in the NVIDIA GPUs: *the shared memory* and *the global memory* [4]. Each SM has the shared memory, an extremely fast on-chip memory with lower capacity, say, 16-48 Kbytes, and low latency. Every SM shares the global memory implemented as an off-chip DRAM, and has large capacity, say, 1.5-6 Gbytes, but its access latency is very long. The efficient usage of the shared memory and the global memory is a key for CUDA developers to accelerate applications using GPUs. In particular, we need to consider *bank conflicts* of the shared memory access and *coalescing* of the global memory access [5]. The address space of the shared memory is mapped into several physical memory banks. If two or more threads access the same memory bank at the same time, the access requests are processed in turn. Hence, to maximize the memory access performance, threads in a warp should access distinct memory banks to avoid the bank conflicts of the shared memory accesses. To maximize the bandwidth between the GPU and the DRAM chips, the consecutive addresses of the global memory must be accessed at the same time. Thus, CUDA threads should perform coalesced access when they access the global memory.

In our previous paper [6], we have introduced two models, *the Discrete Memory Machine (DMM)* and *the Unified Memory Machine (UMM)*, which reflect the essential features of the shared memory and the global memory of CUDA-enabled GPUs. Since the DMM and the UMM are promising as theoretical computing models for GPUs, we have published several efficient algorithms on the DMM [7], [8] and the UMM [9]. The DMM and the UMM have three parameters: the number p of threads, width w, and memory access latency l. Figure 1 illustrates the outline of the architectures of the DMM and the UMM with $p = 20$ threads and width $w = 4$. Each thread works as a Random Access Machine (RAM) [10], which can execute fundamental operations in a time unit. Threads are executed in SIMD [11] fashion, and they run on the same program

and work on different data. The p threads are partitioned into $\frac{p}{w}$ groups of w threads each called *warp*. The $\frac{p}{w}$ warps are dispatched for memory access in turn, and w threads in a dispatched warp send memory access requests to the memory banks (MBs) through the memory management unit (MMU). We do not discuss the architecture of the MMU, but we can think that it is a multistage interconnection network [12] in which memory access requests are moved to destination memory banks in a pipeline fashion. Note that the DMM and the UMM with width w has w memory banks and each warp has w threads. For example, the DMM and the UMM in Figure 1 have 4 threads in each warp and 4 MBs.

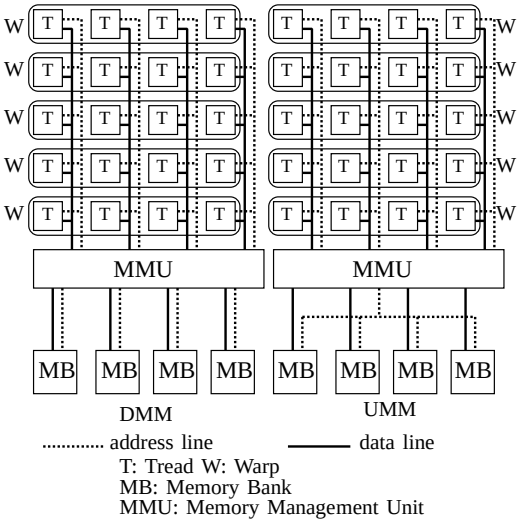


Figure 1. The architectures of the DMM and the UMM with width $w = 4$

MBs constitute a single address space of the memory. A single address space of the memory is mapped to the MBs in an interleaved way such that the word of data of address i is stored in the $(i \bmod w)$-th bank, where w is the number of MBs. The main difference of the two architectures is the connection of the address line between the MMU and the MBs, which can transfer an address value. In the DMM, the address lines connect the MBs and the MMU separately, while a single set of address lines from the MMU is connected to the MBs in the UMM. Hence, in the UMM, the same address value is broadcast to every MB, and the same address of the MBs can be accessed in each time unit. On the other hand, different addresses of the MBs can be accessed in the DMM. Since the memory access of the UMM is more restricted than that of the DMM, the UMM is less powerful than the DMM. Also, we assume that MBs are accessed in a pipeline fashion with latency l. In other words, if a thread sends a memory access request, it takes at least l time units to complete it. A thread can send a new memory access request only after the completion of the previous memory access request and thus, it can send at most one memory access request in l time units.

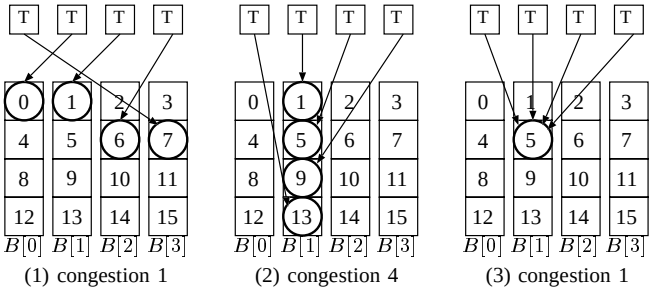


Figure 2. Examples of memory access and the congestion for $w = 4$

It is very important for developing efficient algorithms on the DMM to reduce *the memory access congestion*, the maximum number of unique memory access requests by a warp destined for the same bank. The memory access congestion takes value between 1 and w. The reader should refer to Figure 2 showing examples of the memory access and the congestion. If w threads send memory access requests to distinct banks, the congestion is 1 and the memory access is conflict-free. If all memory access requests are destined to the same bank, the congestion is w. It is not easy and sometimes impossible to minimize the memory access congestion for some problems. For example, a straightforward matrix transpose algorithm that reads a matrix in row major order and writes in column major order involves memory access with congestion w. On the other hand, by an ingenious memory access technique, we can transpose a matrix with congestion 1 [6]. Further, in our previous paper [6], we have developed a complicated graph coloring technique to eliminate bank conflicts in off-line permutation. We have implemented this offline permutation algorithm on GeForce GTX-680 GPU [13]. The experimental results showed that the offline permutation algorithm developed for the DMM runs on the GPU much faster than the conventional offline permutation algorithm [13]. Although it is very important to minimize the memory access congestion, it may be a very hard task.

In latest CUDA-enabled GPUs such as GeForce GTX TITAN, the number w of memory banks and threads in a warp is 32, and the size of a shared memory is no more than 48Kbytes [4]. Hence, a matrix with $w \times w$ double (64-bit) numbers in such CUDA-enabled GPUs occupies 8Kbytes and it is not possible to store more than 6 matrices of size $w \times w$ in a shared memory. Thus, many algorithms designed for CUDA-enabled GPUs use one or several matrices of size $w \times w$ in the shared memory [1], [4], [14]. For example, paper [14] has presented an optimal offline permutation algorithm for the global memory. This optimal algorithm repeats offline permutation for 32×32 matrices in the shared

memory of each streaming multiprocessor in a GPU. Also, an efficient matrix multiplication for a large matrix in the global memory repeats multiplication of 32×32 submatrices in the shared memory [4]. Hence, it makes sense to focus on a matrix of size $w \times w$. Usually, each (i, j) element $(0 \leq i, j \leq w-1)$ of a matrix of size $w \times w$ is mapped to address $i \cdot w + j$ in a conventional implementation. We call such a straightforward implementation, *RAW (RAW access to memory) implementation*. In the RAW implementation, the congestion of stride access is w, while that of contiguous access is 1. Hence, CUDA developers should implement algorithms in GPUs so that it never performs stride access to the shared memory.

The main contribution of this paper is to present sophisticated algorithmic technique called *the random address permute-shift (RAP)*, which reduces the memory access congestion for any memory access to a matrix of size $w \times w$ by a warp of w threads. Let r be a random permutation of $(0, 1, \ldots, w-1)$ uniformly selected at random from all possible $n!$ permutations. In other words, w integers $r_0, r_1, \ldots, r_{w-1}$ are distinct in the range $[0, w-1]$. By the RAP technique, each (i, j) element $(0 \leq i, j \leq w-1)$ of a matrix is mapped to address $i \cdot w + ((j + r_i) \bmod w)$ and thus, it is in memory bank $(j + r_i) \bmod w$. Our first contribution is to show that, by the RAP technique, it is guaranteed that:

- any contiguous access and any stride access has no bank conflict, and
- the congestion is at most $\tilde{O}(\frac{\log w}{\log \log w})$ for any memory access including malicious ones by a warp of w threads, where $\tilde{O}(f)$ denotes expected $O(f)$.

Quite recently, we have presented an algorithmic technique called *the random address shift (RAS)* to reduce the memory access congestion on the DMM [7], [15]. Basically, the random address shift technique is inspired by parallel hashing that averages the access to memory modules [16], [17]. The idea is to arrange address $i \cdot w + j$ in bank $(j + r_i) \bmod w$ for independent random numbers $r_0, r_1, \ldots$ computed beforehand. However, the RAS implementation involves bank conflicts for stride memory access. On the other hand, our new RAP implementation has no bank conflict for stride memory access and the congestion is 1. Table I summarizes the memory access congestion by the RAW, the RAS, and the RAP implementations.

Table I
THE MEMORY ACCESS CONGESTION OF THE RAW, THE RAS, AND THE RAP

	RAW	RAS	RAP
Any	$[1, w]$	$\tilde{O}(\frac{\log w}{\log \log w})$	$\tilde{O}(\frac{\log w}{\log \log w})$
Contiguous	1	1	1
Stride	32	$\tilde{O}(\frac{\log w}{\log \log w})$	1

The second contribution is to show simulation results of memory access by the RAW, the RAS and the RAP. Our simulation results show that the congestions of the RAW, the RAS and the RAP are the same for random memory access. By the RAP, contiguous and stride memory access operations have no bank conflict. Also, when $w = 32$, the congestion of the RAP for a stride memory access is always 1, while the congestions of RAW and the RAS are 32 and 3.53, respectively. Hence, the RAP is much more efficient for the stride memory access.

The third contribution is to implement the RAP technique in a streaming multiprocessor on GeForce GTX TITAN [18] which supports CUDA Compute Capability 3.5 [4]. In particular, we have implemented three matrix transpose algorithms, Contiguous Read Stride Write (CRSW), Stride Read Contiguous Write (SRCW), and Diagonal Read Diagonal Write (DRDW). The CRSW and the SRCW follow the definition of a matrix transpose. More specifically, in the CRSW, a matrix is read in row major order and is written in column major order to transpose it. The SRCW reads a matrix in column major order and writes in row major order. Since memory access requests in column major order are destined for the same bank, these algorithms take a lot of time. The DRDW is optimized for the RAW implementation and performs reading and writing in diagonal order to reduce the memory access congestion to 1. Thus, the DRDW runs much faster than the others in the RAW implementation. However, it may not be easy for CUDA developers to find an efficient algorithm such as the DRDW for complicated problems. The implementation results of CRSW and SRCW algorithms for a 32×32 matrix in the shared memory show that the RAP implementation is much faster than the others. More specifically, the RAP runs only 154.5ns, while the RAW and the RAS run 1595ns and 303.6ns for CRSW algorithm, respectively.

We also present several methods to extend the RAP for arrays larger than w^2. The RAP for larger arrays has fewer bank conflicts using fewer random numbers than the RAS.

From the theoretical analysis, the simulation results, and the implementation results shown in this paper, we can say that the RAP is a potent method to reduce memory access congestion and bank conflicts that spoil high computing power of the GPUs. It is not necessary for CUDA developers to avoid bank conflicts if they use the RAP. The memory access congestion can be automatically reduced by the RAP even if it involves a lot of bank conflicts. Further, it will be a nice idea to implement the RAP technique as embedded hardware in future GPUs. More specifically, a circuit that evaluates $j \cdot w + (k + r_j) \bmod w$ for address conversion by the RAP can be embedded. Using such hardware support, the overhead of address conversion by the RAP can be negligible.

This paper is organized as follows. In Section II, we first define the DMM. Section III introduces fundamental memory access operations and matrix transpose algorithms which

are used to evaluate the performance. In Section IV, we present the random address permute-shift (RAP) technique, and evaluate the memory access congestion by theoretical analysis. Section V shows simulation results to evaluate the actual values of the congestion by the RAW, the RAS, and the RAP. In Section VI, we show experimental results on GeForce GTX TITAN. Section VII introduces several ideas to extend the RAP for larger arrays. Section VIII concludes our work.

II. Discrete Memory Machine (DMM)

The main purpose of this section is to define the Discrete Memory Machine (DMM) introduced in our previous paper [6]. The reader should refer to [6] for the details of the DMM.

Let $m[i]$ ($i \geq 0$) denote a memory cell of address i in the memory. Let $B[j] = \{m[j], m[j+w], m[j+2w], m[j+3w], \ldots\}$ ($0 \leq j \leq w-1$) denote *the j-th bank* of the memory. Clearly, a memory cell $m[i]$ is in the $(i \bmod w)$-th memory bank. We assume that memory cells in different banks can be accessed in a time unit, but no two memory cells in the same bank can be accessed in a time unit. Also, we assume that l time units are necessary to complete an access request and continuous requests are processed in a pipeline fashion through the MMU. Thus, it takes $k+l-1$ time units to complete k access requests to a particular bank.

Let $T(0), T(1), \ldots, T(p-1)$ be p threads. We assume that p threads are partitioned into $\frac{p}{w}$ groups of w threads called *warps*. More specifically, p threads are partitioned into $\frac{p}{w}$ warps $W(0), W(1), \ldots, W(\frac{p}{w}-1)$ such that $W(i) = \{T(i \cdot w), T(i \cdot w+1), \ldots, T((i+1) \cdot w-1)\}$ ($0 \leq i \leq \frac{p}{w}-1$). Warps are dispatched for memory access in turn, and w threads in a warp try to access the memory at the same time. In other words, $W(0), W(1), \ldots, W(\frac{p}{w}-1)$ are dispatched in a round-robin manner if at least one thread in a warp requests memory access. If no thread in a warp needs memory access, such warp is not dispatched for memory access. When $W(i)$ is dispatched, w threads in $W(i)$ send memory access requests, one request per thread, to the memory. Threads are executed in SIMD [11] fashion, and all thread must execute the same instruction. Hence, if one of them sends a memory read request, none of the others can send memory write request. We also assume that a thread cannot send a new memory access request until the previous memory access request is completed. Hence, if a thread send a memory access request, it must wait l time units to send a new one.

Figure 3 shows an example of memory access on the DMM with w ($= 4$) memory banks and memory access latency of l ($= 5$). We assume that each memory access request is completed when it reaches the last pipeline stage. Two warps $W(0)$ and $W(1)$ access to $\langle m[7], m[5], m[15], m[0]\rangle$ and $\langle m[10], m[11], m[12], m[9]\rangle$, respectively. In the DMM, memory access requests by $W(0)$ are separated into two pipeline stages, because $m[7]$ and $m[15]$ are in the same bank $B[3]$. Those by $W(1)$ occupy 1 stage, because all requests are destined for distinct banks, one request for each bank. Thus, the memory requests occupy three stages, and it takes $3+5-1=7$ time units to complete the memory access.

Let us define *the congestion* of memory access by a warp of w threads. Suppose that w threads in a warp access the memory banks. The memory access congestion is the maximum number of unique memory access requests destined for the same bank. We assume that, if two or more threads access the same address, the memory access requests are merged and processed as a single request. Thus, if all w threads in a warp access the same address, the congestion is 1. We also assume that if multiple memory writing requests are sent to the same address, one of them is arbitrary selected and its writing operation is performed. The other writing requests are ignored. Thus, the DMM works as the Concurrent Read Concurrent Write (CRCW) mode with arbitrary resolution of simultaneous writing [19]. For example, the congestion of memory access in Figure 2 (1) is 1, because all requests are destined for distinct banks. In Figure 2 (2), the congestion is 4, because all requests are destined for bank $B[1]$. In Figure 2 (3), all threads access $m[5]$. Thus, these memory requests are merged into one and the congestion is 1.

III. Fundamental memory access operations and matrix transpose algorithms

The main purpose of this section is to show three fundamental memory access operations for a matrix, *contiguous access*, *stride access* and *diagonal access* [6]. We also show three transposing algorithms of a matrix using these three memory access operations.

Suppose that we have a matrix a of size $w \times w$ in the memory of the DMM. We assume that $a[i][j]$ ($0 \leq i, j \leq w-1$) is arranged in address $i \cdot w + j$. Since $(i \cdot w + j) \bmod w = j$, each $a[i][j]$ is in bank $B[j]$. In these memory access operations, each element in a matrix is accessed by a thread. In the contiguous access, threads are assigned to the matrix in row-major order. Threads are assigned to the matrix in column-major order in the stride access. In the diagonal access, threads are assigned in diagonal order. The readers should refer to Figure 4 for illustrating these three memory access operations for $w = 4$.

More formally, these three memory access operations can be written as follows:

[Contiguous Access]
for $i \leftarrow 0$ to $w-1$ do in parallel
  for $j \leftarrow 0$ to $w-1$ do in parallel
    thread $T(i \cdot w + j)$ accesses $a[i][j]$

[Stride Access]
for $i \leftarrow 0$ to $w-1$ do in parallel

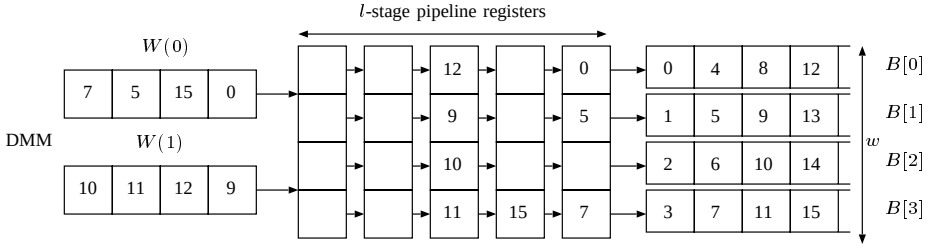


Figure 3. The Discrete Memory Machine (DMM)

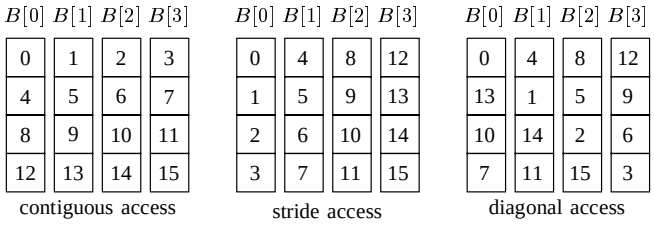


Figure 4. The contiguous access, the stride access, and the diagonal access for $w = 4$

for $j \leftarrow 0$ to $w-1$ do in parallel
  thread $T(i \cdot w + j)$ accesses $a[j][i]$

[Diagonal Access]
for $i \leftarrow 0$ to $w-1$ do in parallel
 for $j \leftarrow 0$ to $w-1$ do in parallel
  thread $T(i \cdot w + j)$ accesses $a[j][(i+j) \bmod w]$
   (or $a[(i+j) \bmod w][j]$)

It should be clear that the congestion of the contiguous access and the diagonal access is 1. On the other hand, in the stride access, w threads in a warp access distinct addresses in the same bank, and the congestion is w. In the contiguous access, w warps send memory access requests in w time units. Thus, it takes $w + l - 1$ time units to complete the contiguous access. In the stride access, w memory access requests sent by a warp occupy w pipeline stages. Hence, it takes $w^2 + l - 1$ time units to complete the stride access. Since the congestion of the diagonal access is 1, the diagonal access takes $w + l - 1$ time units similarly to the contiguous access.

We can design three matrix transpose algorithms, Contiguous Read Stride Write (CRSW), Stride Read Contiguous Write (SRCW), and Diagonal Read Diagonal Write (DRDW), using these three memory access operations. In the CRSW, a matrix is read in row major order and is written in column major order. In other words, the CRSW performs the contiguous read and the stride write for matrix transpose. Similarly, the SRCW performs the stride read and the contiguous write. In the DRDW, a matrix is read and written in diagonal order. The reader should refer to Figure 5 illustrating the three matrix transpose algorithms. The details of the three matrix transpose algorithms are spelled out as follows:

[Contiguous Read Stride Write (CRSW)]
for $i \leftarrow 0$ to $w-1$ do in parallel
 for $j \leftarrow 0$ to $w-1$ do in parallel
  thread $T(i \cdot w + j)$ performs $a[j][i] \leftarrow a[i][j]$

[Stride Read Contiguous Write (SRCW)]
for $i \leftarrow 0$ to $w-1$ do in parallel
 for $j \leftarrow 0$ to $w-1$ do in parallel
  thread $T(i \cdot w + j)$ performs $a[i][j] \leftarrow a[j][i]$

[Diagonal Read Diagonal Write (DRDW)]
for $i \leftarrow 0$ to $w-1$ do in parallel
 for $j \leftarrow 0$ to $w-1$ do in parallel
  thread $T(i \cdot w + j)$ performs
   $a[j][(i+j) \bmod w] \leftarrow a[(i+j) \bmod w][j]$

Let us evaluate the computing time of three transpose algorithms on the DMM. The CRSW transpose and the SRCW transpose involve the stride memory access. Thus, they take $O(w^2 + l)$ time units. The DRDW transpose performs diagonal read/write, it takes $O(w + l)$ time units. Hence, we have,

Lemma 1: The CRSW, the SRCW, and the DRDW transpose algorithms for a matrix of size $w \times w$ take $O(w^2 + l)$ time units, $O(w^2 + l)$ time units, and $O(w + l)$ time units, respectively, using w^2 threads on the DMM with width w and latency l.

We can implement these algorithms in a streaming multiprocessor of a GPU without any modification. We call such implementations *RAW (RAW access to memory) implementations*.

For example, the RAW implementation of the CRSW transpose algorithm for a matrix of size 32×32 is described as follows:

[The RAW implementation of the CRSW]

```
__shared__ double a[32][32],b[32][32];
int i = threadIdx.x/32;
int j = threadIdx.x%32;
double c;
b[j][i] = a[i][j];
```

We assume that matrices a and b allocated in the shared

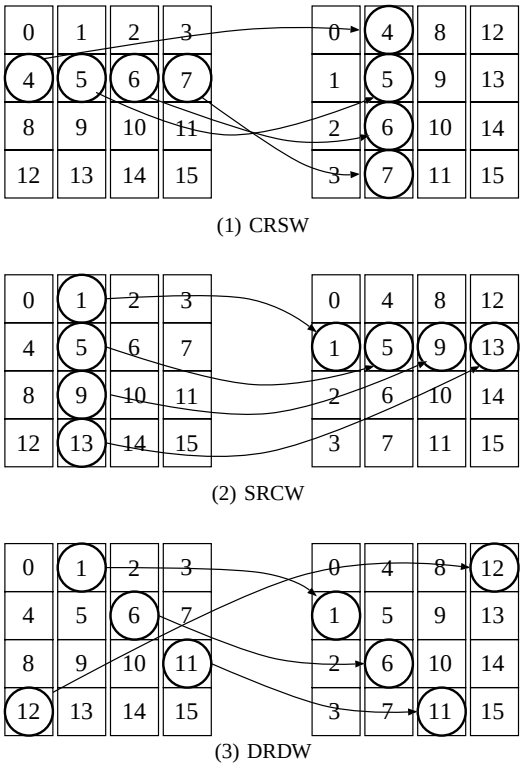


Figure 5. Illustrating the three matrix transpose algorithms for $w = 4$

memory stores the values of a matrix. In the RAW implementation, a CUDA block with 1024 threads are invoked. The value of "threadIdx.x" is a thread ID and takes value from 0 to 1023. The value of $a[i][j]$ is copied $b[j][i]$ by a thread with thread ID $i \cdot 32 + j$.

IV. The random address permute-shift (RAP) technique

The main purpose of this section is to present a novel technique, *the random address permute-shift (RAP)*, in which the memory access congestion for stride access is reduced to 1. Further, the memory access congestion by the RAP is still $\tilde{O}(\frac{\log w}{\log \log w})$ for any memory access by a warp of w threads.

Let a be a matrix of size $w \times w$ on the DMM. Note that each $a[i][j]$ is in bank $B[j]$ of the DMM. The key idea of the RAP is to use a random permutation of $(0, 1, \ldots w - 1)$. Suppose that each of w threads in a warp accesses an element of a at the same time. If all w elements are in distinct banks, the congestion is 1. On the other hand, the congestion is w if they are in the same bank. We will show that, using the RAP, the expected value of the congestion is at most $O(\frac{\log w}{\log \log w})$ for any memory access by w threads including malicious ones.

Let r be a permutation of $(0, 1, \ldots, w-1)$ selected from all possible $w!$ permutations uniformly at random. Hence, $r_0, r_1, \ldots, r_{w-1}$ take distinct integers in the range $[0, w - 1]$. Intuitively, the random address permute-shift technique rotates each i-th row ($0 \leq i \leq w-1$) of matrix a by r_i. In other words, each $a[i][j]$ ($0 \leq i, j \leq w$) is mapped to $a[i][(j + r_i) \bmod w]$. If a thread try to access $a[i][j]$, it accesses $a[i][(j + r_i) \bmod w]$ instead. Hence, $a[i][j]$ is arranged in bank $B[(j + r_i) \bmod w]$ of the DMM. Figure 6 illustrates an example of the RAP for $w = 4$, where we select $r = (2, 0, 3, 1)$. For example, $a[10] (= a[2][2])$ is mapped to $a[9] (= a[2][(2 + 3) \bmod 4])$ in $B[1]$.

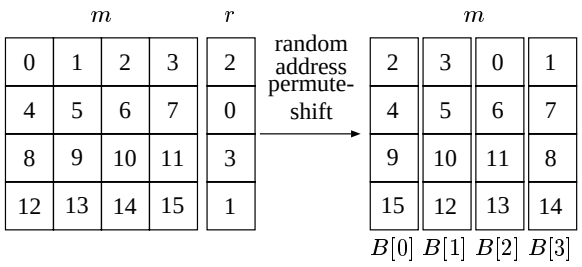


Figure 6. An example of the random address permute-shift

Recall that a memory access by a warp is *contiguous* if all w threads in a warp access the same row, and it is *stride* if all threads in a warp access the same column. Clearly, the congestion of the contiguous access is always 1, because $(0+r_i) \bmod w$, $(1+r_i) \bmod w$, ..., $(w-1+r_i) \bmod w$ are distinct. Also, that of the stride is 1, because $(j+r_0) \bmod w$, $(j + r_1) \bmod w$, ..., $(j + r_{w-1}) \bmod w$ are distinct. In our previous paper [7], we have presented the random address shift (RAS) technique, which uses independent random numbers $r_0, r_1, \ldots$ instead of a random permutation used by the RAP. Clearly, the stride access by the RAS involves bank conflicts with high probability, while that of the RAP is always 1.

We will show that, by the RAP, the congestion of the row-wise access and the column-wise access is 1. Further, the congestion of any memory access is $\tilde{O}(\frac{\log w}{\log \log w})$. More specifically, we prove the following important theorem:

Theorem 2: By the RAP, the congestion is $\tilde{O}(\frac{\log w}{\log \log w})$ for any memory access by a warp. In particular, the congestion of the contiguous access and the stride access is 1.

We will prove that the congestion of any memory access is at most $\tilde{O}(\frac{\log w}{\log \log w})$. For the purpose of the proof, we use an important probability theorem called the Chernoff bound that estimates the tail probability of the Poisson trials as follows:

Theorem 3 (Chernoff Bound [20]): Let X_0, X_1, ..., X_{n-1} be independent Poison trials such that $X_i = 1$ with probability p_i ($0 \leq i \leq n-1$). Let $X = \sum_{i=0}^{n-1} X_i$ and $\mu = E[X] = \sum_{i=0}^{n-1} p_i$. We have the following inequality for any $\delta > 0$:

$$\Pr[X > (1 + \delta)\mu] \quad < \quad \left(\frac{e^{\delta}}{(1 + \delta)^{(1+\delta)}}\right)^{\mu}$$

Please see [20] for the details of the Chernoff bound. In paper [7] we use Theorem 3 to prove Theorem 2 for the RAS. This is possible because random numbers r_0, r_1, ... used by the RAS are independent. However, these random numbers by the RAP are not independent. Hence, it is not possible to use Theorem 3 as it is for the proof of Theorem 2. We use several new proof techniques to prove Theorem 2 by Theorem 3.

For simplicity, we assume that no two threads access the same address. Clearly, this assumption makes sense for the proof of Theorem 2, because it does not decrease the probability of bank conflicts and the memory access congestion. We partition w threads in a warp into two half warps such that each half warp has $\frac{w}{2}$ threads. We will show that the memory access congestion by $\frac{w}{2}$ threads in a half warp is $\tilde{O}(\frac{\log w}{\log\log w})$. This implies that the congestion by w threads in a warp is at most $2 \cdot \tilde{O}(\frac{\log w}{\log\log w}) = \tilde{O}(\frac{\log w}{\log\log w})$. Let $i_0, i_1, \ldots, i_{\frac{w}{2}-1}$ and $j_0, j_1, \ldots, j_{\frac{w}{2}-1}$ be the indexes of a such that each thread $T(k)$ ($0 \leq k \leq \frac{w}{2}-1$) by a half warp accesses $a[i_k][j_k]$. Using the RAP technique, each $T(k)$ accesses $a[i_k][(j_k + r_{i_k}) \bmod w]$ instead. Let $A(i)$ ($0 \leq i \leq w-1$) be the number of memory access requests destined for the i-th row of a. Since no two threads access the same address, we have $\sum_{i=0}^{w-1} A(i) = \frac{w}{2}$.

For a fixed bank $B[u]$ ($0 \leq u \leq w-1$), we will show that more than $\frac{e^2 \ln w}{\ln\ln w}$ memory requests is destined for $B[u]$ with probability at most $\frac{1}{w^2}$. Let i_0, i_1, ..., $i_{w'-1}$ ($0 \leq i_0 < i_1 < \cdots < i_{w'-1} \leq w-1$) denote rows accessed by $\frac{w}{2}$ threads in a half warp. In other words, $A(i_k) \geq 1$ for all k ($0 \leq k \leq w'-1$) and $w' \leq \frac{w}{2}$. Imagine that r_{i_0}, r_{i_1}, ..., $r_{i_{w'-1}}$ are determined one by one for the purpose of evaluating the congestion. In other words, each r_{i_k} is selected from integers in $[0, w-1] - \{r_{i_0}, r_{i_1}, \ldots, r_{i_{k-1}}\}$ at random. First, let us evaluate the probability that a half warp accesses $B[u]$ by memory access requests in the i_0-th row. Since $A(i_0)$ memory cells in the i_0-th row are accessed, the probability is $\frac{A(i_0)}{w}$. Next, we evaluate the probability that a half warp accesses $B[u]$ in the i_1-th row. Since r_{i_1} is selected from $[0, w-1] - \{r_{i_0}\}$ at random, the probability is 0 at most $\frac{A(i_1)}{w-1}$. Similarly, the probability that a half warp accesses $B[u]$ in the i_2-th row is at most $\frac{A(i_2)}{w-2}$, because r_{i_2} is selected from $[0, w-1] - \{r_{i_0}, r_{i_1}\}$ at random, In general, the probability that a half warp accesses $B[u]$ in the i_k-th row is at most $\frac{A(i_k)}{w-k}$ for each k ($0 \leq k \leq w'-1$). From $w' \leq \frac{w}{2}$, we have $\frac{A(i_k)}{w-k} \leq \frac{2A(i_k)}{w}$. To evaluate the number of memory cells in $B[u]$ accessed by a half warp, let $X_0, X_1, \ldots, X_{w'-1}$ be independent random binary variables such that $X_k = 1$ with probability $\frac{2A(i_k)}{w}$. Further, let $X = X_0 + X_1 + \cdots + X_{w'-1}$. Clearly, X is the random variable that provides the upper bound of the number of memory access destined for bank $B[u]$ by a half warp. Since random variables $X_0, X_1, \ldots, X_{w'-1}$ are independent, we can apply Theorem 3 to evaluate the tail probability of X and we have the following lemma:

Lemma 4: For random variable X defined above, we have,

$$\Pr[X > \frac{e^2 \ln w}{\ln\ln w}] \quad < \quad \frac{1}{w^2}.$$

Proof: Clearly, the expected value of X is

$$\mu \quad = \quad E[X] = \sum_{k=0}^{w'-1} \frac{2A(i_k)}{w} = 1.$$

Hence, from Theorem 3 with $\mu = 1$, we have

$$\Pr[X > (1+\delta)] \quad < \quad \frac{e^\delta}{(1+\delta)^{(1+\delta)}}$$

for any $\delta > 0$. Let $1 + \delta = \frac{e^2 \ln w}{\ln\ln w}$. We will prove that $\frac{e^\delta}{(1+\delta)^{(1+\delta)}} < \frac{1}{w^2}$, that is, $\ln \frac{e^\delta}{(1+\delta)^{(1+\delta)}} < -2\ln w$ as follows:

$$\begin{aligned}
&\ln \frac{e^\delta}{(1+\delta)^{(1+\delta)}} \\
&= \quad \delta - (1+\delta)\ln(1+\delta) \\
&= \quad \frac{e^2 \ln w}{\ln\ln w} - 1 - \frac{e^2 \ln w}{\ln\ln w} \ln \frac{e^2 \ln w}{\ln\ln w} \\
&< \quad -\frac{e^2 \ln w}{\ln\ln w}(-1 + \ln e^2 + \ln\ln w - \ln\ln\ln w) \\
&< \quad -\frac{e^2 \ln w}{\ln\ln w} \cdot \frac{\ln\ln w}{2} < -2\ln w
\end{aligned}$$

This completes the proof. ■

Let Y be a random variable denoting the memory access congestion by $\frac{w}{2}$ threads in a half warp. In other words, Y is the maximum number of memory access requests over all banks $B[u]$ ($0 \leq u \leq w-1$). From Lemma 4, we have

$$\Pr[Y > \frac{e^2 \ln w}{\ln\ln w}] \leq \Pr[X > \frac{e^2 \ln w}{\ln\ln w}] \cdot \frac{w}{2} < \frac{1}{2w}.$$

Thus, we have,

$$\Pr[0 \leq Y \leq \frac{e^2 \ln w}{\ln\ln w}] < 1 \text{ and } \Pr[\frac{e^2 \ln w}{\ln\ln w} < Y \leq w] < \frac{1}{2w}.$$

Hence, the expected value of Y is at most:

$$\begin{aligned}
E[Y] \quad &\leq \quad \Pr[0 \leq Y \leq \frac{e^2 \ln w}{\ln\ln w}] \cdot \frac{e^2 \ln w}{\ln\ln w} \\
&\quad\quad + \Pr[\frac{e^2 \ln w}{\ln\ln w} < Y \leq w] \cdot w \\
&< \quad 1 \cdot \frac{e^2 \ln w}{\ln\ln w} + \frac{1}{2w} \cdot w = O(\frac{\log w}{\log\log w}).
\end{aligned}$$

We have proved that the congestion of any memory access by a half warp is $\tilde{O}(\frac{\log w}{\log\log w})$ by the RAP. Since the congestion of a warp is not more than the sum of those of the first warp and the second warp, we have Theorem 2.

V. Simulation results

The main purpose of this section is to show simulation results to evaluate the congestions. We have shown in Theorem 2, the memory access congestion by the RAP is $\tilde{O}(\frac{\log w}{\log \log w})$ for any memory access. Sometimes, the big-O notation has a large constant factor, and the actual value is too large to use it in practical applications. We will show that the actual value here is enough small for the purpose of practical GPU implementations.

Table II shows the congestion of memory access obtained by simulation. The number w of memory banks and threads in a warp of latest CUDA-enabled GPUs is 32. The value of w may be increased in future GPUs. Thus, we have performed simulation for various values of w up to 256.

From the table, we can confirm that the contiguous memory access has no bank conflict for all implementations. The RAW and RAS implementations for the stride access involve bank conflicts while the RAP implementation has no bank conflict. Since the diagonal access is optimized for the RAW implementation, the congestion of the RAW implementation is 1. On the other hand, the RAS and the RAP implementations have bank conflicts, but the congestion is moderately small. Also, the congestion by the RAP is slightly larger than that by the RAS. For example, when $w = 32$, the congestion by the RAP is 3.61 while that by the RAS is 3.53. This is because the probability of bank conflicts by the RAP is slightly larger than the RAS. For example, two memory access requests to distant addresses are destined for the same bank with probability $\frac{1}{w}$ if the RAS is used. On the other hand, the probability is $\frac{1}{w-1}$ if the RAP is used. For the random memory access, all three implementations have the same congestion for each w.

Consequently, we can say that the congestion of the RAP implementation is smaller than or equal to that of the RAW and the RAS implementations for most memory accesses excluding the diagonal memory access. The RAP has the largest congestion for the diagonal memory access, but the overhead is small. Hence, we should use the RAP implementation if

- addresses accessed by threads are not known beforehand, or
- a CUDA developer has difficulty to minimize bank conflicts.

A CUDA developer can reduce the congestion automatically in most cases by using the RAP.

VI. Experimental results using GTX GeForce TITAN

The main purpose of this section is to show how we have implemented the RAP in a CUDA-enabled GPU. We also show the experimental results for matrix transpose by the RAW, the RAS, and the RAP implementations.

We can implement the three matrix transpose algorithms using the RAP technique. For example, the CRSW transpose algorithm by the RAP for a matrix of size 32×32 is described as follows:

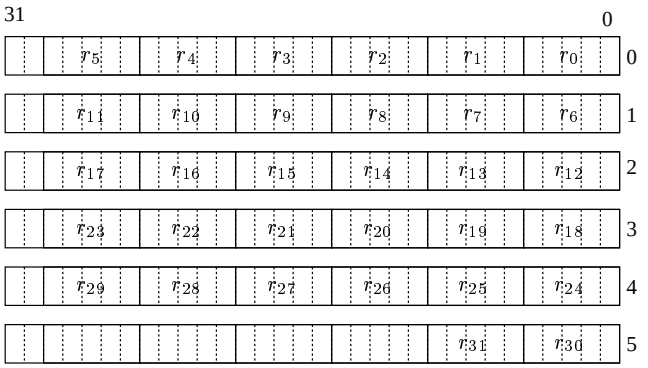


Figure 7. Arrangement of random numbers $r_i (0 \leq i \leq 31)$ in local registers r[*]

[The RAP implementation of the CRSW transpose]

```
__shared__ double a[32][32], b[32][32];
int r[6];
int i = threadIdx.x/32;
int j = threadIdx.x%32;
b[(j+(r[i/6]>>(5*(i%6))))&0x1f][i]
  = a[i][(j+(r[i/6]>>(5*(i%6))))&0x1f];
```

The transpose is performed for matrix a of size 32×32 and the resulting values are written in matrix b. We assume that $a[i][(j + r_i) \bmod 32]$ $(0 \leq i, j \leq w-1)$ stores the (i, j) elements of a matrix. Also, an array r of six local registers stores random numbers $r_0, r_1, \ldots, r_{31}$ in the range $[0, 31]$ such that each $r[i]$ $(0 \leq i \leq 5)$ stores 6 random numbers $r_{i\cdot 6}, r_{i\cdot 6+1}, \ldots, r_{i\cdot 6+5}$. Since each $r[i]$ has 32 bits and each r_j has 5 bits, this is possible. The reader should refer to Figure 7 illustrating how random numbers r are stored in local registers $r[*]$. Since the value of $a[i][(j + r_i) \bmod 32]$ is copied to $b[(j + r_i) \bmod 32][i]$, the transpose is completed correctly.

We have implemented three matrix transpose algorithms, Contiguous Read Stride Write (CRSW), Stride Read Contiguous Write (SRCW), and Diagonal Read Diagonal Write (DRDW) using the RAW, the RAS, and the RAP. For the CRSW and the SRCW transpose, the RAP implementation runs about 10 times faster than the RAW implementation and 2 times faster than the RAS implementation. The DRDW transpose performs the worst memory access for the RAP implementation. Nonetheless, it runs only 2.5 times slower than the RAW implementation. It follows that, the RAP is practical and works efficiently in current GPUs. When a CUDA developer implements some algorithm in the GPUs, it is not necessary to analyze and reduce the memory access congestion. It is sufficient to apply the RAP and the resulting implementation has small memory access congestion.

Table II
THE CONGESTION OF MEMORY ACCESS TO A MATRIX OF SIZE $w \times w$

	RAW Implementation					RAS Implementation					RAP Implementation				
w	16	32	64	128	256	16	32	64	128	256	16	32	64	128	256
Contiguous	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Stride	16	32	64	128	256	3.08	3.53	3.96	4.38	4.77	1	1	1	1	1
Diagonal	1	1	1	1	1	3.08	3.53	3.96	4.38	4.77	3.20	3.61	4.00	4.41	4.78
Random	2.92	3.44	3.90	4.34	4.75	2.92	3.44	3.90	4.34	4.75	2.92	3.44	3.90	4.34	4.75

Table III
THE CONGESTION ON THE DMM AND THE COMPUTING TIME ON THE GPU FOR CRSW, SRCW, AND DRDW ALGORITHMS BY THE RAW, THE RAS, AND THE RAP IMPLEMENTATIONS

	RAW Implementation			RAS Implementation			RAP Implementation		
	congestion		time (in ns)	congestion		time (in ns)	congestion		time (in ns)
	read	write	on the GPU	read	write	on the GPU	read	write	on the GPU
CRSW Transpose	1	32	1595	1	3.53	303.6	1	1	154.5
SRCW Transpose	32	1	1596	3.53	1	297.1	1	1	159.1
DRDW Transpose	1	1	158.4	3.53	3.53	427.4	3.61	3.61	433.3

VII. THE RANDOM ADDRESS PERMUTE-SHIFT FOR HIGHER DIMENSION

So far, we have presented the random address permute-shift (RAP) for a matrix of size $w \times w$. The main purpose of this section is to discuss several ideas to extend the RAP for larger arrays than $w \times w$. For simplicity, we focus on a 4-dimensional array a of size $w \times w \times w \times w$. Note that, each element $a[i][j][k][l]$ ($0 \leq i, j, k, l \leq w-1$) is allocated in address $i \cdot w^3 + j \cdot w^2 + k \cdot w + l$ and it is in bank $B[l]$.

Basically, the RAP technique maps each $a[i][j][k][l]$ to $a[i][j][k][(l + f(i,j,k)) \bmod w]$ for some linear function f that determines the length of shift from i, j, and k. We use several memory accesses by a warp of w threads to evaluate the congestion as follows:

Contiguous: $a[i][j][k][0 : w-1]$, that is, $a[i][j][k][0]$, $a[i][j][k][1]$, ..., $a[i][j][k][w-1]$ are accessed.
Stride1: $a[i][j][0 : w-1][l]$ are accessed.
Stride2: $a[i][0 : w-1][k][l]$ are accessed.
Stride3: $a[0 : w-1][j][k][l]$ are accessed.
Random: w elements in a are selected at random are accessed.
Malicious: Malicious memory accesses that maximize the memory access congestion.

We introduce several possible RAP techniques as follows:

One permutation (1P): For a random permutation r of $(0, 1, \ldots, w-1)$, $f(i,j,k) = r_k$.
Repeated one permutation (R1P): For a random permutation r of $(0, 1, \ldots, w-1)$, $f(i,j,k) = r_i + r_j + r_k$.
Three random permutations (3P): For three independent random permutations r, s, and t of $(0, 1, \ldots, w-1)$, $f(i,j,k) = r_i + s_j + t_k$.
w^2 random permutations (w^2P): For w^2 random permutations $r^0, r^1, \ldots r^{w^2-1}$ of $(0, 1, \ldots, w-1)$, $f(i,j,k) = r_k^{i \cdot w + j}$.
one random permutation and w^2 random numbers (1Pw^2R): For a random permutation r of $(0, 1, \ldots, w-1)$ and w^2 independent random integers $s_0, s_1, \ldots, s_{w^2-1}$ in $[0, w-1]$, $f(i,j,k) = s_{i \cdot w + j} + r_k$.

Table IV summarizes the congestion and used random numbers by each RAP for an array of size w^4. Due to the stringent page limitation, we omit the details of explanation, but the reader should have no difficulty to confirm that the congestion and the random numbers are correct. From the table, we can see that R1P and 3P have good performance in terms of the congestion and the used random numbers. Note that R1P have malicious inputs with high congestion. For example, 6 memory access requests to $a[0][1][2][l]$, $a[0][2][1][l]$, $a[1][0][2][l]$, $a[1][2][0][l]$, $a[2][0][1][l]$, and $a[2][1][0][l]$ are destined to bank $B[(l + r_0 + r_1 + r_2) \bmod w]$. Since we can have $\frac{w}{6}$ groups of such 6 memory access requests each, the congestion can be as large as $6 \cdot \tilde{O}(\frac{\log \frac{w}{6}}{\log \log \frac{w}{6}})$, which is much larger than $\tilde{O}(\frac{\log \frac{w}{6}}{\log \log \frac{w}{6}})$ from the practical point of view. Thus, we believe that 3P is the best method to extend the RAP for larger arrays.

VIII. CONCLUSION

We have presented a novel algorithmic technique called the random address permute-shift (RAP) that achieves $\tilde{O}(\frac{\log w}{\log \log w})$ memory access congestion for any memory access requests by a warp of w threads. In particular, the congestion of any contiguous and any stride memory access is always 1. We have also applied the RAP to matrix transpose on the shared memory of a streaming multiprocessor on the GeForce GTX TITAN. The experimental results show that, even the direct transpose algorithms run very fast by the RAP technique. From the experimental results, we can say that our new RAP technique is practical and a potent

Table IV
THE CONGESTION AND THE USED RANDOM NUMBERS BY THE RAW, THE RAS, AND THE RAPS FOR AN ARRAY OF SIZE w^4

	RAW	RAS	RAP				
			1P	R1P	3P	w^2P	1Pw^2R
Contiguous	1	1	1	1	1	1	1
Stride1	w	$\tilde{O}(\frac{\log w}{\log\log w})$	1	1	1	1	1
Stride2	w	$\tilde{O}(\frac{\log w}{\log\log w})$	w	1	1	$\tilde{O}(\frac{\log w}{\log\log w})$	$\tilde{O}(\frac{\log w}{\log\log w})$
Stride3	w	$\tilde{O}(\frac{\log w}{\log\log w})$	w	1	1	$\tilde{O}(\frac{\log w}{\log\log w})$	$\tilde{O}(\frac{\log w}{\log\log w})$
Random	$\tilde{O}(\frac{\log w}{\log\log w})$	$\tilde{O}(\frac{\log w}{\log\log w})$	$\tilde{O}(\frac{\log w}{\log\log w})$	$\tilde{O}(\frac{\log w}{\log\log w})$	$\tilde{O}(\frac{\log w}{\log\log w})$	$\tilde{O}(\frac{\log w}{\log\log w})$	$\tilde{O}(\frac{\log w}{\log\log w})$
Malicious	w	$\tilde{O}(\frac{\log w}{\log\log w})$	w	$6 \cdot \tilde{O}(\frac{\log \frac{w}{6}}{\log\log \frac{w}{6}})$	$\tilde{O}(\frac{\log w}{\log\log w})$	$\tilde{O}(\frac{\log w}{\log\log w})$	$\tilde{O}(\frac{\log w}{\log\log w})$
Random numbers	0	w^3	w	w	$3w$	w^3	$w^2 + w$

method to reduce the memory access congestion for the shared memory automatically.

REFERENCES

[1] W. W. Hwu, *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, 2011.

[2] K. Ogawa, Y. Ito, and K. Nakano, "Efficient Canny edge detection using a GPU," in *Proc. of International Conference on Networking and Computing*. IEEE CS Press, Nov. 2010, pp. 279–280.

[3] A. Uchida, Y. Ito, and K. Nakano, "Fast and accurate template matching using pixel rearrangement on the GPU," in *Proc. of International Conference on Networking and Computing*. IEEE CS Press, Dec. 2011, pp. 153–159.

[4] NVIDIA Corporation, "NVIDIA CUDA C programming guide version 5.0," 2012.

[5] ——, "NVIDIA CUDA C best practice guide version 3.1," 2010.

[6] K. Nakano, "Simple memory machine models for GPUs," in *Proc. of International Parallel and Distributed Processing Symposium Workshops*, May 2012, pp. 788–797.

[7] K. Nakano, S. Matsumae, and Y. Ito, "The random address shift to reduce the memory access congestion on the discrete memory machine," in *Proc. of International Symposium on Computing and Networking*, Dec. 2013, pp. 95–103.

[8] A. Kasagi, K. Nakano, and Y. Ito, "Offline permutation algorithms on the discrete memory machine with performance evaluation on the GPU," *IEICE Transactions on Information and Systems*, vol. Vol. E96-D, no. 12, pp. 2617–2625, Dec. 2013.

[9] K. Nakano, "Sequential memory access on the unified memory machine with application to the dynamic programming," in *Proc. of International Symposium on Computing and Networking*, Dec. 2013, pp. 85–94.

[10] A. V. Aho, J. D. Ullman, and J. E. Hopcroft, *Data Structures and Algorithms*. Addison Wesley, 1983.

[11] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. C-21, pp. 948–960, 1972.

[12] S.-H. Hsiao and C. Y. R. Chen, "Performance evaluation of circuit switched multistage interconnection networks using a hold strategy," *IEEE Transactions on Parallel and Distributed Systems*, pp. 632–640, Sept. 1992.

[13] A. Kasagi, K. Nakano, and Y. Ito, "An implementation of conflict-free off-line permutation on the GPU," in *Proc. of International Conference on Networking and Computing*, 2012, pp. 226–232.

[14] ——, "An optimal offline permutation algorithm on the hierarchical memory machine, with the GPU implementation," in *Proc. of International Conference on Parallel Processing*. IEEE CS Press, Oct. 2013, pp. pp. 1–10.

[15] K. Nakano and S. Matsumae, "The super warp architecture with random address shift," in *Proc. of High Performance Computing*, Dec. 2013.

[16] K. Mehlhorn and U. Vishkin, "Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories," *Acta Informatica*, vol. 21, no. 4, pp. 339 – 374, Nov. 1984.

[17] M. Dietzfelbinger and F. M. auf der Heide, "Simple, efficient shared memory simulations," in *Proc.of ACM Symposium on Parallel Algorithms and Architectures*, June 1993, pp. 110 – 119.

[18] NVIDIA Corporation. (2013) NVIDIA GeForce GTX TITAN. [Online]. Available: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan/

[19] A. Gibbons and W. Rytter, *Efficient Parallel Algorithms*. Cambridge University Press, 1988.

[20] R. Motwani and P. Raghavan, *Randomized Algorithms*. Cambridge University Press, 1995.

GPUのPTXコードを用いた ランダムアドレスシフトの厳密評価

広島大学　藤田 徹

研究概要

GPUのアセンブリ言語であるPTXコードを用いて，メモリアクセスの高速化手法であるランダムアドレスシフトの厳密な性能評価を実施

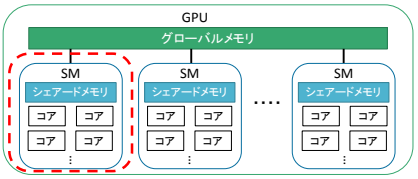
- 命令をPTXコードで記述することによって，ランダムアドレスシフトの実行クロックサイクル数を計測

GPU（Graphics Processing Unit）

- グラフィックス処理に適したハードウェア
- 内部に多数のコアを搭載し，並列演算能力に優れる
- NVIDIA社が提供するGPUの統合開発環境CUDAを利用

GPUアーキテクチャ

NVIDIA社のGPUには複数のストリーミングマルチプロセッサ（SM）が搭載されている



- 複数のコアとシェアードメモリから構成
- シェアードメモリは同じSM内のコアのみアクセス可能

各コアにはスレッドが割り当てられる

- 処理はスレッドを32個毎にまとめたワープ単位で実行される

今回は1つのSMのみに注目して計測を行う

シェアードメモリとバンクコンフリクト

シェアードメモリは1ワードずつ32個のバンクによって構成されている

複数のスレッドが同じバンクの異なるアドレスに同時にアクセス

- アクセスが逐次化され，スループットが低下 ➡ バンクコンフリクト

（例）バンク数4の場合

衝突は発生しない ⇩ 同時にアクセス

シェアードメモリとバンクコンフリクト

シェアードメモリは1ワードずつ32個のバンクから構成されている

複数のスレッドが同じバンクの異なるアドレスに同時にアクセス

- アクセスが逐次化され，スループットが低下 ➡ バンクコンフリクト

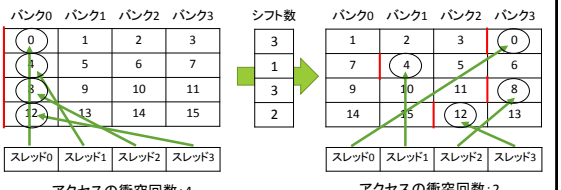
（例）バンク数4の場合

衝突が発生 ⇩ アクセスが逐次化

論文名：The Random Address Shift to Reduce the Memory Access Congestion on the Discrete Memory Machine
著者：Koji Nakano, Susumu Matsumae, and Yasuaki Ito
出典：International Symposium on Computing and Networking, pp. 95–103, Dec. 2013

ランダムアドレスシフト

シェアードメモリに格納する要素をシフト数にしたがって行方向にサイクリックシフト



アクセスの衝突回数：4　　アクセスの衝突回数：2

アドレスシフトを行うことによってバンクコンフリクトが低減

シフト数の決定方法

Random Address Shift

✓各行のシフト数は独立した乱数によって決定される

シフト数	バンク0	バンク1	バンク2	バンク3
3	1	2	3	(0)
1	7	(4)	5	6
3	9	10	11	(8)
2	14	15	(12)	13

Random Address Permute-Shift

✓ランダムに置換した数列に従ってシフトする

✓各行のシフト数はすべて異なる

シフト数	バンク0	バンク1	バンク2	バンク3
2	2	3	(0)	1
1	7	(4)	5	6
3	9	10	11	(8)
0	(12)	13	14	15

アクセス時間の計測

GPU：GeForce GTX 680

- 1SM当たりのコア数：192
- 動作クロック：1006MHz

次の4つのアクセスパターンについてワープ数を1から32まで変化させて1000回計測を行い，平均時間を算出

- Contiguous
- Stride
- Diagonal
- Random

アクセスに必要な実行クロックサイクル数を計測することで厳密な性能評価を実施

PTXコードを用いたクロックサイクル数の計測方法

GPUのアセンブリ言語

- インラインアセンブラを用いてソースコード内に記述できる

この範囲のクロックサイクル数を計測

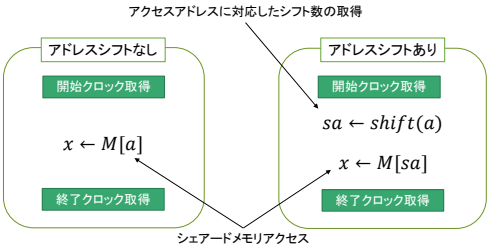
```
asm volatile (
        "mov.u32 %%a0, %3;¥n¥t"
        "mov.u32 %%a1, sh_mem;¥n¥t"
        "bar.sync 0;¥n¥t"
        "mov.u32 %0, %%clock;¥n¥t"
        "shl.b32 %%a2, %%a0, 3;¥n¥t"
        "add.u32 %%a2, %%a1, %%a2;¥n¥t"
        "ld.volatile.shared.u64 %2, [%%a2];¥n¥t"
        "mov.u32 %1, %%clock;¥n¥t"
        :"=r"(begin), "=r"(end), "=l"(temp)
        :"r"(AID[threadIdx.x])
);
```

クロックレジスタの値を取得

➢ クロックカウンタの値が格納されている

測定したい命令をクロックカウンタの取得命令で挟み，その差分を求めることで命令の実行クロックサイクル数が計測できる

計測範囲



計測するアクセスパターン

1. Contiguous Access
   - 各スレッドは行方向にアクセス

バンク0	バンク1	バンク2	バンク3
0	1	2	3
(4)	(5)	(6)	(7)
8	9	10	11
12	13	14	15

アクセスは衝突しない

- Random Shift

	バンク0	バンク1	バンク2	バンク3
3	1	2	3	0
1	(7)	(4)	(5)	(6)
3	9	10	11	8
2	14	15	12	13

アクセスは衝突しない

- Permute Shift

	バンク0	バンク1	バンク2	バンク3
2	2	3	0	1
1	(7)	(4)	(5)	(6)
3	9	10	11	8
0	12	13	14	15

アクセスは衝突しない

計測するアクセスパターン

2. Stride Access
   - 各スレッドは列方向にアクセス

バンク0	バンク1	バンク2	バンク3
0	(1)	2	3
4	(5)	6	7
8	(9)	10	11
12	(13)	14	15

全てのスレッドでアクセスが衝突

- Random Shift

	バンク0	バンク1	バンク2	バンク3
3	(1)	2	3	0
1	7	4	(5)	6
3	(9)	10	11	8
2	14	15	12	(13)

一部のスレッドでアクセスが衝突

- Permute Shift

	バンク0	バンク1	バンク2	バンク3
2	2	3	0	(1)
1	7	4	(5)	6
3	(9)	10	11	8
0	12	(13)	14	15

アクセスは衝突しない

計測するアクセスパターン

3. Diagonal Access
   - 各スレッドは斜め方向にアクセス

バンク0	バンク1	バンク2	バンク3
(0)	1	2	3
4	(5)	6	7
8	9	(10)	11
12	13	14	(15)

アクセスは衝突しない

- Random Shift

	バンク0	バンク1	バンク2	バンク3
3	1	2	3	(0)
1	7	4	(5)	6
3	9	(10)	11	8
2	14	(15)	12	13

一部のスレッドでアクセスが衝突

- Permute Shift

	バンク0	バンク1	バンク2	バンク3
2	2	3	(0)	1
1	7	4	(5)	6
3	9	(10)	11	8
0	12	13	14	(15)

一部のスレッドでアクセスが衝突

計測するアクセスパターン

4. Random Access
   - 各スレッドはランダムにアクセス

バンク0	バンク1	バンク2	バンク3
0	1	2	(3)
(4)	5	6	7
8	9	(10)	(11)
12	13	14	15

一部のスレッドでアクセスが衝突

- Random Shift

	バンク0	バンク1	バンク2	バンク3
3	1	2	(3)	0
1	7	(4)	5	6
3	9	(10)	(11)	8
2	14	15	12	13

一部のスレッドでアクセスが衝突

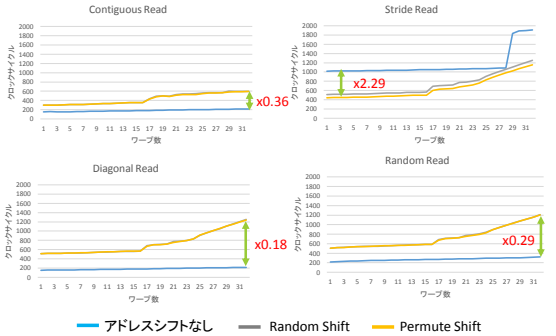
- Permute Shift

	バンク0	バンク1	バンク2	バンク3
2	2	(3)	0	1
1	7	(4)	5	6
3	9	(10)	(11)	8
0	12	13	14	15

一部のスレッドでアクセスが衝突

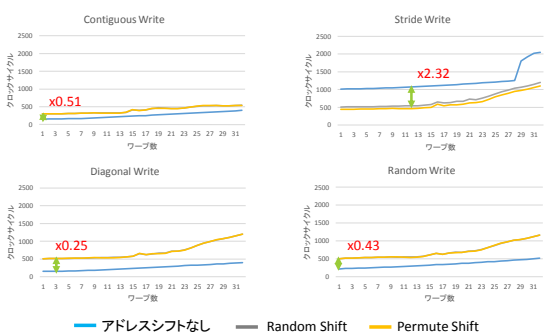
計測結果（read命令）

アドレスシフトを行わない場合に対する高速化率



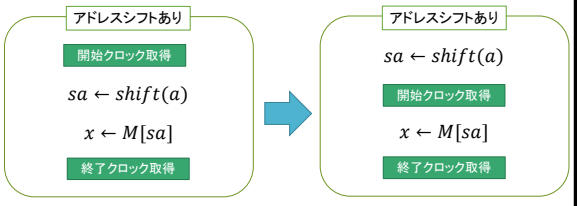
計測結果（write命令）

アドレスシフトを行わない場合に対する高速化率



計測範囲の変更

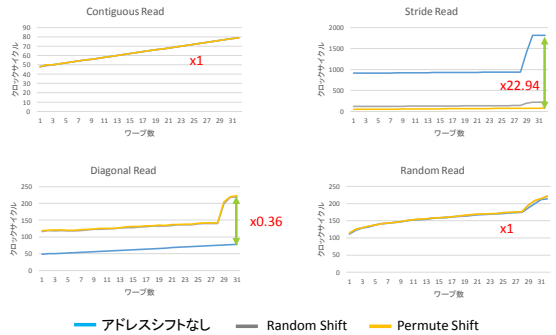
アドレスシフトありの場合の計測範囲をメモリアクセスのみに変更する



アドレスシフトがハードウェア化された場合を想定
- ➢ アドレスシフトの計算時間を無視できる

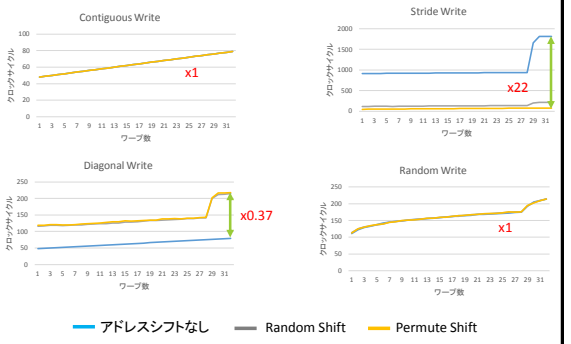
計測結果（read命令）

アドレスシフトを行わない場合に対する高速化率



計測結果（write命令）

アドレスシフトを行わない場合に対する高速化率



アドレスシフトなし　Random Shift　Permute Shift

まとめ

GPUのアセンブリ言語であるPTXコードを用いて，ランダムアドレスシフトの厳密な性能評価を行った

アドレスシフトを求める時間を含めた場合

- strideアクセスで約2.3倍の高速化となり，その他のアクセスパターンではアドレスシフトを行わない場合が高速となった

アドレスシフトを求める時間を含めない場合

- strideアクセスで約23倍の高速化となり，Contiguous，Randomアクセスではアドレスシフトを行わない場合と大きな差は見られなかった

GPUの共有メモリアクセスのレイテンシとスループットの計測

岡本 悟史
広島大学大学院工学研究科

概 要

グラフィックス処理を行うためのデバイスであるGPUの演算能力を汎用演算に応用する技術をGPGPUと呼び，近年活発に研究が進められている．NVIDIA社のGPUには複数の演算コアで共有可能で尚且つアクセスレイテンシの非常に小さなオンチップ共有メモリが存在し，GPGPUによる処理の高速化において重要な役割を担っている．しかしこの共有メモリはバンクコンフリクトの発生等の要因によってメモリアクセスのレイテンシ，及びスループットが大きく変動する特徴がある．

本研究では，GPUのクロックカウンタを利用して共有メモリアクセス時間の計測を行い，バンクコンフリクト，メモリアクセス命令の実行回数，実行時のスレッド数が共有メモリのレイテンシ，スループットにどのような影響を与えるかを明らかにする．NVIDIA GeForce GTX780Tiを用いて共有メモリアクセス命令の実行クロックサイクル数を計測することで，最適なアクセスを行った際のレイテンシが16clock cycleであるという結果が得られた．また，バンクコンフリクト，メモリアクセス命令数の実行回数，実行時のスレッド数とメモリアクセスに必要な実行時間の関係を表す回帰式も得ることが出来た．

1. はじめに

複数のプロセッサが同時に処理を行うことで演算スループットを向上させる並列計算は，高い計算性能を実現する方法として非常に有効である．GPU(Graphics Processing Unit) は内部に搭載した多数の演算コア，高いメモリバンド幅から並列計算への応用が盛んに行われ，HPC(High Performance Computing) 分野で注目を集めている．本来グラフィックス処理を行うためのデバイスであるGPUを汎用演算に用いるこの技術はGPGPU(General-Purpose computing on GPUs)と呼ばれている．NVIDIA社のGPUでは同社の提供するGPU向け統合開発環境CUDA(Compute Unified Device Architecture)[1]を用いることで並列処理を実装することが出来る．

NVIDIA社のGPUは並列処理に用いる大量のデータを効率良く扱うために図1のようなメモリ階層構造を取り入れている．CUDAを用いたGPGPUを実装する上で重要となるのがグローバルメモリ，シェアードメモリの二つのメモリである．グローバルメモリはオフチップDRAM上にあり，数GBと容量が非常に多いメモリである．しかしグローバルメモリはオフチップメモリであるためレイテンシが非常に大きいという欠点がある．これに対してシェアードメモリはオンチップ共有メモリであり，容量が小さいがオフチップメモリに比べてアクセスが非常に早く，GPGPUによる処理の高速化において重要な役割を担っている．

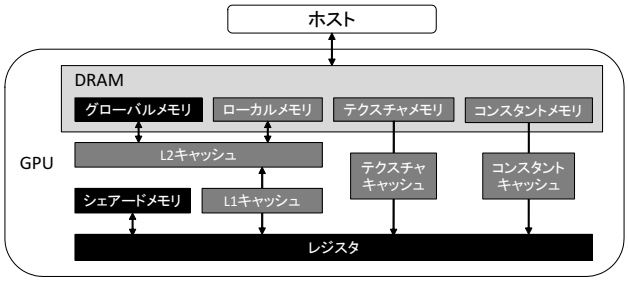


図 1: GPUのメモリ階層構造

しかし，シェアードメモリのレイテンシ及びスループットはアクセスのパターン等によって大きく変動してしまう．NVIDIA社はグローバルメモリやシェアードメモリ等のアクセスの特徴をCUDA Programming Guide[2]に記載しているが具体的な数値による詳細な説明はされていない．そのため，GPUのベンチマークを行うことでメモリスループット等を計測する研究[3, 4]や，GPUのシミュレータを用いたアーキテクチャの分析[5]等が進められている．

本研究では，NVIDIA社のGPUであるGeForce 780Tiを用いたシェアードメモリアクセス時間の測定を行うことによってレイテンシ，及びスループットを明らかにすることを目的とする．

2. GPU アーキテクチャ

2.1. ハードウェアアーキテクチャ

GPUのハードウェアアーキテクチャを図2に示す．GPUは複数のSM(Streaming Multiprocessor)を持ち，また一つのSM内には複数の演算コアが存在する．オンチップ共有メモリであるシェアードメモリは各SM上に存在し，SM内の演算コア間で共有することができる．表1は本研究に用いるGeForce GTX 780TiについてNVIDIA社によって公開されている性能をまとめた表である．グローバルメモリが置かれるDRAMは3072MBであるのに対してシェアードメモリは各SM当たり16から48KBと非常に容量が少ないことが読み取れる．

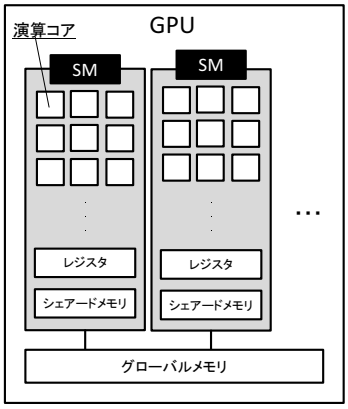


図 2: ハードウェアアーキテクチャ

表 1: GeForce GTX 780Ti の性能一覧

GPU	
コアアーキテクチャ	GK110
ベースクロック	875MHz
ブーストクロック	928MHz
演算コア数	2880
SMX 数	15 基
DRAM サイズ	3GB
SMX	
SMX 当たりの演算コア数	192
レジスタ数	65536
シェアードメモリ	16KB or 32KB or 48KB

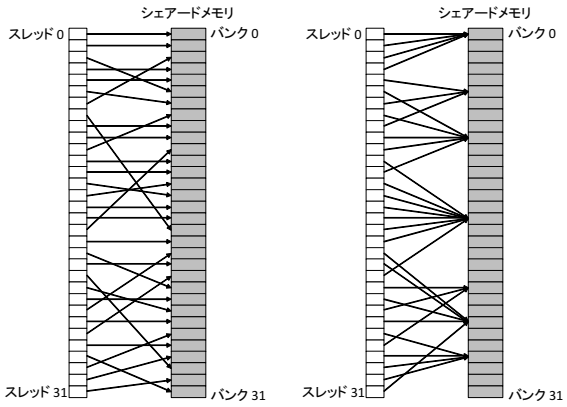
2.2. CUDA

NVIDIA 社の GPU で汎用並列計算を行うための統合開発環境として CUDA が提供されている．CUDA はスレッドレベル並列性によって並列計算を実現する．複数の演算コアで各スレッドが同じ命令を実行し，全体のスループットを向上させている．32 個のスレッドをまとめたものはワープと呼ばれ，ワープ内のスレッドは同時に同じ命令を実行する．

2.3. バンクコンフリクト

シェアードメモリアクセスもワープ単位で実行されるため，ワープ内のシェアードメモリアクセスは同時に実行される．しかし，アクセスのパターンによってはアクセスが逐次化され，シェアードメモリアクセスを完了するまでの時間が長くなってしまう可能性がある．

シェアードメモリはバンクと呼ばれる単位に区切られ，バンクそれぞれに 0〜31 の番号が割り当てられる．ワープ内のスレッドが同時にアクセスを行う際にそれらのバンク番号が全て異なっていれば，一度に全てのデータにアクセスできる (図 3(a))．しかしバンク番号が異なる場合，アクセスが逐次化されることで速度が非常に遅くなってしまう (図 3(b))．このような現象をバンクコンフリクトと呼び，処理の高速化のためにはバンクコンフリクトを出来る限り回避することが重要である．



(a) バンクコンフリクトが発生しないアクセス　(b) バンクコンフリクトが発生するアクセス

図 3: シェアードメモリとバンクコンフリクト

3. 計測方法

シェアードメモリは SM 毎に搭載されているため，計測には一つの SM のみを使用する．ある一つの SM におけるシェアードメモリアクセスの挙動を知ることが出来れば，他の SM も同じ用に振る舞うため GPU 全体での動きも予想することが出来る．

3.1. PTX(Parallel Thread Execution)

シェアードメモリアクセスのみの実行時間を計測するために，CUDA のアセンブリ言語である PTX(Parallel Thread Execution) を使用した．PTX を用いることで，より機械語に近い命令を直接記述することが出来る．double 型 64bit のシェアードメモリ読み込みを行うための命令はソースコード 1 のように記述できるようになる．なおソースコード中の reg は格納先のレジスタ，addr はアクセスするアドレスを格納したレジスタである．

ソースコード 1: シェアードメモリ読み込み命令

```
ld.shared.f64 reg, [addr]
```

3.2. クロックカウンタ

時間計測には GPU のクロックカウンタを利用した．ソースコード 2 のように，PTX の mov 命令を使用し，レジスタ%clock から値を取得することでクロックカウンタの値が得られる．

ソースコード 2: クロックカウンタの値を取得する命令

```
mov.u32 reg, %clock
```

ソースコード 3 は CUDA プログラム中に PTX 命令を記述できるインライン PTX を利用した計測用のソースコードである．処理内容は，まずスレッド間同期を行う bar.sync 命令で全スレッドが同時に計測を始められるようにする．そしてメモリアクセス前のクロックをクロックカウンタから取得し，シェアードメ

モリアクセスを行う．最後に再びクロックカウンタから処理後のクロックを取得する．

ソースコード 3: 実行クロックサイクル数の計測

```
asm volatile {
  "bar.sync 0;\n\t"
  "mov.u32 %0, %%clock;\n\t"
  "ld.volatile.shared.f64 %2, [%%a0];\n\t"
  "mov.u32 %1, %%clock;\n\t"
  :"=r"(begin), "=r"(end), "=l"(reg)
}
dummy[0] = reg;
```

こうして処理前後のクロックを取得した後，それらの差分を取ることでシェアードメモリアクセスに必要な実行クロックサイクル数を求めることができる．

ソースコード 3 からシェアードメモリアクセスを取り除き，クロックの取得が二回並ぶようなコードで計測を行い，差分を取ると 16clock cycle という結果が得られる．これは mov 命令でクロックカウンタの値を取得するのに必要なレイテンシと考えられるため，計測したシェアードメモリアクセスの実行クロックサイクル数からは引いておく必要がある．

なお，ソースコード 3 でメモリアクセス命令に volatile が指定されているのは PTX コードをコンパイルする際に，コンパイラによる最適化でメモリアクセスが消されてしまわないようにするためである．また計測後にシェアードメモリアクセスに用いたレジスタの値をメモリに書き込むことで依存関係を作っておくことも同様の理由で必要である．

3.3. 計測条件

クロックカウンタの取得によるシェアードメモリアクセス実行クロックサイクル数の計測を行う上で，レイテンシ，スループットに影響を与える三つの条件を考慮する必要がある．

一つ目は実行時のワープ数である．ワープ数が増加するとシェアードメモリアクセス命令の発行数が増加し，それだけアクセスを終えるまでのクロックサイクル数が増加する．クロックカウンタの取得はワープ単位で実行されるため，そのワープがメモリアクセスを開始したタイミング，及びメモリアクセスが完了したタイミングで個別にクロックの取得が行われる．アクセスを開始してから全てのワープがアクセスを終えるまでの時間を導出するため，図 4 のように処理前は最小値，処理後は最大値を求めた差を使用する．SM 当たりの最大スレッド数は 1024 スレッドであるため，ワープ数は 32 まで増加させて計測を行うことが出来る．

二つ目はメモリアクセス命令の実行回数である．ワープが実行するシェアードメモリアクセス命令が増えることで同様にクロックサイクル数が増加する．ソースコード 4 は命令実行回数が三回の場合のものである．格納先レジスタに別のレジスタを指定しているのは，コンパイラによる最適化で命令が一度しか実行されなくなるのを防ぐためである．

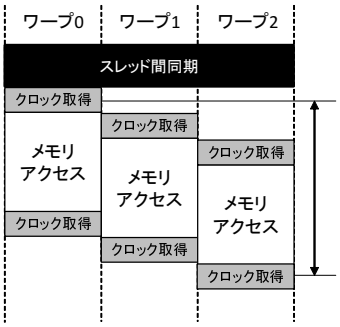


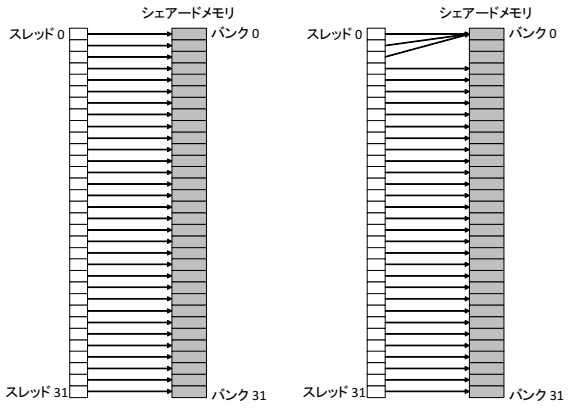
図 4: 複数ワープにおける実行クロックサイクル数

ソースコード 4: 命令実行回数が 3 回のときの測定

```
asm volatile {
  "bar.sync 0;\n\t"
  "mov.u32 %0, %%clock;\n\t"
  "ld.volatile.shared.f64 %2, [%%a0];\n\t"
  "ld.volatile.shared.f64 %3, [%%a0];\n\t"
  "ld.volatile.shared.f64 %4, [%%a0];\n\t"
  "mov.u32 %1, %%clock;\n\t"
  :"=r"(begin), "=r"(end), "=l"(reg0), "=l"(reg1),
       "=l"(reg2)
}
```

三つ目はバンクコンフリクトである．2.3 節で説明したように，バンクコンフリクトが生じるとアクセスが逐次化されクロックサイクル数が増加する．ここで，各バンクに同時にアクセスするスレッド数の最大値を congestion と定義する．バンクコンフリクトとシェアードメモリアクセス時間の関係は congestion の大きさに応じて決まる．図 5(a) はバンクコンフリクトが発生しないアクセスであり，各バンクへのアクセス数は全て 1 であるため，congestion は 1 となる．一方，図 5(b) はバンク 0 に対して 3 つのスレッドのアクセスが集中している．そのため congestion は 3 となり，メモリアクセスの逐次化によってアクセスの完了までにかかる時間が長くなってしまう．

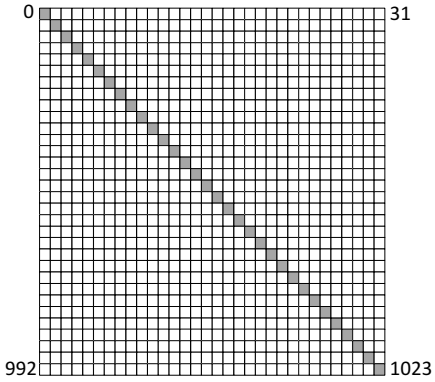
本研究ではバンク 0 に同時にアクセスするスレッド数を変化させることで congestion を操作しつつ計測を行う．しかし，シェアードメモリが同じアドレスにアクセスを行った場合，バンクコンフリクトは発生せずにアクセスがブロードキャストされる．そのため 1024 要素のシェアードメモリを用いて次のように congestion の操作を行う．図 6(a) は congestion が 1 のときのアクセスインデックスを表した図である．シェアードメモリの要素を，左上のインデックスが 0 として 32 要素ずつ行方向に並べ，アクセスするインデックスは灰色で示している．バンク番号は 32 要素ずつ割り当てられるため列方向に並んでいるメモリは全て同じバンクへのアクセスとなり，各列で一度しかアクセスが行われていない場合はバンクコンフリクトが発生しない．左上から対角線を引くようにインデッ



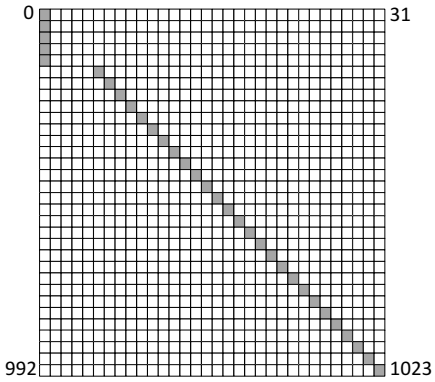
(a) congestion=1　　(b) congestion=3

図 5: シェアードメモリと congestion

クスを決定すると，列方向にアクセスが並ばず，バンクコンフリクトは発生しない．図 6(b) は congestion が 5 のときのアクセスインデックスを表している．5 つのスレッドのインデックスを左端の列，すなわちバンク 0 に集めることで congestion を大きくする．



(a) congestion=1



(b) congestion=5

図 6: アクセスインデックスと congestion

4. 計測結果

メモリアクセス時間の計測に用いた環境を表 2 に示す．得られた実行クロックサイクル数は 100 回計測を繰り返し行った平均を使用している．

表 2: 計測環境

GPU	GeForce GTX 780Ti
CUDA 環境	CUDA 5.5
計測する命令	ld.shared(シェアードメモリ読み込み)
アクセスサイズ	double 型 (64bit)

4.1. レイテンシの計測

命令実行回数，ワープ数，congestion 全てが 1 のとき，計測結果はシェアードメモリアクセスの最小レイテンシとなる．

実際に計測を行ったところ，16clock cycle という結果が得られた．グローバルメモリのレイテンシが 300〜400clock cycle であることと比較すると，バンクコンフリクトを考慮したシェアードメモリアクセスが非常に高速であることが分かる．

4.2. スループットの計測

シェアードメモリアクセスの実行クロックサイクル数に影響を与える命令実行回数，ワープ数，congestion を 3.3 節で説明したように変化させながら計測を行う．

シェアードメモリアクセスの条件は表 3 の範囲で計測を行った．

表 3: 計測条件

命令実行回数	1〜32
ワープ数	1〜32
congestion	1〜32

ここで命令実行回数を c，実行時のワープ数を w，congestion を c とおくことで，実行クロックサイクル数 $T(i,w,c)$ が定数 C_1 と C_2 を用いて式 (1) で表されると仮定する．

$$T(i,w,c) = C_1 \times iwc + C_2 \tag{1}$$

実際に計測を行い取得したデータを iwc と実行クロックサイクル数 T についてプロットを行ったグラフが図 7 である．

iwc と T がほぼ線形関係にあるため，計測結果に対して最小自乗法を用いることで線形回帰を行う．計測で得られた全データに対して最小自乗法による線形回帰を行うと，式 (2) が得られる．図 7 の赤色の直線は，この回帰式をプロットしたものである．線形回帰によって得られたこの式を用いることで，シェアードメモリアクセスの命令実行回数，ワープ数，congestion の値から実際にアクセスに必要な実行クロックサイクル数を予測することが出来る．

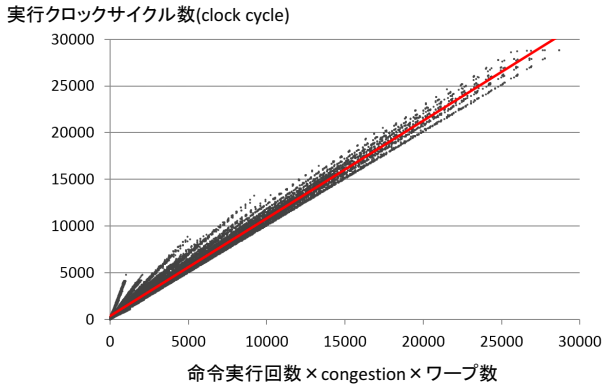


図 7: iwc と実行クロックサイクル数の関係と線形回帰式

$$T(i, w, c) = 1.047 \times iwc + 337.698 \qquad (2)$$

5. まとめ

NVIDIA 社の GPU に搭載されたオンチップ共有メモリであるシェアードメモリの実行クロックサイクル数を計測し，命令の実行回数，ワープ数，バンクコンフリクトとの関係を確認した．

GeForce GTX780Ti 上での計測の結果，バンクコンフリクトの生じない最適なアクセスにおけるレイテンシは 16clock cycle であることが分かった．また，様々な条件で計測したデータを用いて最小自乗法による線形回帰を行うことで，バンクコンフリクト，メモリアクセス命令数の実行回数，実行時のスレッド数とメモリアクセスに必要な実行時間の関係を表す回帰式を得ることが出来た．得られた式を用いることで，実際にアクセスに必要な実行クロックサイクル数を予測することが出来る．

参考文献

[1] NVIDIA. CUDA ZONE. http://www.nvidia.com/page/home.html.

[2] NVIDIA. Programming Guide :: CUDA Toolkit Documentation. http://docs.nvidia.com/cuda/cuda-c-programming-guide

[3] Wong, H. Demystifying GPU microarchitecture through microbenchmarking. Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on, pp.235-246, 28-30 March 2010

[4] Vasily Volkov, James W. Demmel. Benchmarking GPUs to Tune Dense Linear Algebra. SC '08 Proceedings of the 2008 ACM/IEEE conference on Supercomputing Article, No.31

[5] Ali Bakhoda, George L. Yuan, Wilson W.L. Fung, Henry Wong and Tor M. Aamondt: Analyzing CUDA Workloads Using a Detailed GPU Simulator, 2009 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS).

C2CU : A CUDA C Program Generator for Bulk Execution of a Sequential Algorithm

Daisuke Takafuji, Koji Nakano, and Yasuaki Ito

Department of Information Engineering
Hiroshima University
Kagamiyama 1-4-1, Higashi Hiroshima, 739-8527 Japan

Abstract. A sequential algorithm is oblivious if an address accessed at each time does not depend on input data. Many important tasks including matrix computation, signal processing, sorting, dynamic programming, and encryption/decryption can be performed by oblivious sequential algorithms. Bulk execution of a sequential algorithm is to execute it for many independent inputs in turn or in parallel. The main contribution of this paper is to develop a tool that generates a CUDA C program for the bulk execution of an oblivious sequential algorithm. More specifically, our tool automatically converts a C language program describing an oblivious sequential algorithm into a CUDA C program that performs the bulk execution of the C language program. Generated C programs can be executed in CUDA-enabled GPUs. We have implemented CUDA C programs for the bulk execution of bitonic sorting algorithm, Floyd-Warshall algorithm, and Montgomery modulo multiplication. Our implementations running on GeForce GTX Titan for the bulk execution can be 199 times faster for bitonic sort, 54 times faster for Floyd-Warshall algorithm, and 78 times faster for Montgomery modulo multiplication, over the implementations on a single Intel Xeon CPU.

Keywords: GPGPU, CUDA, bulk execution, oblivious algorithms, Floyd-Warshall algorithm, Montgomery modulo multiplication

1 Introduction

A Graphics Processing Unit (GPU) is a specialized circuit designed to accelerate computation for building and manipulating images [1–3]. Latest GPUs are designed for general purpose computing and can perform computation in applications traditionally handled by the CPU. Hence, GPUs have recently attracted the attention of many application developers [1, 4–7]. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [8], the computing engine for NVIDIA GPUs. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in NVIDIA GPUs. In many cases, GPUs are more efficient than multicore processors [9], since they have hundreds of processor cores and very high memory bandwidth.

CUDA uses two types of memories in the NVIDIA GPUs: *the shared memory* and *the global memory* [8]. The shared memory is an extremely fast on-chip memory with lower capacity, say, 16-48 Kbytes. The global memory is implemented as an off-chip DRAM, and thus, it has large capacity, say, 1.5-6 Gbytes, but its access latency is very long. The efficient usage of the shared memory and the global memory is a key for CUDA developers to accelerate applications using GPUs. In particular, we need to consider *the bank conflict* of the shared memory access and *the coalescing* of the global memory access [6, 9, 10]. The address space of the shared memory is mapped into several physical memory banks. If two or more threads access the same memory bank at the same time, the access requests are processed in turn. Hence, to maximize the memory access performance, CUDA threads should access the distinct memory banks to avoid the bank conflicts of the memory accesses. To maximize the bandwidth between the GPU and the DRAM chips, the consecutive addresses of the global memory must be accessed at the same time. Thus, CUDA threads should perform coalesced access and avoid stride access when they access the global memory. However, it is not an easy task for CUDA developers to design efficient parallel algorithms that does not perform stride memory access.

The bulk execution of a sequential algorithm is to execute it for many independent inputs in turn or in parallel. For example, suppose that we have p arrays $b_0, b_1, \ldots b_{p-1}$ of n points each. We can execute the Fourier transform of each b_j $(0 \leq j \leq p-1)$ by executing the FFT algorithm for n points on a single CPU in turn or on a parallel machine in parallel. The bulk execution of an FFT is frequently used in the area of image processing and signal processing. Further, the bulk execution is widely used in many applications. For example, plain text is partitioned into substrings with the same size when we encrypt it. The substrings are encrypted in turn to obtain encrypted text.

Intuitively, a sequential algorithm is *oblivious* if an address accessed at each time unit is independent of the input. For example, the prefix-sums of an array b of size n can be computed by executing $b[i] \leftarrow b[i]+b[i-1]$ for all i $(1 \leq i \leq n-1)$ in turn. This prefix-sum algorithm is oblivious because the address accessed at each time unit is independent of the values stored in b. The readers may think that the oblivious memory access is too restricted, and most useful algorithms are not oblivious. However, many important and complicated tasks including many matrix computations, signal processing, sorting, dynamic programming, and encryption/decryption can be performed by oblivious sequential algorithms.

In our previous paper [11], we have introduced an algorithmic technique performing the bulk execution of a sequential algorithm on the GPU and evaluated the performance using the Unified Memory Machine (UMM). The UMM is a theoretical parallel computing machine used to evaluate the performance of the computation on the GPU. The resulting implementation on the UMM performs the bulk execution for p independent inputs in $O(\frac{pt}{w} + lt)$ time units using p threads on the UMM if a sequential algorithm is oblivious, where w is the number of threads in a warp, l is the global memory access latency, and t is the running time of a sequential algorithm. It also proved that this implementation

is time optimal. Further, it implemented the prefix-sum algorithm and the dynamic programming algorithm using this algorithmic technique and obtained a speedup factor of 150 over the sequential computation by a single CPU. However, developers need to write CUDA C programs for the bulk execution of a sequential algorithm. Since it needs deep knowledge of CUDA programming and GPU architecture to optimize CUDA C programs, it is not an easy task to write efficient CUDA C programs for the bulk execution.

The main contribution of this paper is to present a tool, *C2CU*, that converts a sequential C program into a CUDA C program with no stride memory access. More specifically, a sequential program written by C programming language is given to C2CU. C2CU converts it into a CUDA C program that performs the bulk execution of a sequential program on CUDA-enabled GPUs. The CUDA C program thus obtained performs no stride global memory access of GPUs. Hence, even developers with few knowledge of CUDA C programming and GPU architecture can automatically generate a CUDA C program for the bulk execution. Once they write a C program for a sequential algorithm, they can obtain a CUDA C program for the bulk execution using our tool C2CU.

To see the performance of CUDA C programs generated by our C2CU converter, we have measured the running time of the bulk execution of three oblivious sequential algorithms: bitonic sort [12, 13], Floyd-Warshall algorithm [14–16], and Montgomery modulo multiplication [17–19]. For this purpose, we first have written sequential algorithms for these three algorithms by C programming language. We then have converted them into CUDA C programs using our C2CU converter. CUDA C programs thus obtained have been executed on GeForce GTX Titan. They run 199 times faster for bitonic sort, 54 times faster for Floyd-Warshall algorithm, and 78 times faster for Montgomery modulo multiplication, over the implementations on a single Intel Xeon CPU.

2 The bulk execution of sequential algorithms on the UMM

The main purpose of this section is to review the bulk execution of sequential algorithms on the Unified Memory Machine(UMM). Please see [11] for the details.

Intuitively, a sequential algorithm is *oblivious* if an address accessed in each time unit is independent of the input. More specifically, there exists a function $a : \{0, 1, \ldots, t-1\} \to \mathcal{N}$, where t is the running time of the algorithm and $\mathcal{N}$ is a set of all non-negative integers such that, for any input of the algorithm, it accesses address $a(i)$ or does not access the memory at each time i $(0 \leq i \leq t-1)$. In other words, at each time i $(0 \leq i \leq t-1)$, it never accesses an address other than $a(i)$.

Let us see an example of oblivious algorithms. Suppose that an array b of n integers are given. The prefix-sum computation is a task to store each i-th prefix-sum $b[0]+b[1]+\cdots+b[i]$ in $b[i]$. Let r be a register variable. The following algorithm computes the prefix-sum of n numbers.

[**Algorithm Prefix-sums**]
$r \leftarrow 0$
for $i \leftarrow 0$ to $n-1$ do
  $r \leftarrow r + b[i]$
  $b[i] \leftarrow r$

Since $b[0]$, $b[1]$, ..., $b[n-1]$ are added to r in turn, the prefix-sums are stored in b correctly when this algorithm terminates. Let us see the address accessed in each time unit to confirm that this algorithm is oblivious. For simplicity, we ignore access to registers and local computation such as addition and we assume that such operations can be done in zero time unit. Clearly, memory access operations performed in this algorithm are: read $b[0]$, write $b[0]$, read $b[1]$, write $b[1]$, ..., read $b[n-1]$, and write $b[n-1]$. Hence, the memory access function a is $a(2i) = a(2i+1) = i$ for all i $(0 \leq i \leq n-1)$, and thus, this algorithm is oblivious.

Suppose that we need to execute a sequential algorithm for many independent inputs on a single CPU in turn or on a parallel machine at the same time. We call such computation the *bulk execution*. For example, suppose that we have p arrays $b_0, b_1, \ldots, b_{p-1}$ of size n each on the UMM. The goal of the bulk execution of the prefix-sums is to execute the prefix-sums of every b_j $(0 \leq j \leq p-1)$ on the UMM in parallel. We use p threads and each thread j $(0 \leq j \leq p-1)$ executes the prefix-sums of b_i by Algorithm Prefix-sums. Let r_j $(0 \leq j \leq p-1)$ be a register of thread j. The prefix-sums can be computed in parallel by the following algorithm:

[**Parallel Algorithm Prefix-sums**]
for $j \leftarrow 0$ to $p-1$ do in parallel
  $r_j \leftarrow 0$
  for $i \leftarrow 0$ to $n-1$ do
    $r_j \leftarrow r_j + b_j[i]$
    $b_j[i] \leftarrow r_j$

In our previous paper [11], we have evaluated the running time of the bulk execution of the prefix-sums algorithm for column-wise arrangement on the Unified Memory Machine (UMM) [20, 21]. The UMM captures the essence of the global memory access of CUDA-enabled GPUs. The UMM has three parameters: the number p of threads, width w, and memory access latency l. Each thread is a Random Access Machine (RAM) [22], which can execute fundamental operations in a time unit. Threads are executed in SIMD [23] fashion, and run on the same program and work on the different data. The p threads are partitioned into $\frac{p}{w}$ groups of w threads each called *warp*. The $\frac{p}{w}$ warps are dispatched for the memory access in turn, and w threads in a dispatched warp send the memory access requests to the memory banks (MBs) through the memory management unit (MMU). We do not discuss the architecture of the MMU, but we can think that it is a multistage interconnection network in which the memory access requests are moved to destination memory banks in a pipeline fashion. Note that the UMM with width w has w memory banks and each warp has w threads.

MBs constitute a single address space of the memory. A single address space of the memory is mapped to the MBs in an interleaved way such that the word of data of address i is stored in the (i mod w)-th bank $B[i \bmod w]$, where w is the number of MBs. In the UMM, a single set of address lines from the MMU is connected to the MBs. Hence, the same address value is broadcast to every MB, and the same address of the MBs can be accessed at each time unit. Also, we assume that MBs are accessed in a pipeline fashion with latency l. In other words, if a thread sends a memory access request, it takes at least l time units to complete it. A thread can send a new memory access request only after the completion of the previous memory access request and thus, it can send at most one memory access request in l time units. Let $A[j] = \{j \cdot w, j \cdot w + 1, \ldots, (j + 1) \cdot w - 1\}$ denote the j-th address group. In the UMM, if multiple memory access requests by a warp are destined for different address groups, they are processed separately. Figure 1 illustrates the memory access by two warps $W(0)$ and $W(1)$. Since memory access requests by $W(0)$ are destined for three address groups, they occupy three pipeline stages. On the other hand, those by $W(1)$ are destined for the same bank, they occupy only one stages. Thus it takes $3(\text{stages}) + 1(\text{stage}) + 5(\text{pipeline stages}) - 1 = 8$ time units to complete memory access requests in Figure 1.

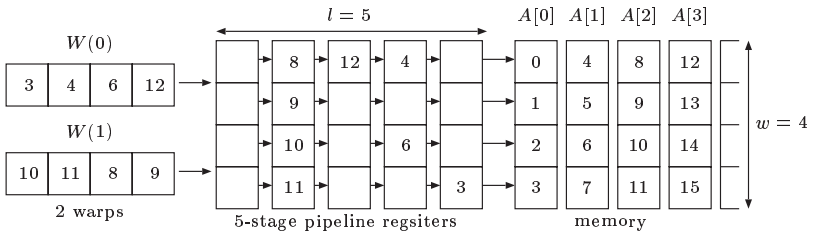


Fig. 1. The memory access of Unified Memory Machine (UMM) with width $w = 4$ and latency $l = 5$

Suppose that each element $b_j[i]$ $(0 \leq i \leq n-1, 0 \leq j \leq p-1)$ is arranged in address $i \cdot p + j$ of the global memory as illustrated in Figure 2. Suppose that the bulk execution of an oblivious algorithm running in t time units is performed for p inputs with column-wise arrangement on the UMM. Clearly, pt memory access operations are performed at all and all memory access operations by all warps are coalesced. Also, each thread on the UMM performs t memory access operations, each of which takes l time units. Thus, we have the following theorem:

Theorem 1 ([11]). *A column-wise oblivious computation of size $n \times p$ runs $O(\frac{pt}{w} + lt)$ time units using p threads on the UMM with width w and latency l, where t is the running time of the corresponding oblivious sequential algorithm.*

Please see [11] for the details of the proof of Theorem 1.

0 $b_0[0]$	1 $b_1[0]$	2 $b_2[0]$	3 $b_3[0]$	4 $b_4[0]$	5 $b_5[0]$	6 $b_6[0]$	7 $b_7[0]$
8 $b_0[1]$	9 $b_1[1]$	10 $b_2[1]$	11 $b_3[1]$	12 $b_4[1]$	13 $b_5[1]$	14 $b_6[1]$	15 $b_7[1]$
16 $b_0[2]$	17 $b_1[2]$	18 $b_2[2]$	19 $b_3[2]$	20 $b_4[2]$	21 $b_5[2]$	22 $b_6[2]$	23 $b_7[2]$
24 $b_0[3]$	25 $b_1[3]$	26 $b_2[3]$	27 $b_3[3]$	28 $b_4[3]$	29 $b_5[3]$	30 $b_6[3]$	31 $b_7[3]$
32 $b_0[4]$	33 $b_1[4]$	34 $b_2[4]$	35 $b_3[4]$	36 $b_4[4]$	37 $b_5[4]$	38 $b_6[4]$	39 $b_7[4]$
40 $b_0[5]$	41 $b_1[5]$	42 $b_2[5]$	43 $b_3[5]$	44 $b_4[5]$	45 $b_5[5]$	46 $b_6[5]$	47 $b_7[5]$

Fig. 2. Column-wise arrangement of $p = 8$ arrays of $n = 6$ elements each

3 Our C2CU converter

The main purpose of this section is to describe C2CU converter, that converts a sequential algorithm written by C programming language into CUDA C program for the bulk execution on CUDA-enabled GPUs.

Figure 3 illustrates the behavior of C2CU converter. A sequential program written by C programming language is converted into a CUDA C program. The converted C program accepts p independent inputs. They are copied to the device memory (global memory) of the GPU. The CUDA device program with p threads is spawned, and each thread executes the sequential program for one input. After all threads terminate, p outputs obtained by all threads are copied to the host memory.

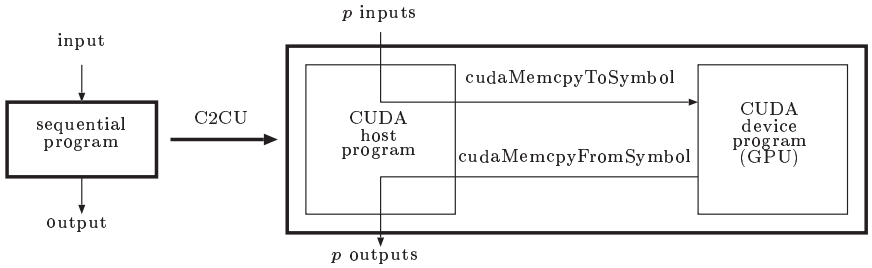


Fig. 3. The behavior of C2CU converter

Let us see how C2CU converter generates CUDA C program using Floyd-Warshall algorithm [14–16] as an example. Floyd-Warshall algorithm is a well

known graph theoretic algorithm that computes the distances of the shortest paths of all pairs of nodes in a directed graph. It uses a 2-dimensional array D of size $n \times n$ for an n-node graph. We assume that, initially, $D[i][j]$ $(0 \leq i, j \leq n-1)$ stores the distance of an edge from node i to j if it exists and $+\infty$ otherwise. Floyd-Warshall algorithm is described as follows:

[**Algorithm Floyd-Warshall**]
for $k \leftarrow 0$ to n do
  for $i \leftarrow 0$ to n do
    for $j \leftarrow 0$ to n do
      if ($D[i][j] > D[i][k] + D[k][j]$)
        $D[i][j] \leftarrow D[i][k] + D[k][j]$

After termination of the algorithm, $D[i][j]$ stores the distance of the shortest path from node i to j. If there is no such path, it stores $+\infty$.

Figure 4 shows a C program for Floyd-Warshall algorithm. It should be clear that this C program computes the all-pairs shortest distance by Floyd-Warshall algorithm. The values of D is updated by calling `update_dist`, although it is not necessary to be a function. The reason is to show our C2CU converter supports function calls. The C program in Figure 4 is a direct implementation of Floyd-Warshall algorithm except that it has a directive `#pragma kernel` in line 22. Most C compilers such as GNU C compiler ignores this directive. Hence, this C program can be compiled correctly, and it computes all-pairs shortest distance in an input graph by Floyd-Warshall algorithm. A directive `#pragma kernel` is used to specify a function for the bulk execution on the GPU. A function call just after directive `#pragma kernel` will be executed on the GPU in the CUDA C program obtained by C2CU.

Figure 5 shows a CUDA C program generated by our tool C2CU from the C program in Figure 4. Users can specify the number p of inputs (i.e. the number p of threads) and the number of threads in each CUDA block, by using options for C2CU. These values are defined as `__P__` $(= p)$ and `__T__` in lines 2 and 3. In Figure 5, they are 2048 and 64, respectively. Thus, 32 CUDA blocks with 64 threads each are spawned by CUDA kernel call `floyd_warshall<<<__B__,__T__>>>()` in line 31. Since the generated CUDA C program accepts p inputs, a 3-dimensional array D of size $N \times N \times p$ allocated in the host memory are used to store them. Also, a 3-dimensional array `__D` of the same size allocated in the device memory (i.e. the global memory of the GPU) are used. In line 30, cudaMemcpyToSymbol is used to copy p inputs stored in D to `__D`. After the bulk execution by CUDA kernel call `floyd_warshall<<<__B__,__T__>>>()` in line 31, cudaMemcpyToSymbol is used to copy `__D`, which stores the resulting values, to D.

CUDA kernel call `floyd_warshall<<<__B__,__T__>>>()` in line 31 invokes `__B__` CUDA blocks with `__T__` threads each. Thus, `__P__` $(= p)$ threads execute Floyd-Warshall algorithm on the CUDA-enabled GPU. Since `blockDim.x` is the number `__B__` of threads in a CUDA block and `blockIdx.x` and `threadIdx.x` take values in $[0, \texttt{\_\_B\_\_} - 1]$ and $[0, \texttt{\_\_T\_\_} - 1]$, respectively, `__id__` in line 15 takes value from 0 to $p-1$. Hence device function `update_dist(i,j,k,__id__)` is executed for `__id__` in $[0, p-1]$ on the GPU in parallel. The reader should

```
 1: #define N 1024
 2: float D[N][N];
 3: void update_dist(int i, int j, int k){
 4:   if( D[i][j] > D[i][k] + D[k][j] ) {
 5:     D[i][j] = D[i][k] + D[k][j];
 6:   }
 7: }
 8:
 9: void floyd_warshall(){
10:   int i,j,k;
11:   for(k=0;k<N;k++) {
12:     for(i=0;i<N;i++) {
13:       for(j=0;j<N;j++) {
14:         update_dist(i,j,k);
15:       }
16:     }
17:   }
18: }
19:
20: int main(int argc, char *argv[]){
21:   input_array();
22: #pragma kernel
23:   floyd_warshall();
24:   ...
```

Fig. 4. A C program of the Floyd-Warshall algorithm

have no difficulty to confirm that CUDA C program in Figure 5 executes Floyd-Warshall algorithm for p inputs in parallel.

Let us see how C2CU converts a C program into a CUDA C program for general cases and confirm that the generated CUDA C programs performs coalesced memory access. If an original C program uses d dimensional array a of size $s_1 \times s_2 \times \cdots \times s_d$, a CUDA C program generated by C2CU uses $d+1$ dimensional array a of size $s_1 \times s_2 \times \cdots \times s_d \times p$. If the original C program accesses $a[i_1][i_2]\cdots[i_d]$ then each thread with ID id of the corresponding CUDA C program accesses $a[i_1][i_2]\cdots[i_d]$[__id__]. Since $a[i_1][i_2]\cdots[i_d][0]$, $a[i_1][i_2]\cdots[i_d][1]$, ..., $a[i_1][i_2]\cdots[i_d][p-1]$ are allocated in consecutive addresses, these memory accesses by p threads are coalesced.

4 Experiment results

The main purpose of this section is to show experimental results on GeForce GTX Titan. GeForce GTX Titan has 14 streaming multiprocessors with 192 cores each. Hence, it can run 2688 threads in parallel. Note that, a single kernel call to GeForce GTX Titan can run more than 2688 threads in a time sharing manner using CUDA [8] parallel programming platform. All input and output

```
 1: #define N 1024
 2: #define __P__ 2048
 3: #define __T__ 64
 4: #define __B__  __P__/__T__
 5: float D[N][N][__P__];
 6: __device__ float __D[N][N][__P__];
 7:
 8: __device__ void update_dist(int i, int j, int k, int __id__){
 9:   if( __D[i][j][__id__] > __D[i][k][__id__] + __D[k][j][__id__] ) {
10:     __D[i][j][__id__] = __D[i][k][__id__] + __D[k][j][__id__];
11:   }
12: }
13:
14: __global__ void floyd_warshall(){
15:   int __id__ = blockIdx.x * blockDim.x + threadIdx.x;
16:   int i,j,k;
17:   for(k=0;k<N;k++) {
18:     for(i=0;i<N;i++) {
19:       for(j=0;j<N;j++) {
20:         update_dist(i,j,k,__id__);
21:       }
22:     }
23:   }
24: }
25:
26: int main(int argc, char *argv[])
27: {
28:   input_array();
29: #pragma kernel
30:   cudaMemcpyToSymbol(__D, D, sizeof(float)*N*N*__P__, 0);
31:   floyd_warshall<<<__B__,__T__>>>();
32:   cudaMemcpyFromSymbol(D, __D, sizeof(float)*N*N*__P__, 0);
33:   ...
```

Fig. 5. A CUDA program for the bulk execution of Floyd-Warshall algorithm generated by C2CU

data are stored in the global memory of the GPU and we do not use the shared memory of the streaming multiprocessors.

We have used three sequential algorithms as follows:

- bitonic sort [12, 13],
- Floyd-Warshall algorithm [14–16], and
- Montgomery modulo multiplication [17–19].

Bitonic sort is a well-known parallel sorting algorithm developed by K.E. Batcher [12]. It can be described as a sorting network with comparators as illustrated in Figure 6. Since elements compare-exchanged in each stage are fixed, bitonic sort can be written as an oblivious sequential algorithm.

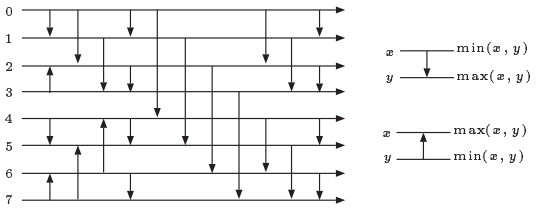


Fig. 6. Bitonic sort for $n = 8$

Montgomery modulo multiplication is used to speed the modulo multiplication $X \cdot Y \cdot 2^{-R} \bmod M$ for R-bit numbers X, Y, and M. The idea of Montgomery modulo multiplication is not to use direct modulo computation, which is very costly in terms of the computing time and hardware resources. By iterative computation of Montgomery modulo multiplication, the modulo exponentiation $P^E \bmod M$ can be computed, which is a key operation for RSA encryption and decryption [24]. Since R is at least 1024 to use Montgomery modulo multiplication for RSA encryption and decryption, addition/multiplication is repeated to perform R-bit addition/multiplication. Figure 7 illustrates how the product $a \cdot b$ of two integers a and b of large bits is computed. Both a and b are partitioned into four integers and the sum of pair-wise products is computed. Using this idea, we can design an oblivious sequential algorithm to compute the product of two integers with large bits in an obvious way. Since Montgomery modulo multiplication repeats computation of the product and the sum of two large integers, it can also be computed by an oblivious sequential algorithm.

We have written a C program for bitonic sort that sorts $n = 32$, 1K (= 1024), and 32K (= 32768) float (32-bit) numbers. We have converted into a CUDA C program for the bulk execution of bitonic sort with parameter $p = 64, 128, \ldots, 4\text{M}$. However, due to the global memory capacity of the GPU, it is executed for up to $p = 128\text{K}$ and $p = 4\text{K}$ when $n = 1\text{K}$ and $n = 32\text{K}$, respectively. The CUDA C program invokes p threads in $\frac{p}{64}$ CUDA blocks with

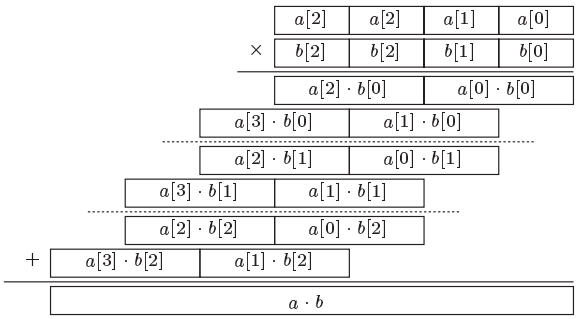
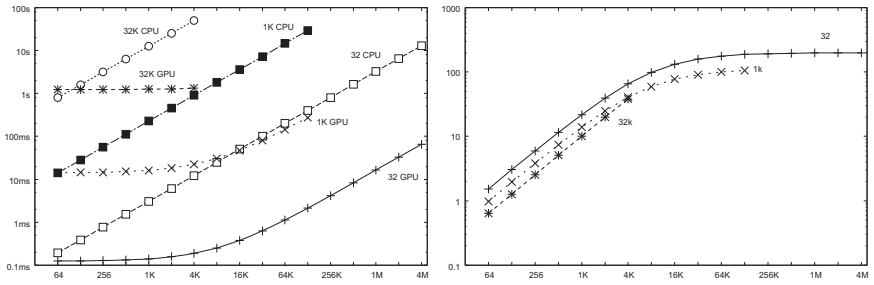


Fig. 7. Multiplication of two integers with large bits

64 threads each to sort p inputs of n numbers each. To see the speedup factor, the original C program is repeatedly executed p times on the Intel Xeon (2.66GHz)

Figure 8 (1) shows the resulting computing time for the bulk execution of bitonic sort. Recall that, from Theorem 1, the bulk execution of a sequential algorithm can be computed in $O(\frac{pt}{w}+lt)$ time units, where p is the total number of threads, l is the memory access latency, and t is the running time of the original sequential algorithm. The bulk execution of bitonic sort for $n = 32$ takes about 0.13ms when $p \leq$1K. Further, the computing time is proportional to p when $p \geq$16K and it runs 65.1ms when $p =$ 4M. Thus, we can think that $O(lt) =$ 0.13ms and $O(\frac{pt}{w}) = (15.5p)$ns. More specifically, the bulk execution of bitonic sort for $n = 32$ and p can be computed in approximately 0.13ms+$(15.5p)$ns. Figure 8 (2) shows the speedup factor of the GPU over the CPU. We can see that the bulk execution of bitonic sort on the GPU can achieve a speedup of factor more than 180 when $n = 32$ and $p \geq$ 128K. Further, when $n = 32$ and $p =$ 4M, the GPU is 199 times faster than the CPU.

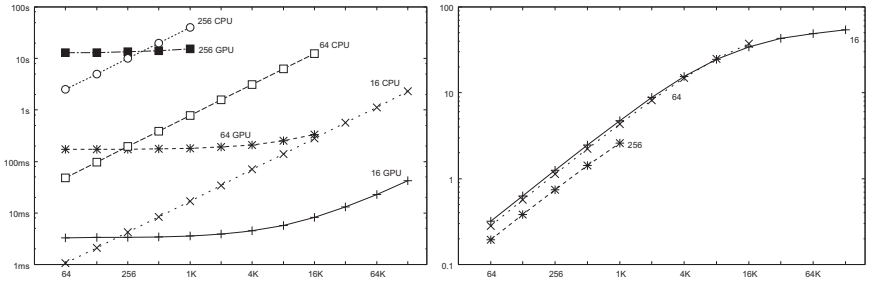


(1) The computing time (2) GPU/CPU speedup factor

Fig. 8. The computing time (ms) of bitonic sort on CPU and GPU, and the speedup for $n = 32$, 1K, 32K, and $p = 64$, 128, ..., 4M.

We have written a C program for Floyd-Warshall algorithm for graphs with $n = 16$, 64, and 256 nodes. We use float (32-bit) numbers to store the length of each edge. The C program is converted into a CUDA C program using C2CU with parameters $p = 16$, 64, and 256. However, due to the global memory capacity of the GPU, it is executed for up to $p = 16$K and $p = 1$K when $n = 64$ and $n = 256$, respectively.

Figure 9 (1) shows the resulting computing time for the bulk execution of Floyd-Warshall algorithm. We will verify $O(\frac{pt}{w}+lt)$ time units shown in Theorem 1. The bulk execution of Floyd-Warshall algorithm for $n = 16$ takes about 3.4ms when $p \leq 512$. Also, the computing time is proportional to p when $p \geq 4$K and it runs 42.6ms when $p = 128$K. Thus, we can think that $O(ln^3) = 3.4$ms and $O(\frac{pn^3}{w}) = (325p)$ns. More specifically, the bulk execution of the Floyd-Warshall algorithm for $n = 32$ and p can be computed in approximately 3.4ms+(325)ns. Figure 9 (2) shows the speedup factor of the GPU over the CPU. We can see that the bulk execution on the GPU can achieve a speedup of factor more than 30 when $n = 16$ and $p \geq 8$K. Further, when $n = 16$ and $p = 128$K, the GPU is 54 times faster than the CPU.



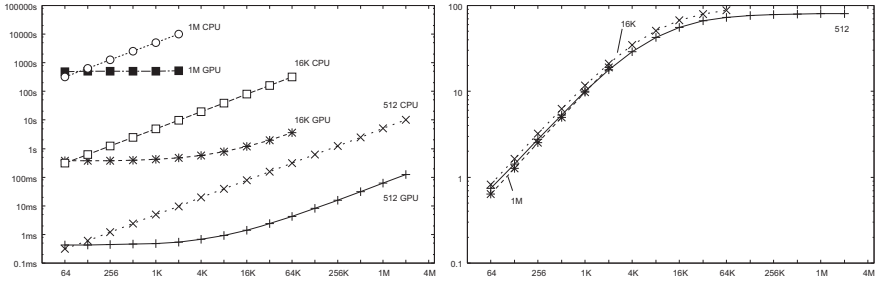
(1) The computing time (2) GPU/CPU speedup factor

Fig. 9. The computing time (ms) of the Floyd-Warshall algorithm on CPU, and GPU and the speedup for $n = 16$, 64, 256, and $p = 64$, 128, ..., 128K

Finally, we have written a C program for Montgomery modulo multiplication for $n = 512$, 16K (= 16384), and 1M (= 1048576) bits. We use C2CU to convert it into a CUDA C program with parameter $p = 64, 128, \ldots, 2$M. However, due to the global memory capacity, it is executed for up to $p = 64$K and $p = 2$K when $n = 16$K and $n = 1$M, respectively.

Figure 10 (1) shows the resulting computing time for the bulk execution of the Montgomery modulo multiplication. Again, we will verify $O(\frac{pt}{w} + lt)$ time units shown in Theorem 1. The bulk execution of the algorithm for $n = 512$ takes about 0.45ms when $p \leq 512$. Also, the computing time is proportional to p when $p \geq 128$K and it runs 124ms when $p = 2$M. Thus, we can think that $O(ln^2) = 0.45$ms and $O(\frac{pn^2}{w}) = (59.1p)$ns. More specifically, the bulk execution

of the algorithm for $n = 512$ can be computed in approximately 124ms+$(5.9p)$ns. Figure 9 (2) shows the speedup factor of GPU computation using the GPU over the CPU. We can see that the GPU can achieve a speedup of factor more than 70 when $n = 512$ and $p \geq 32$K. Further, when $n = 512$ and $p = 2$M, the GPU is 78 times faster than the CPU.



(1) The computing time (2) GPU/CPU speedup factor

Fig. 10. The computing time (ms) of the Montgomery modulo multiplication on CPU, and GPU. and the speedup for $p = 64, 128, \ldots, 4$M

5 Conclusion

The main contribution of this paper is to develop C2CU converter, which converts a C language program of a sequential algorithm into a CUDA C program for the bulk execution on the GPU. The experimental results show that the generated CUDA C program on GeForce GTX Titan can achieve up to 199 times speed-up over the original C program running on an Intel Xeon CPU. Thus, C2CU is a promising tool to obtain high GPGPU acceleration very easily.

References

1. Hwu, W.W.: GPU Computing Gems Emerald Edition. Morgan Kaufmann (2011)
2. Man, D., Uda, K., Ito, Y., Nakano, K.: A GPU implementation of computing Euclidean distance map with efficient memory access. In: Proc. of International Conference on Networking and Computing. (Dec. 2011) 68–76
3. Uchida, A., Ito, Y., Nakano, K.: Fast and accurate template matching using pixel rearrangement on the GPU. In: Proc. of International Conference on Networking and Computing, IEEE CS Press (Dec. 2011) 153–159
4. Ogawa, K., Ito, Y., Nakano, K.: Efficient Canny edge detection using a GPU. In: Proc. of International Conference on Networking and Computing, IEEE CS Press (Nov. 2010) 279–280

5. Nishida, K., Ito, Y., Nakano, K.: Accelerating the dynamic programming for the matrix chain product on the GPU. In: Proc. of International Conference on Networking and Computing. (Dec. 2011) 320–326
6. Nishida, K., Ito, Y., Nakano, K.: Accelerating the dynamic programming for the optial poygon triangulation on the GPU. In: Proc. of International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP, LNCS 7439). (Sept. 2012) 1–15
7. Uchida, A., Ito, Y., Nakano, K.: An efficient GPU implementation of ant colony optimization for the traveling salesman problem. In: Proc. of International Conference on Networking and Computing, IEEE CS Press (Dec. 2012) 94–102
8. NVIDIA Corporation: NVIDIA CUDA C programming guide version 5.0 (2012)
9. Man, D., Uda, K., Ueyama, H., Ito, Y., Nakano, K.: Implementations of a parallel algorithm for computing euclidean distance map in multicore processors and GPUs. International Journal of Networking and Computing **1**(2) (July 2011) 260–276
10. NVIDIA Corporation: NVIDIA CUDA C best practice guide version 3.1 (2010)
11. Tani, K., Takafuji, D., Nakano, K., Ito, Y.: Bulk execution of oblivious algorithms on the unified memory machine, with gpu implementation. In: Proc. of International Parallel and Distributed Processing Symposium Workshops. (May 2014) 586–595
12. Batcher, K.E.: Sorting networks and their applications. In: Proc. AFIPS Spring Joint Comput. Conf. Volume 32. (1968) 307–314
13. Akl, S.G.: Parallel Sorting Algorithms. Academic Press (1985)
14. Floyd, R.W.: Algorithm 97: Shortest path. Communications of the ACM **5**(6) (June 1962) 345
15. Warshall, S.: A theorem on boolean matrices. Journal of the ACM **9**(1) (Jan. 1962) 11–12
16. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. MIT Press (1990)
17. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation **44**(170) (1985) 519–521
18. Shigemoto, K., Kawakami, K., Nakano, K.: Accelerating montgomery modulo multiplication for redundant radix-64k number system on the FPGA using dual-port block RAMs. In: Proc. of International Conference on Embedded and Ubiquitous Computing(EUC). (2008) 44–51
19. Bo, S., Kawakami, K., Nakano, K., Ito, Y.: An RSA encryption hardware algorithm using a single DSP block and a single block RAM on the fpga. International Journal of Networking and Computing **1**(2) (July 2011) 277–289
20. Nakano, K.: Simple memory machine models for GPUs. International Journal of Parallel, Emergent and Distributed Systems **29**(1) (2014) 17–37
21. Nakano, K.: Sequential memory access on the unified memory machine with application to the dynamic programming. In: Proc. of International Symposium on Computing and Networking. (Dec. 2013) 85–94
22. Aho, A.V., Ullman, J.D., Hopcroft, J.E.: Data Structures and Algorithms. Addison Wesley (1983)
23. Flynn, M.J.: Some computer organizations and their effectiveness. IEEE Transactions on Computers **C-21** (1972) 948–960
24. Blum, T., Paar, C.: High-radix montgomery modular exponentiation on reconfigurable hardware. IEEE Trans. on Computers **50**(7) (2001) 759–764

GPUを用いた高スループット計算システムの実装

谷　和也
広島大学大学院工学研究科

概 要

近年，膨大なデータの処理は現代のコンピュータ・システムの課題として広く知られている．本研究では，膨大なデータに対して処理を行う際の高スループットな計算システムの実装を提案する．実装にはGPUとNVIDIA社が提供するGPU向けの統合開発環境であるCUDA[1]を使用する．GPU(Graphics Processing Units)とはグラフィクス処理用に開発されたハードウェアであり，多数の演算コアを搭載している．これを汎用演算に利用し，システムを実現する．本システムでは膨大なデータに対しては分割して処理を行う．性能の評価として膨大な数の32点FFTを計算するプログラムを作成し，本システム上で実行したときの結果を考察する．GPUとしてGeforce GTX TITANを使用して構築した本システムでは50ギガバイトの膨大な32点FFT入力データに対して，32MBごとに分割して処理したときが総処理時間12799.53ms，スループットは3.91GB/secとなり，処理時間とスループットのバランスに優れていた．

1. はじめに

GPU(Graphics Processing Units)とは多数のプロセッサコアが搭載されたハードウェアである．多数のコアで並列計算が可能で，非常に高い演算能力を有している．もとはグラフィクス処理用に開発されたハードウェアであったが，近年ではその演算能力を汎用演算に用いるGPGPU(General-Purpose computing on Graphics Processing Units)と呼ばれる試みが盛んに行われており，GPGPU向けのGPUが多く開発されている．

本研究で用いたCUDAとはNVIDIA社が提供するGPGPU向けの統合開発環境である．GPUに対応したソフトウェアアーキテクチャであり，GPU上の多数のコアに対してスレッドを割り当て並列計算を行う．複数のスレッドの集合はブロックと呼ばれ，GPUのストリーミングマルチプロセッサに割り当てられる．GPUには全てのスレッドが共有するグローバルメモリと，同じブロック内でのみ共有するシェアードメモリが存在する．本研究では，膨大なデータ処理のため，比較的大容量であるグローバルメモリを使用して実装を行う．

CUDAを使用して開発したプログラムはCPUによる処理部分を記述したコードと，GPUによる処理部分を記述したコードからなる．GPU上で実行される関数はカーネルと呼ばれ，CPU側のコードから呼び出される．GPUのメモリ領域はCPUから独立しているため，カーネルが処理するデータはGPUメモリ上へと転送してやる必要がある．当然，処理終了後に結果を得るためには，同様にGPUメモリからCPUメモリ上へと転送しなければならない．以上のことから，GPUを用いた処理は以下の3ステップで行われる．

- データ転送 (HostToDevice)
- カーネル実行 (Kernel)
- データ転送 (DeviceToHost)

ここではCPUはホストと呼ばれ，GPUはデバイスと呼ばれている．すなわち，HostToDeviceとはCPUからGPUへのデータを転送を意味する．全体的な手順としては，まず入力データがCPU側で作成されGPUへと転送される．その後CPU側からカーネルが呼び出され，GPUのグローバルメモリにあるデータに対して処理が行われる．そして得られた結果は再びCPUへと送り返される．

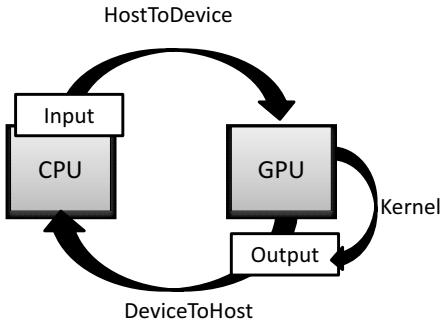


図 1: CPU-GPU プログラムの概要

2. CUDA アーキテクチャ

図2にGPUのハードウェアアーキテクチャを示す．GPUにはSM(Streaming Multiprocessor)が複数あり，SMには多数のプロセッサコアが存在している．各SM内にあるシェアードメモリは同一のSM内でのみ共有可能なメモリ領域で，アクセスが非常に速い代わりに容量が小さいという特徴を持っている．一方で各SMが接続されているグローバルメモリは全てのプロセッサコアで共有が可能なメモリである．ただし，グローバルメモリはシェアードメモリと比べてアクセスのオーバーヘッドが大きいため，効率的なアク

セスをするのが重要である．

GPU のハードウェアアーキテクチャに対応するソフトウェアアーキテクチャが CUDA である．図 3 に CUDA アーキテクチャを示す．CUDA はスレッド，ブロック，グリッドが GPU と対応して階層構造になっている．スレッドが処理の最小単位であり，GPU におけるプロセッサコアに対応する．同様にスレッドの集合であるブロックは SM に対応し，グリッドは GPU 全体と対応している．

グローバルメモリアクセス

本研究では膨大な数のデータを扱うため，大容量のグローバルメモリを使用している．グローバルメモリアクセスでは各スレッドが連続したメモリ領域へアクセスする場合は一度でアクセスすることができる．これはコアレスドアクセスと呼ばれている (図 4)．したがって CPU から GPU へと入力データが渡される際に，メモリアクセスがコアレスドアクセスとなるようなデータに並べ替えてから転送している．この並び替えの方法が図 10 にあるが，これについてはまた第 3. 章で詳しく説明する．

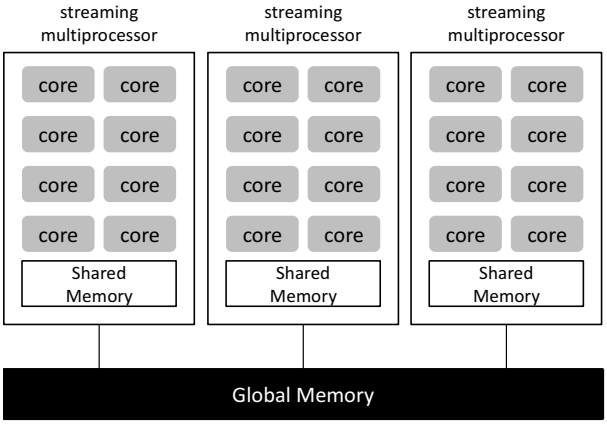


図 2: GPU アーキテクチャ

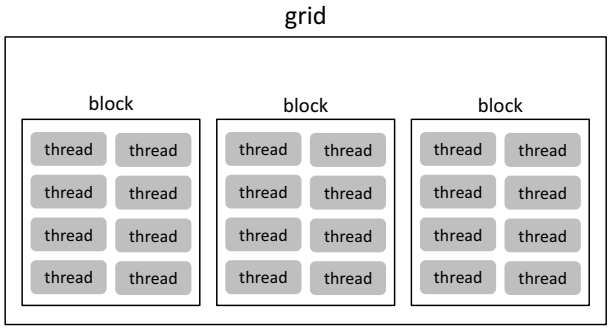


図 3: CUDA アーキテクチャ

ストリーム

CUDA におけるストリームには GPU の処理を管理するキューとしての役割がある．デフォルトではストリームは 1 つのみ用意されており，GPU へ発行した命令は発行された順に従って逐次的に実行されることが保証されている．また，ストリームは複数生成することができ，プログラマが任意のストリームに命令を投入していくことが可能になっている．ストリーム間で異なる命令を処理することができるため，分割した各データに対するメモリコピー及びカーネル実行を各ストリームに割り当てれば別々に結果を得ることも可能である．また，今回の研究には使用していないが，ストリームにはもっと発展的な機能が備わっている．それは，ストリーム同士には依存関係がなく独立に処理されるため，非同期に実行が可能でストリーム同士の並列処理が可能であるということである．これにより，処理をオーバーラップすることで全体の処理時間をさらに短縮できる．しかしオーバーラップの可能性はハードウェアに依存し，オーバーラップした場合は処理の順序が複雑化するため，これについては今後の課題として考えている．

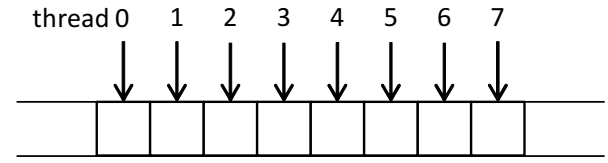


図 4: コアレスドアクセス

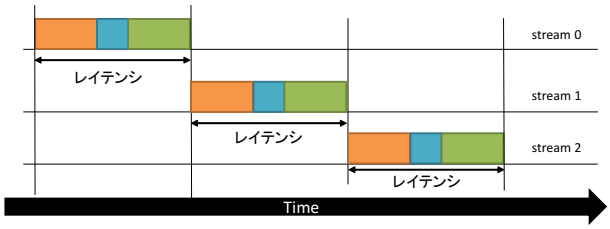


図 5: ストリームを利用したデータの分割処理

3. 実装方法

本研究では膨大な数のデータが外部のソースから計算を行うデバイスに送信されてくる状況を想定し，そのような状況下で高いスループットを保ったまま，効率よく出力を得るためのシステムを提案する．外部のデータソースを CPU，計算を行うデバイスを GPU で構築する．このシステムでは通常の GPU を用いた演算手法ではデータの数が多いほど転送時間と処理時間が長くなるため効率が良いとはいえない．膨大なデータの転送と処理を一括で行うのは結果を得るまでに時間がかかることになる．ここで，GPU への転送開始から計算結果の CPU 転送が終了するまでの時間をレイテンシと呼ぶものとする．

そこで，分割した入力データを一定の周期で GPU に転送するようなプログラムを作成することで，CPU から GPU へデータが流れてくる状況を実現した．図

6 は一定の周期でデータが転送されるプログラムの概要を表している．左の時間軸は CPU のタイムラインであり，このタイムラインにおいて 10ms ごとにコールバック関数が呼び出されている．コールバック関数は第 1. 章で紹介した通り，GPU の処理の基本である 3 ステップを含んでいる．この 3 ステップの GPU 処理が完了したらコールバック関数を終了し，次の周期が来るまでは CPU はスリープさせておく．データの到着周期よりも GPU の処理にかかる時間が長くなると，現実のシステムにおいて入力されてくるデータに対して処理が間に合わなくなり，エラーが発生する可能性がある．したがって，本研究のシステムでは GPU 処理にかかる時間が周期よりも長くならないように，GPU の処理の実行時間を予め計測しておき，その時間を例外なくカバーできるような周期を設定する必要があった．図 7 および図 8 に分割サイズが 32MB と 128MB の 2 つにおいて全レイテンシをグラフ化したものを示す．このグラフからレイテンシに大きく外れる値はほとんどないとわかる．また，32MB よりも 128MB のほうが値の振れ幅が小さいため，データを分割するサイズが大きい場合，レイテンシは安定した値を計測できる．このようにして全ての分割サイズについて求めた周期は表 1 にある．

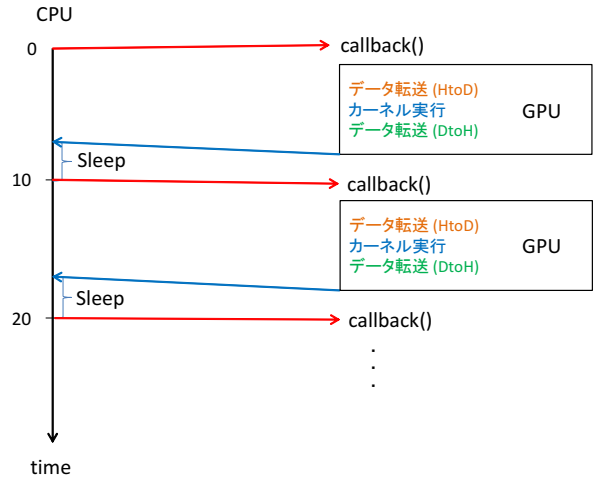


図 6: 周期 10ms でデータが GPU へ転送されるプログラム

分割サイズ (MB)	1	2	4	8	16	32	64	128	256	512	1024
周期 (ms)	2	2	2	3	5	8	16	30	60	120	240

表 1: 分割サイズごとのコールバック関数を呼ぶ周期

このシステムでは分割されたデータごとに処理が行われるため，計算結果を細かく得ることが可能となる．しかし，分割するサイズによっては良い結果をもたらさない可能性が考えられる．例えば，小さく分割し過ぎた場合は実行時間の短い多数のカーネルを実行することになり，カーネル実行のオーバーヘッドにより全体の実行時間が長くなることがある．全体の実行時間やスループットのパフォーマンスの低下を抑えつつ，レイテンシを短縮できるような効率的な分割サイズを発見するため，FFT のプログラムを用いて実験を行った．

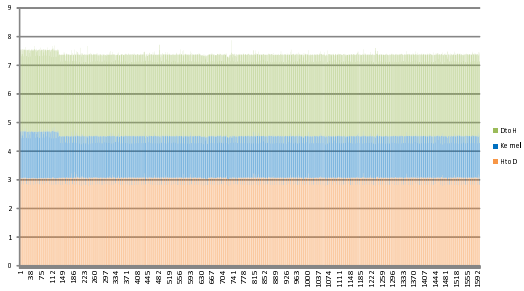


図 7: 分割サイズ 32MB のときのレイテンシ

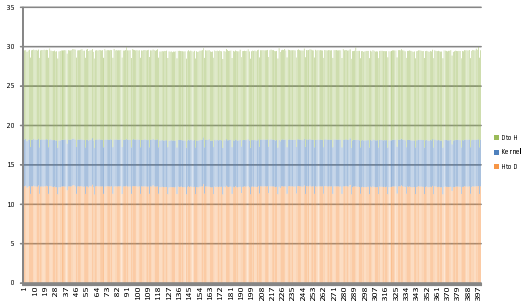


図 8: 分割サイズ 128MB のときのレイテンシ

32 点の FFT を CUDA の各スレッドで実行するプログラムを作成し，このプログラムに対して膨大な数の入力データを与えて実験を行った．具体的には $4*10^6$ 個の FFT の入力データ (1GB) を 1 つの配列内に用意し，これを 50 回繰り返し利用することで $200*10^6$ 個の FFT 入力データとして使用している．この入力データ配列 (1GB) を CUDA のストリームという機能を用いて分割して処理することでレイテンシの短縮を試みる．様々な分割サイズで時間の計測を行い，パフォーマンスを確認した結果，データを適切に分割すれば全体の実行時間をほとんど延ばすことなくスループットについても高い値を保つことができた．

CUDA による FFT の実装

高速フーリエ変換 (FFT) は離散フーリエ変換を高速に計算するアルゴリズムである [3]．本研究では 1 次元 FFT である Cooley-Tukey 型 FFT と呼ばれるアルゴリズム [2] の実装を行った．ここでは FFT の実装方法ではなく，FFT の実装に CUDA アーキテクチャがどう使用されたかということについて説明する．本研究では膨大な数の FFT データを処理するために，各スレッドが FFT を逐次的に計算している．FFT の入力データは全てまとめて 1 つの配列に格納されており，各スレッドはその配列の中の自分の担当する領域にアクセスする．このスレッドの振る舞いを

示しているのが図 9 である．

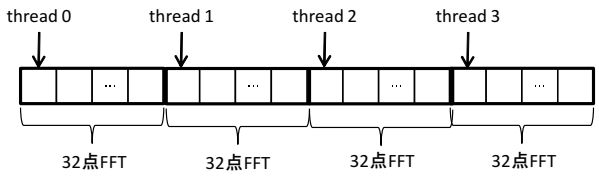


図 9: FFT 入力データに対するスレッド割り当て

このアクセス方法では各スレッドが連続した領域にアクセスしていないのでコアレスドアクセスにはならない．コアレスドアクセスを行うようにするためには，配列をブロックごとに転置してから GPU のメモリへとコピーすればよい (図 10)．FFT では 32 点へのアクセス順は一定になる特徴があるため，ブロック内の全スレッドのアクセスが連続した領域に対して行われコアレスドアクセスとなる．

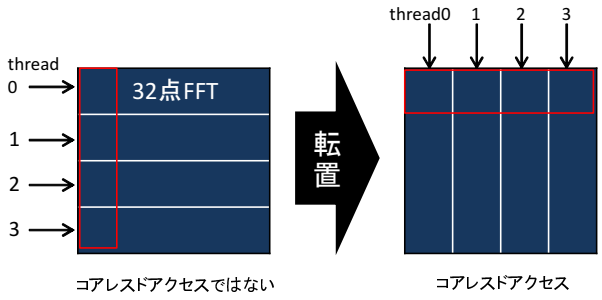


図 10: GPU に転送する前の転置処理

4. 実験結果・考察

実験に使用した GPU は NVIDIA Geforce GTX TITAN でストリーミングマルチプロセッサが 14 基，プロセッサコアが 2688 個，グローバルメモリは 6GB 搭載されている．使用した CPU は Intel Core i7 である．GPU を用いて 32 点 FFT を実行するプログラムを実装し，レイテンシ，総実行時間，及びスループットを計測した．$200*10^6$ 個の FFT データ (50GB) を入力とするプログラムを 11 通りの分割サイズで計測して結果を比較した．分割サイズは 1MB, 2MB, 4MB, 8MB, 16MB, 32MB, 64MB, 128MB, 256MB, 512MB, 1024MB の 11 通りである．

表 2 は分割サイズごとにスループット，レイテンシ，総実行時間を比較したものである．ただし，表 2 のスループットは分割されたデータ内で計測されたレイテンシの最大値である．より小さく分割する 1MB, 2MB, 4MB ではレイテンシが短くなる代わりに総実行時間がかなり長くなっている．一方で 1024MB ではレイテンシが最長となっている．計測結果についてのグラフを図 11 に示す．図 11 は総実行時間とレイテンシを表すグラフである．横軸には分割サイズとスループットが表示されている．グラフから，総実行時間とレイテンシのバランスが優れているのは 32MB 付近であることが見て取れる．このときスループットについても高い値を示している．

サイズ (MB)	スループット (GB/sec)	レイテンシ (ms)	総実行時間 (ms)
1	0.49	0.56	102396.28
2	0.98	0.73	51197.86
4	1.95	1.22	25598.51
8	2.60	2.58	19198.26
16	3.13	4.39	15998.85
32	3.91	7.85	12799.53
64	3.91	15.37	12798.81
128	4.17	29.95	11999.54
256	4.17	59.80	11998.84
512	4.17	119.97	12000.03
1024	4.17	239.73	11999.85

表 2: 分割サイズごとの計測値

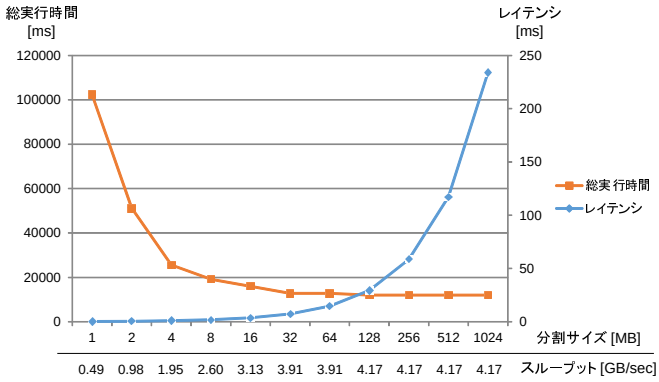


図 11: 計測結果のグラフ

5. まとめ

本研究では GPU を用いて膨大な数の入力データを分割し効率的に処理する高スループット計算システムを実装した．各スレッドが 32 点の FFT を実行するプログラムを分割サイズを変えて実行時間を計測し，計 11 通りのサイズで比較を行った．その結果，32MB のデータごとに処理を行うとき，高スループットを維持しながら実行時間とレイテンシのバランスが優れていることがわかった．

参考文献

[1] NVIDIA Corp., ”CUDA ZONE”, https://developer.nvidia.com/cuda-zone

[2] Takuya Ooura, ”FFT の概略と設計法”, http://www.kurims.kyoto-u.ac.jp/ ooura/fftman/

[3] 佐川雅彦・貴家仁志, ”高速フーリエ変換とその応用”, 1992

Parallel Algorithms for the Summed Area Table on the Asynchronous Hierarchical Memory Machine, with GPU implementations

Akihiko Kasagi, Koji Nakano, and Yasuaki Ito
Department of Information Engineering
Hiroshima University
Kagamiyama 1-4-1, Higashi Hiroshima, 739-8527 Japan

***Abstract*—The Hierarchical Memory Machine (HMM) is a theoretical parallel computing model that captures the essence of computing on CUDA-enabled GPUs. The summed area table (SAT) of a matrix is a data structure frequently used in the area of computer vision which can be obtained by computing the column-wise prefix-sums and then the row-wise prefix-sums. The main contribution of this paper is to introduce the asynchronous Hierarchical Memory Machine (asynchronous HMM), which supports asynchronous execution of CUDA blocks, and show a global-memory-access-optimal parallel algorithm for computing the SAT on the asynchronous HMM. A straightforward algorithm (2R2W SAT algorithm) on the asynchronous HMM, which computes the prefix-sums in every column using one thread each and then computes the prefix-sums in every row, performs 2 read operations and 2 write operations per element of a matrix. The previously published best algorithm (2R1W SAT algorithm) performs 2 read operations and 1 write operation per element. We present a more efficient algorithm (1R1W SAT algorithm) which performs 1 read operation and 1 write operation per element. Clearly, since every element in a matrix must be read at least once, and all resulting values must be written, our 1R1W SAT algorithm is optimal in terms of the global memory access. We also show a combined algorithm ($(1+r)$R1W SAT algorithm) of 2R1W and 1R1W SAT algorithms that may have better performance. We have implemented several algorithms including 2R2W, 2R1W, 1R1W, $(1+r)$R1W SAT algorithms on GeForce GTX 780 Ti. The experimental results show that our $(1+r)$R1W SAT algorithm runs faster than any other SAT algorithms for large input matrices. Also, it runs more than 100 times faster than the best SAT algorithm using a single CPU.**

***Keywords*-memory machine models, prefix-sums, summed area table, image processing, GPU, CUDA**

I. INTRODUCTION

The GPU (Graphics Processing Unit), is a specialized circuit designed to accelerate computation for building and manipulating images [1], [2]. Latest GPUs are designed for general purpose computing and can perform computation in applications traditionally handled by the CPU. Hence, GPUs have recently attracted the attention of many application developers [1]. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [3], the computing engine for NVIDIA GPUs. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in NVIDIA GPUs.

CUDA-enabled GPUs has streaming multiprocessors (SMs) each of which executes multiple threads in parallel. CUDA can use two types of memories in the NVIDIA GPUs: *the shared memory* and *the global memory* [3]. Each SM has the shared memory, an extremely fast on-chip memory with lower capacity, say, 16-48 KBytes, and low latency. Every SM shares the global memory implemented as an off-chip DRAM with large capacity, say, 1.5-6 GBytes, but its access latency is very long. The efficient usage of the shared memory and the global memory is a key for CUDA developers to accelerate applications using GPUs. In particular, we need to consider *bank conflicts* of shared memory access and *coalescing* of global memory access [4], [5]. The address space of shared memory is mapped into several physical memory banks. If two or more threads access the same memory banks at the same time, the access requests are processed in turn. Hence, to maximize the memory access performance, CUDA threads should access distinct memory banks to avoid the bank conflicts of the memory accesses. To maximize the bandwidth between the GPU and the DRAM chips, the consecutive addresses of the global memory must be accessed at the same time. Thus, CUDA threads should perform coalesced access when they access the global memory.

In our previous paper [6], we have introduced two models, *the Discrete Memory Machine (DMM)* and *the Unified Memory Machine (UMM)*, which reflect the essential features of the shared memory and the global memory of NVIDIA GPUs. The outline of the architectures of the DMM and the UMM is illustrated in Figure 1. In both architectures, *a sea of threads (Ts)* are connected to *the memory banks (MBs)* through *the memory management unit (MMU)*. Each thread is a Random Access Machine (RAM) [7], which can execute fundamental operations in a time unit. Threads are executed in SIMD [8] fashion, and they run on the same program and work on the different data. MBs constitute a single address space of the memory. A single address space of the memory is mapped to the MBs in an interleaved way such that the word of data of address i is stored in the $(i \bmod w)$-th bank, where w is the number of MBs. The

main difference of the two architectures is the connection of the address line between the MMU and the MBs, which can transfer an address value. In the DMM, the address lines connect the MBs and the MMU separately, while a single address line from the MMU is connected to the MBs in the UMM. Hence, in the UMM, the same address value is broadcast to every MB, and the same address of the MBs can be accessed in each time unit. On the other hand, different addresses of the MBs can be accessed in the DMM.

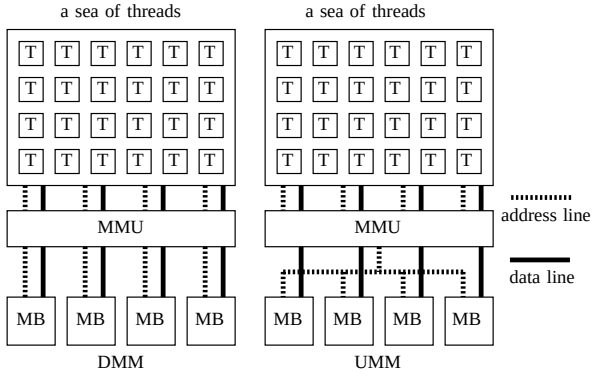


Figure 1. The architectures of the DMM and the UMM with $w = 4$

Quite recently, we have introduced the Hierarchical Memory Machine (HMM) [9], which is a hybrid of the DMM and the UMM. The HMM is a more practical parallel computing model that reflects the hierarchical architecture of CUDA-enabled GPUs. Figure 2 illustrates the architecture of the HMM. The HMM consists of d DMMs and a single UMM. Each DMM has w memory banks and the UMM has w memory banks. We call the memory banks of each DMM *the shared memory* and those of the UMM *the global memory* after CUDA-enabled GPUs. Each DMM can work independently and can perform the computation using its shared memory. Also, all threads of DMMs work as a single UMM and can access to the global memory. While the memory access latency of the shared memory of GPUs is very low, that of the global memory is several hundred clock cycles [3]. Hence, we assume that the latency of the shared memory is 1, and we use parameter l to denote the latency of the global memory in the HMM.

Suppose that a matrix a of size $\sqrt{n} \times \sqrt{n}$ is given. The summed area table (SAT) [10] is a matrix b of the same size such that

$$b[i][j] = \sum_{0 \leq i' \leq i, 0 \leq j' \leq j} a[i'][j'].$$

It should have no difficulty to confirm that the SAT can be obtained by computing the column-wise prefix-sums and the row-wise prefix-sums as illustrated in Figure 3. Once we have the summed area table, the sum of any rectangular area

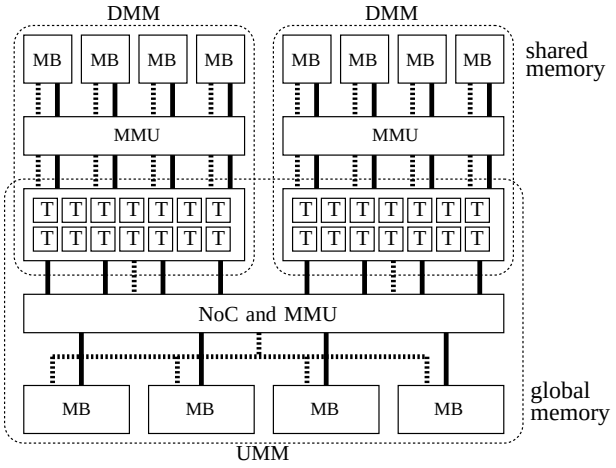


Figure 2. The architecture of the HMM with $d = 2$ DMMs and width $w = 4$

of a can be computed by evaluating

$$\sum_{u < i \leq d, l < j \leq r} a[i][j] = b[d][r] + b[u][r] + b[d][l] - b[u][l].$$

Thus, the sum of a rectangular area can be computed using four elements of the summed area table b. Since the sum of any rectangular area can be computed in $O(1)$ time the summed area table has a lot of applications in the are of image processing and computer vision [11]. In our previous paper [12], we have presented a parallel algorithm that computes the SAT in $O(\frac{n}{w} + \frac{nl}{p} + l \log n)$ time units using p threads on the UMM with width w and latency l. This algorithm is optimal in the sense that any SAT algorithm takes at least $\Omega(\frac{n}{w} + \frac{nl}{p} + l \log n)$ time units. However, this algorithm repeats pairwise addition and has a large constant factor in the computing time and it is not practically efficient.

The main contribution of this paper is to introduce the asynchronous Hierarchical Memory Machine (asynchronous HMM), which supports asynchronous execution of CUDA blocks, and show a global-memory-access-optimal parallel algorithm for computing the summed area table stored in the global memory of the asynchronous HMM. A straightforward algorithm (2R2W SAT algorithm) on the asynchronous HMM, which computes the column-wise prefix-sums and then the row-wise prefix-sums, performs 2 read operations and 2 write operations per element of a matrix. The best known algorithm (2R1W SAT algorithm) so far performs 2 read operations and 1 write operation per element [13]. We present a more efficient algorithm (1R1W SAT algorithm) on the asynchronous HMM, which performs only 1 read operation and 1 write operation per element. Clearly, since every element in a matrix must be read at least once and all resulting values must be written, our 1R1W SAT algorithm is optimal in terms of the global memory access. We also show a combined algorithm ($(1 + r)$R1W SAT algorithm)

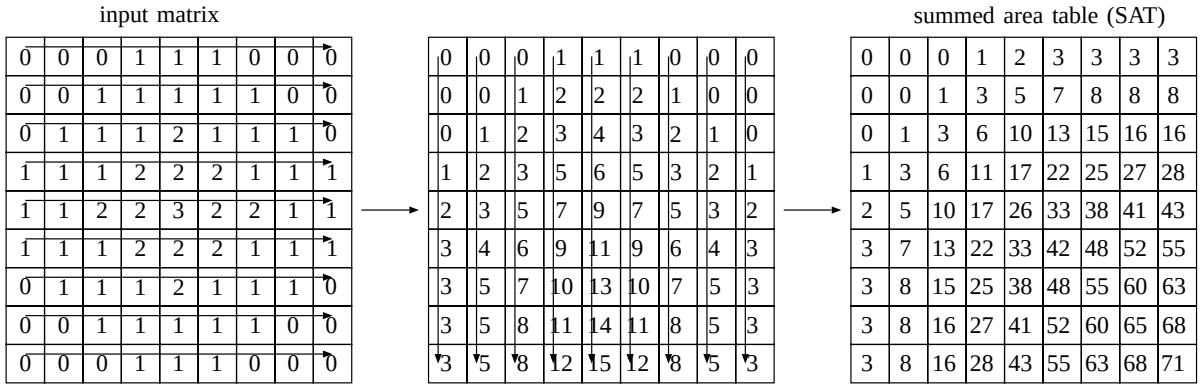


Figure 3. The summed area table (SAT) of a 9×9 matrix and 2R2W SAT algorithm

of 2R1W and 1R1W SAT algorithms that can run faster than any other algorithms for large matrices. Table I shows the total number of memory access operations to the global memory and the shared memory, the number of barrier synchronization steps, and the global memory access cost. The global memory access cost, which is computed from the number of global memory access operations and the number of barrier synchronization steps, approximates the computing time on the HMM. For simplicity, in the table, we omit small terms to focus on dominant terms. For example, 2R2W SAT algorithm performs $n - \sqrt{n}$ coalesced write operations, but we simply write n in the corresponding entry.

We have also implemented several algorithms including 2R2W, 2R1W, 1R1W, $(1 + r)$R1W SAT algorithms on GeForce GTX 780 Ti. The experimental results show that our $(1 + r)$R1W SAT algorithm runs faster than any other SAT algorithms for large input matrices. Further, it runs more than 100 times faster than the best SAT algorithm using a single CPU.

II. The DMM, the UMM, the HMM and the asynchronous HMM

We first define *the Discrete Memory Machine (DMM)* of width w and latency l. Let $B[j] = \{j, j + w, j + 2w, j + 3w, \ldots\}$ $(0 \leq j \leq w - 1)$ be a set of address of *the j-th memory bank* of the memory. In other words, address i is in the $(i \bmod w)$-th memory bank. We assume that addresses in different banks can be accessed in a time unit, but no two addresses in the same bank can be accessed in a time unit. Also, we assume that l time units are necessary to complete an access request and continuous requests are processed in a pipeline fashion. Thus, it takes $k+l-1$ time units to complete memory access requests to k addresses in a particular bank.

We assume that p threads are partitioned into $\frac{p}{w}$ groups of w threads called *warps*. More specifically, p threads $T(0)$, $T(1)$, $\ldots$, $T(p-1)$ are partitioned into $\frac{p}{w}$ warps $W(0), W(1), \ldots, W(\frac{p}{w}-1)$ such that $W(i) = \{T(i \cdot w), T(i \cdot w + 1), \ldots, T((i+1) \cdot w - 1)\}$ $(0 \leq i \leq \frac{p}{w} - 1)$. Warps are dispatched for memory access in turn, and w threads in a warp try to access the memory at the same time. In other words, $W(0), W(1), \ldots, W(\frac{p}{w} - 1)$ are dispatched in a round-robin manner if at least one thread in a warp requests memory access. If no thread in a warp needs memory access, such warp is not dispatched for memory access. When $W(i)$ is dispatched, w threads in $W(i)$ send memory access requests, at most one request per thread, to the memory. We also assume that a thread cannot send a new memory access request until the previous memory access request is completed. Hence, if a thread sends a memory access request, it must wait at least l time units to send a new memory access request.

We next define *the Unified Memory Machine (UMM)* of width w and latency l as follows. Let $A[j] = \{j \cdot w, j \cdot w + 1, \ldots, (j+1) \cdot w - 1\}$ denote a set of addresses in *the j-th address group*. We assume that addresses in the same address group are processed at the same time. However, if they are in the different groups, one time unit is necessary for each of the groups. Also, similarly to the DMM, p threads are partitioned into warps and each warp accesses the memory in turn.

Figure 4 shows examples of memory access on the DMM and the UMM. We assume that each memory access request is completed when it reaches the last pipeline stage. Two warps $W(0)$ and $W(1)$ access to $\langle 7, 5, 15, 0\rangle$ and $\langle 10, 11, 12, 9\rangle$, respectively. In the DMM, memory access requests by $W(0)$ are separated into two pipeline stages, because addresses 7 and 15 are in the same bank $B(3)$. Those by $W(1)$ occupies 1 stage, because all requests are in distinct banks. Thus, the memory requests occupy three stages, it takes $3 + 5 - 1 = 7$ time units to complete the memory access. In the UMM, memory access requests by $W(0)$ are destined for three address groups. Hence the memory requests occupy three stages. Similarly those by $W(1)$ occupy two stages. Hence, it takes $5+5-1 = 9$ time units to complete the memory access.

Table I
THE PERFORMANCE OF SAT ALGORITHMS ON THE HMM

SAT algorithms	global memory access		shared memory access	barrier synchronization steps	global memory access cost
	Coalesced Read/Write	Stride Read/Write	Read/Write		
2R2W	n / n	n / n	-	1	$2n + 2\frac{n}{w} + 2l$
4R4W	$4n$ / $4n$	-	n / n	3	$8\frac{n}{w} + 4l$
4R1W	-	$4n$ / n	-	$2\sqrt{n}$	$5n + 2\sqrt{n}l$
2R1W	$2n$ / n	-	$4n$ / $4n$	$2d + 2$	$3\frac{n}{w} + (2d + 3)l$
1R1W	n / n	-	$2n$ / $2n$	$2\frac{\sqrt{n}}{w}$	$2\frac{n}{w} + 2\frac{\sqrt{n}}{w}l$
1.25R1W	$1.25n$ / n	-	$2.5n$ / $2.5n$	$\frac{\sqrt{n}}{w} + 4d + 4$	$1.25\frac{n}{2} + (\frac{\sqrt{n}}{w} + 4d + 5)l$
$(1 + r)$R1W	$(1 + r)n$ / n	-	$(2 + \sqrt{r})n$ / $(2 + \sqrt{r})n$	$2\frac{(1-\sqrt{r})\sqrt{n}}{w} + 4d + 4$	$(1 + r)\frac{n}{w} + (2\frac{(1-\sqrt{r})\sqrt{n}}{w} + 4d + 5)l$

d is the depth of recursion, which takes value no more than 1 from the practical point of view.

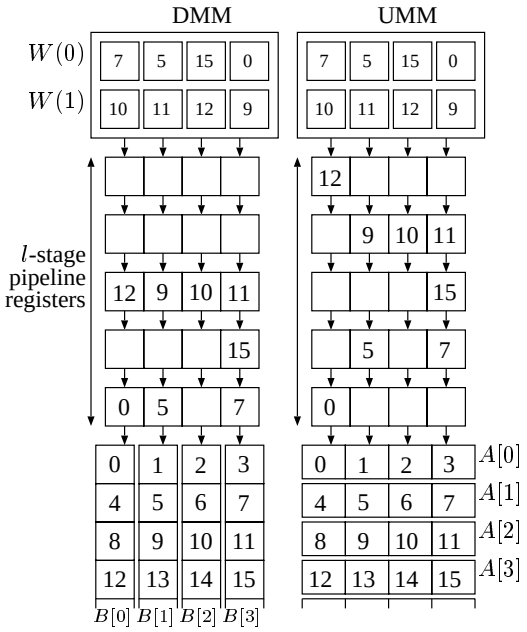


Figure 4. Examples of memory access on the DMM and the UMM

Next, we define *the Hierarchical Memory Machine (HMM)* [9]. The HMM consists of d DMMs and a single UMM as illustrated in Figure 2. Each DMM has w memory banks and the UMM also has w memory banks. We call the memory banks of each DMM *the shared memory* and those of the UMM *the global memory*. Each DMM works independently. Threads are partitioned into warps of w threads, and each warp are dispatched for the memory access for the shared memory in turn. Further, each warp of w threads in all DMMs can send memory access requests to the global memory. Figure 2 illustrates the architecture of the HMM with $d = 2$ DMMs. Each DMM and the UMM has $w = 4$ memory banks. The shared memory of each DMM and the global memory of the UMM correspond to "the shared memory" of each streaming multiprocessor and "the global memory" of GPUs. We also assume that the shared memory in each DMM of the HMM can store up to $O(w^2)$ numbers. The capacity of the shared memory of latest CUDA-enabled GPUs is up to 48KBytes and the number w of the banks is 32 [3]. Since an array of 32^2 double (64-bit) numbers occupy 8KBytes, each shared memory can store at most 6 such matrices. Thus, it is reasonable to assume that DMM can store $O(w^2)$ numbers in the shared memory.

For more realistic model for GPUs, we introduce *the asynchronous Hierarchical Memory Machine* (asynchronous HMM). In the asynchronous HMM, DMMs work asynchronously in the sense that some DMMs may work slightly slower or faster than the others. Instead, all threads in all DMMs can execute a barrier synchronization instruction. If a thread in a DMM executes the barrier synchronization instruction, it must wait until all the other threads in all DMMs execute it. Also, after all threads execute the barrier synchronization instruction, all DMMs are *reset*, that is, the shared memory of all DMMs are initialized and all data stored in it are lost. The reader may think that this reset assumption of all DMMs is not reasonable. However, this assumption is mandatory for program scalability of DMMs in the HMM. More specifically, suppose that a programmer write a program of the HMM with d DMMs. It may be possible to execute this program in the HMM with d' DMMs such that $d' < d$. If this is the case, the program of d' DMMs are executed until a barrier synchronization instruction is executed. After that, the d' DMMs are reset and the program of next d' DMMs are executed. The same procedure is repeated until all threads execute the barrier synchronization instruction. Hence, it makes sense to assume that all DMMs are reset after each barrier synchronization step. The previous DMMs is responsible for copying the data stored in the shared memory to the global memory before barrier synchronization if they are used after the synchronization. Actually, we need to terminate a CUDA kernel call for a GPU when barrier synchronization of all threads is necessary [3]. When a CUDA kernel call is terminated for barrier synchronization, the data stored in the shared memory by a CUDA block are lost. This is because

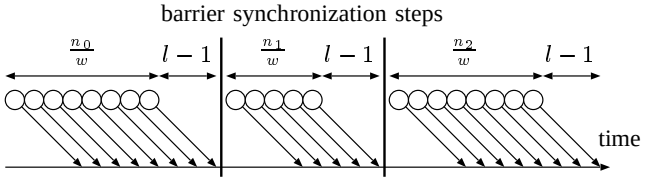


Figure 5. Timing chart of coalesced memory access to the global memory with two barrier synchronization steps

CUDA blocks are executed in streaming multiprocessors with small shared memory one by one in turn.

III. The global memory access cost on the HMM and the diagonal arrangement on the DMM

Let C, S, and B be the total number of coalesced global memory access operations, the total number of stride global memory access operations, and the number of barrier synchronization steps performed on the HMM. *The global memory access cost* is defined to be $\frac{C}{w}+S+(B+1)(l-1)$. We will show that the global memory access cost approximates the computing time on the HMM if the computation performed in each DMM is negligible.

Suppose that an algorithm performs n coalesced memory access operations and two barrier synchronization steps. Clearly, by two barrier synchronization steps, the memory access is partitioned into three stages as illustrated in Figure 5. Let n_0, n_1, and n_2 such that $n = n_0 + n_1 + n_2$ be the numbers of memory access operations performed in the three stages. Since w coalesced memory access operations by a warp of w threads occupy one pipeline stage, the three stages takes $\frac{n_0}{w}+l-1$, $\frac{n_1}{w}+l-1$, and $\frac{n_2}{w}+l-1$ time units respectively. Hence, the algorithm runs $\frac{n}{w}+3(l-1)$ time units. Also, if n memory access operations are stride, each memory access operation occupy one pipeline stage, the algorithm runs in $n+3(l-1)$ time units. In general, if an algorithm performs C coalesced memory access operations, S stride memory access operations, and B barrier synchronization steps, it runs $\frac{C}{w}+S+(B+1)(l-1)$, which is the global memory access cost.

Suppose that we have a matrix of size $w \times w$ in the shared memory of a DMM in the HMM. Since a column of the matrix is in the same bank, column-wise access by w threads has bank conflicts, while row-wise access is conflict-free. In our previous paper [6], we have presented *a diagonal arrangement* of a matrix such that each (i, j) element is arranged in $a[i][(i + j) \bmod w]$. Figure 6 illustrates the diagonal arrangement of a 4×4 matrix. We can confirm that both a row-wise access to $(1, 0), (1, 1), (1, 2), (1, 3)$ and a column-wise access to $(0, 1), (1, 1), (2, 1), (3, 1)$ are conflict-free. Thus, we have,

Lemma 1: In the diagonal arrangement of a $w \times w$ matrix, both a row-wise access and a column-wise access are conflict-free.

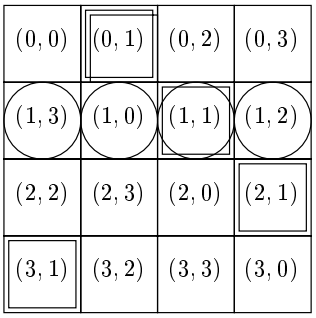


Figure 6. Diagonal arrangement of a 4×4 matrix

The diagonal arrangement is used to compute the SAT of a $w \times w$ matrix in a shared memory and transpose of a matrix in the global memory of the HMM.

IV. 2R2W and 4R4W SAT algorithms

Let s_i denote a local register of thread $T(i)$ ($0 \le i \le n-1$). As illustrated in Figure 3, the summed area table (SAT) of a $\sqrt{n} \times \sqrt{n}$ matrix a can be computed by the column-wise prefix-sums and the row-wise prefix-sums:

[2R2W SAT algorithm]
for $i \leftarrow 0$ to $\sqrt{n}$ do in parallel // column-wise prefix-sums
  $T(i)$ performs $s_i \leftarrow a[0][i]$
  for $j \leftarrow 1$ to $\sqrt{n}-1$ do
    $T(i)$ performs $s_i \leftarrow s_i + a[j][i]$
    $T(i)$ performs $a[j][i] \leftarrow s_i$
barrier_synchronization
for $i \leftarrow 0$ to $\sqrt{n}$ do in parallel // row-wise prefix-sums
  $T(i)$ performs $s_i \leftarrow a[i][0]$
  for $j \leftarrow 1$ to $\sqrt{n}-1$ do
    $T(i)$ performs $s_i \leftarrow s_i + a[i][j]$
    $T(i)$ performs $a[i][j] \leftarrow s_i$

In the computation of the column-wise prefix-sums, $a[0][0], a[0][1], \ldots, a[0][\sqrt{n}-1]$ are read. After that, for each j ($1 \le j \le \sqrt{n}-1$), $a[j][0], a[j][1], \ldots, a[j][\sqrt{n}-1]$ are read and written. Clearly, memory access to these elements are coalesced. In the computation of the column-wise prefix-sums, $a[0][0], a[1][0], \ldots, a[\sqrt{n}-1][0]$ are read. After that, for each j ($1 \le j \le \sqrt{n}-1$), $a[0][j], a[1][j], \ldots, a[\sqrt{n}-1][j]$ are read and written. Memory access to these elements are coalesced. Hence, 2R2W SAT algorithm performs $2n-\sqrt{n}$ coalesced memory access operations and $2n-\sqrt{n}$ stride memory access operations. Since 2R2W SAT algorithm has one barrier synchronization step, we have,

Lemma 2: The global memory access cost of 2R2W SAT algorithm is at most $2n+2\frac{n}{w}+2(l-1)$.

We can avoid stride memory access when we compute the row-wise prefix-sums by transposing a matrix. More specifically, the row-wise prefix-sums can be obtained by transpose, column-wise prefix-sums, and transpose. It has

been shown in [14] that transpose of a matrix of size $\sqrt{n} \times \sqrt{n}$ in the global memory of the HMM can be done in $2n$ coalesced memory access operations with no barrier synchronization step. The idea of the transpose it to partition the matrix into $\frac{\sqrt{n}}{w} \times \frac{\sqrt{n}}{w}$ blocks with $w \times w$ elements each. We can transpose a block via a $w \times w$ matrix with diagonal arrangement in a shared memory of a DMM efficiently. First, a block in the global memory is read in row-wise and it is written in a $w \times w$ matrix with diagonal arrangement in row-wise. After that, the $w \times w$ matrix is read in column-wise and it is written in a a block in the global memory in row-wise. The reader should refer to Figure 7 illustrating transpose of a block using a 4×4 matrix with diagonal arrangement. By executing this block transpose for all blocks in parallel so that a pair of corresponding two blocks are swapped appropriately, the transpose of a $\sqrt{n} \times \sqrt{n}$ can be done. The reader should refer to [14] for the details of the transpose.

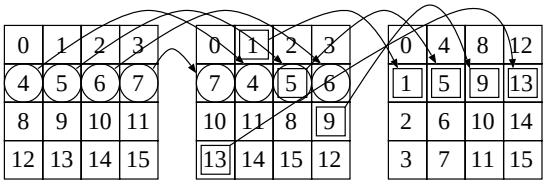


Figure 7. Transpose of a block using a 4×4 matrix with diagonal arrangement

By executing matrix transpose twice, we can design 4R4W SAT algorithm as follows:

[4R4W SAT algorithm]

Step 1: Compute the column-wise prefix-sums

Step 2: Transpose

Step 3: Compute the column-wise prefix-sums

Step 4: Transpose

After Steps 1, 2, and 3, barrier synchronization is necessary. Also, each step needs no more than $2n$ coalesced global memory access. Thus, we have,

Lemma 3: The global memory access cost of 4R4W SAT algorithm is at most $8\frac{n}{w} + 4(l-1)$.

V. 2R1W SAT ALGORITHM

The main purpose of this section is to review a SAT algorithm for GPU shown in [13]. Since this SAT algorithm performs $2n$ read and n write operations to the global memory, we call it 2R1W SAT algorithm. 2R1W SAT algorithm that we will explain is slightly different from that in [13] for easy understanding of the algorithm.

Suppose that a $\sqrt{n} \times \sqrt{n}$ matrix a is partitioned into $\frac{\sqrt{n}}{w} \times \frac{\sqrt{n}}{w}$ blocks of $w \times w$ elements each. 2R1W SAT algorithm has three steps as follows:

[2R1W SAT algorithm]

Step 1: Each DMM reads a block in the global memory and write it in the shared memory. The column-wise sums, the row-wise sums, and the sum of the block are computed. More specifically, for a block a' of size $w \times w$,

- column-wise sums: $C[i] = \sum_{j=0}^{w-1} a'[j][i]$ for all i ($0 \leq i \leq w-1$),
- row-wise sums: $R[i] = \sum_{j=0}^{w-1} a'[i][j]$ for all i ($0 \leq i \leq w-1$), and
- sum: $S = \sum_{i=0}^{w-1} \sum_{j=0}^{w-1} a'[i][j]$.

The column-wise sums of all blocks excluding the bottom blocks are written in the global memory such that they constitute a matrix of size $(\frac{\sqrt{n}}{w} - 1) \times \sqrt{n}$ in the global memory. Similarly, the row-wise sums of all blocks constitute a matrix of size $\sqrt{n} \times (\frac{\sqrt{n}}{w} - 1)$, and the sums constitute a matrix of size $(\frac{\sqrt{n}}{w} - 1) \times (\frac{\sqrt{n}}{w} - 1)$. The reader should refer to Figure 8 for illustrating the resulting values for a matrix in Figure 3 with $w = 3$. Let $\mathcal{C}$, $\mathcal{R}$, and $\mathcal{S}$ denote matrices of the resulting values for the column-wise sums, the row-wise sums, and the sums, respectively. In Figure 8, the sizes of $\mathcal{C}$, $\mathcal{R}$, and $\mathcal{S}$ are 2×9, 9×2, and 2×2, respectively.

Step 2: The column-wise prefix-sums of $\mathcal{C}$ and the row-wise prefix-sums of $\mathcal{R}$ are computed in the same way as 2R2W SAT algorithm. If $\mathcal{S}$ is no larger than $w \times w$ then we compute the SAT of $\mathcal{S}$ using a single DMM. Otherwise, we execute 2R1W SAT algorithm recursively for $\mathcal{S}$. The reader should refer to Figure 8 for the resulting values of $\mathcal{C}$, $\mathcal{R}$, and $\mathcal{S}$.

Step 3-1: Each DMM reads a block from the global memory and write it in the shared memory. Let a' denote a block read by a particular DMM. It reads w elements in $\mathcal{C}$, and adds them to the top row of a' so that each of the resulting sums is the sum of all elements above it, inclusive, in the same column. Similarly, it reads w elements in $\mathcal{R}$, and adds them to the leftmost column of a' so that each of the resulting sums is the sum of all elements to the left-side of it, inclusive, in the same row. Further, an element in $\mathcal{S}$ is added to the top left corner of a' so that the resulting sum is the the sum of all blocks above and to the left of it, inclusive. The reader should refer to Figure 9 illustrating these operations for a block. Also, Figure 8 illustrates the resulting values of all blocks.

Step 3-2: Each DMM computes the SAT of a block obtained in Step 3-1 and the resulting values are written in the global memory. Figure 9 illustrates the values of a block before and after this step. The reader should have no difficulty to confirm that the block thus obtained stores the SAT of the input matrix correctly.

Let us evaluate the global memory access cost. In Step 1, all elements in a are read, and $\mathcal{C}$, $\mathcal{R}$, and $\mathcal{S}$ are written. Thus, n elements are read from the global memory and less than $2\frac{n}{w} + \frac{n}{w^2}$ elements are written in the global memory. Note that we should write $\mathcal{R}^T$, that is, transposed $\mathcal{R}$ in the global memory for the purpose of coalesced memory access for the row-wise prefix-sums computation in Step 2. If this is the case, the row-wise prefix-sums of $\mathcal{R}$ corresponds to the

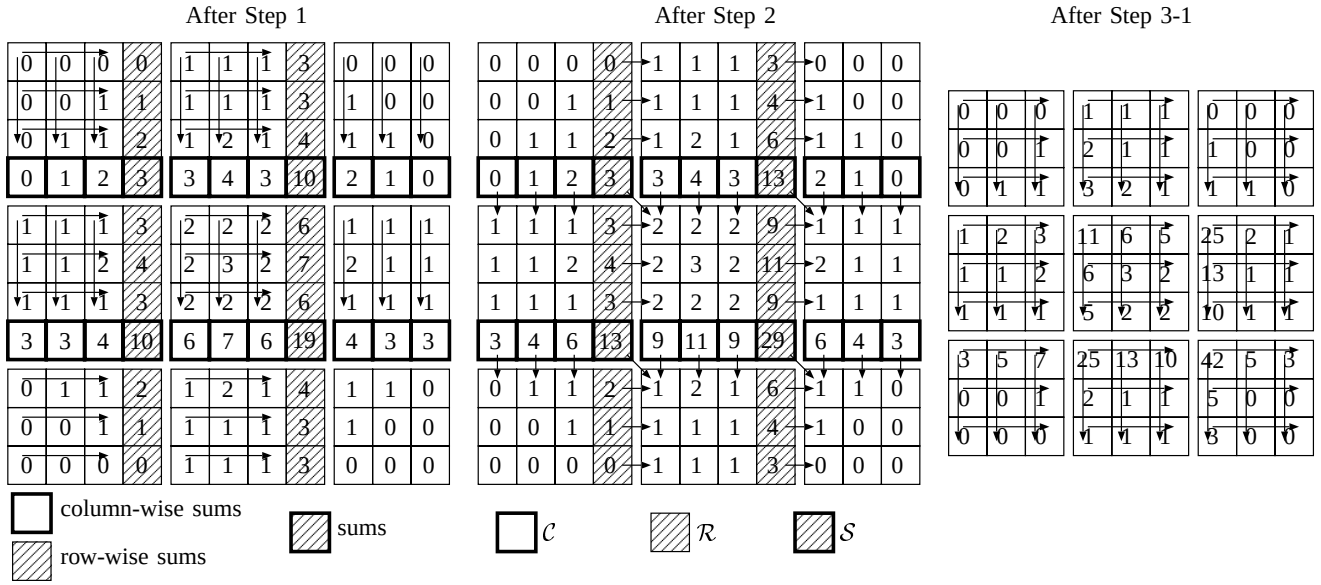


Figure 8. 2R1W SAT algorithm executed for a matrix in Figure 3 with $w = 3$

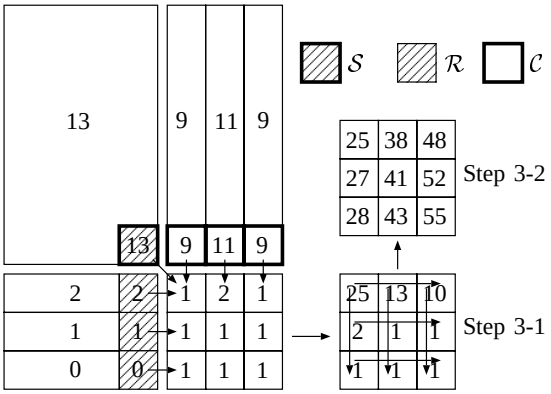


Figure 9. Step 3 of 2R1W SAT algorithm

column-wise prefix-sums of $\mathcal{R}^T$, which can be computed by coalesced memory access to the global memory. Also, in Step 1, the column-wise sums, the row-wise sums, and the SAT of a block in a shared memory are computed. This computation can be done without bank conflicts by diagonal arrangement of a block. Each of the d DMMs performs the computation of the column-wise sums for $\frac{n}{w^2 d}$ blocks of size $w \times w$. Since the memory access is conflict-free, this takes only $\frac{n}{wd}$ time units, which is so small that it can be hidden by latency overhead.

Step 2 performs the computation of the the column-wise prefix-sums and the row-wise prefix-sums for matrices of $\frac{n}{w}$ elements. It also computes the SAT of $\frac{n}{w^2}$ elements, recursively, which performs global memory access to $O(\frac{n}{w^2})$ elements. In Step 3-1 all elements in a and the resulting values of $\mathcal{C}$, $\mathcal{R}$, and $\mathcal{S}$ are read from the global memory. In Step 3-2 the resulting SAT is written in the global memory. Thus, 2R1W SAT algorithm performs at most $3n+8\frac{n}{w}+O(\frac{n}{w^2})$ coalesced memory access operations. This includes the memory access by recursive computation of $\mathcal{S}$.

Let us evaluate the number of barrier synchronization steps. Barrier synchronization step is necessary after Steps 1 and 2 if the SAT of $\mathcal{S}$ is computed without recursion. If the SAT of $\mathcal{S}$ is computed recursively, additional two barrier synchronization steps is necessary for each recursion. Hence, if 2R1W SAT algorithm involves d recursions, it performs $2d + 2$ barrier synchronization steps. Thus, we have,

Lemma 4: The global memory access cost of 2R1W SAT algorithm with recursion depth d is $3\frac{n}{w} + 8\frac{n}{w^2} + O(\frac{n}{w^3}) + (2d + 3)(l - 1)$.

Since $w = 32$ in current GPUs, $d = 0$ if $n \leq 2^{20}$ and $d = 1$ if $n \leq 2^{30}$. Thus, d is no more than 1 from the practical point of view.

VI. Our 1R1W SAT algorithm

The main purpose of this section is to show our novel SAT algorithm called 1R1W SAT algorithm. This algorithm performs only $n + O(\frac{n}{w})$ read operations and $n + O(\frac{n}{w})$ write operations to the global memory. Before showing 1R1W SAT algorithm, we present 4R1W SAT algorithm. By combining techniques used in 4R1W SAT and 2R1W SAT algorithms, we can obtain 1R1W SAT algorithm.

Let b be the SAT of an input $\sqrt{n} \times \sqrt{n}$ matrix a. Suppose that the values of $b[i-1][j-1]$, $b[i][j-1]$, and $b[i-1][j]$ are already computed. We can obtain the value of $b[i][j]$ by evaluating the following formula:

$$b[i][j] = a[i][j] + b[i][j-1] + b[i-1][j] - b[i-1][j-1] \quad (1)$$

From this formula, 4R1W SAT algorithm computes the SAT in a diagonal scan order from the top left to the bottom right. More specifically, 4R1W algorithm has $2\sqrt{n}-1$ stages and each Stage k ($0 \leq k \leq 2\sqrt{n}-2$) computes Formula (1) for all i and j such that $i+j=k$. Figure 10 illustrates the computation performed in Stage 7.

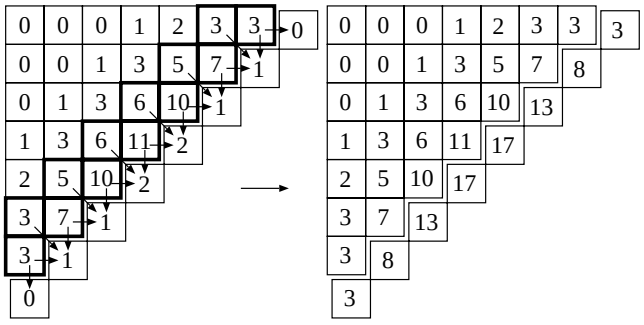


Figure 10. Stage 7 of 4R1W SAT algorithm

Let us evaluate the performance of 4R1W SAT algorithm. To compute each $b[i][j]$, 3 elements in b and 1 element in a are read. Also, the resulting value is written in b. Thus, $4n$ reading operations and n writing operations are performed. Unfortunately, all memory access are stride. Further, barrier synchronization step is necessary after every stage from 0 to $2\sqrt{n}-3$. Thus, we have,

Lemma 5: The global memory access cost of 4R1W SAT algorithm is $5n+(2\sqrt{n}-1)l$.

We are now in a position to show our new 1R1W SAT algorithm. The idea is to extend 4R1W SAT algorithm to perform SAT computation in block-wise. In each block-wise computation, a similar computation to 2R1W SAT algorithm is performed. Again, an input matrix a of size $\sqrt{n} \times \sqrt{n}$ is partitioned into $\frac{\sqrt{n}}{w} \times \frac{\sqrt{n}}{w}$ blocks of size $w \times w$ each. Let $A(i,j)$ ($0 \leq i,j \leq \frac{\sqrt{n}}{w}-1$) denote a block in the i-th row and in the j-th column. 1R1W SAT algorithm has $2\frac{\sqrt{n}}{w}-1$ stages. Each Stage k ($0 \leq k \leq 2\frac{\sqrt{n}}{w}-2$) computes the SAT of block $A(i,j)$ with $i+j=k$. The reader should refer to Figure 11 for illustrating the computation performed in Stage 3. In Stage 3, $A(2,1)$ and $A(1,2)$ are computed using the resulting values in $A(2,0)$, $A(1,1)$, and $A(0,2)$. From these resulting values, we can obtain $\mathcal{C}$, $\mathcal{R}$, and $\mathcal{S}$ for each of $A(2,1)$ and $A(1,2)$. For example, in Figure 11, $\mathcal{C}$ for $A(2,1)$ is $(9,11,9)$. This can be obtained by the resulting value 13 of the bottom top corner in $A(1,0)$ and the resulting values $(22,33,42)$ in the bottom row of $A(1,1)$. More specifically, we can obtain $\mathcal{C}$ by computing pairwise subtraction $(22,33,42)-(13,22,33)=(9,11,9)$. After the values of $\mathcal{C}$, $\mathcal{R}$, and $\mathcal{S}$, these values are added to $A(2,1)$ in the same way as Step 3-1 of 2R1W SAT algorithm. Finally, we compute the SAT of $A(2,1)$ in the same way as Step 3-2 of 2R1W SAT algorithm.

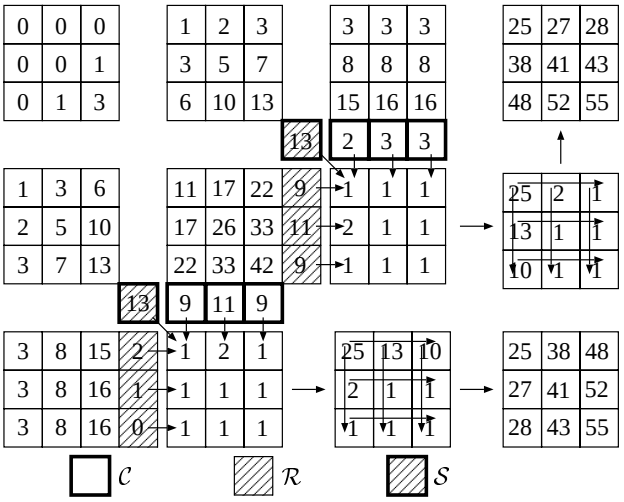


Figure 11. Stage 3 of 1R1W SAT algorithm for a matrix in Figure 3

Let us evaluate the performance of 1R1W SAT algorithm. In each stage, the values of a block in the global memory are read and the resulting values are written to the global memory. Also, the values necessary to compute $\mathcal{C}$, $\mathcal{R}$, and $\mathcal{S}$ are read and written to the global memory. For each block, $2w+1$ elements are read from the global memory for this task. Since we have $\frac{n}{w^2}$ blocks, $n+(2w+1)\cdot\frac{n}{w^2}$ elements are read and $n+(2w+1)\cdot\frac{n}{w^2}$ elements are written in all stages. Barrier synchronization is necessary after each of Stages from 0 to $2\frac{\sqrt{n}}{w}-3$. Thus, we have,

Theorem 6: The global memory access cost of 4R1W SAT algorithm is $2\frac{n}{w}+O(\frac{n}{w^2})+(2\frac{\sqrt{n}}{w}-2)l$.

VII. $(1+r)$R1W SAT ALGORITHM

The main purpose of this section is to accelerate the SAT computation further by combining 1R1W and 2R1W SAT algorithms. The idea of further acceleration is to use 2R1W SAT algorithm in early and late stages of 1R1W SAT algorithms to reduce the latency overhead.

Again, suppose that a $\sqrt{n} \times \sqrt{n}$ matrix a is partitioned into $\frac{\sqrt{n}}{w} \times \frac{\sqrt{n}}{w}$ blocks of size $w \times w$ each. As illustrated in Figure 12, for any fixed parameter r ($0<r<1$), we partition blocks into (A) top left triangle, (B) bottom right triangle, and (C) remaining blocks. Clearly, (A) and (B) have $\frac{\sqrt{r}\sqrt{n}}{w}+(\frac{\sqrt{r}\sqrt{n}}{w}-1)+\cdots+1=\frac{\sqrt{r}\sqrt{n}}{w}\cdot(\frac{\sqrt{r}\sqrt{n}}{w}+1)/2 \approx \frac{rn}{2w^2}$ blocks each. We first use 2R1W SAT algorithm to compute the SAT of (A). After that, we use 1R1W SAT algorithm for (C). Finally, 2R1W SAT algorithm is used for the computation of the SAT in (B).

Let us evaluate the performance. Since (A) and (B) have approximately $\frac{rn}{2}$ elements in $\frac{rn}{2w^2}$ blocks each, 2R1W SAT algorithm for (A) and (B) performs $rn+O(\frac{rn}{w})$ read operations and $\frac{rn}{2}+O(\frac{rn}{w})$ write operations each. Also, since (C) has $(1-r)n$ elements, 1R1W SAT algorithm for

(C) performs $(1-r)n + O(\frac{(1-r)n}{w})$ read operations and $(1-r)n + O(\frac{(1-r)n}{w})$ write operations. Hence, this SAT algorithm performs $(1+r)n + O(\frac{n}{w})$ read operations and $n+O(\frac{n}{w})$ write operations. Thus, we call this SAT algorithm $(1+r)$R1W SAT algorithm. Further, 2R1W SAT algorithm for (A) and (B) needs $2+2d$ barrier synchronization steps each, where d is the depth of the recursion of 2R1W SAT algorithm. Since 1R1W SAT algorithm for (C) has $2\frac{(1-\sqrt{r})\sqrt{n}}{w} - 1$ stages, it needs $2\frac{(1-\sqrt{r})\sqrt{n}}{w} - 2$ barrier synchronization steps. Also, after the computation of the SAT for (A) and (B), 1 barrier synchronization steps each is necessary. Totally, $(1+r)$R1W SAT algorithm executes $2\frac{(1-\sqrt{r})\sqrt{n}}{w} + 4 + 4d$ barrier synchronization steps. Thus, we have,

Theorem 7: The global memory access cost of $(1+r)$R1W SAT algorithm is $(2+r)\frac{n}{w} + O(\frac{n}{w^2}) + (2\frac{(1-\sqrt{r})\sqrt{n}}{w} + 5 + 4d)l$.

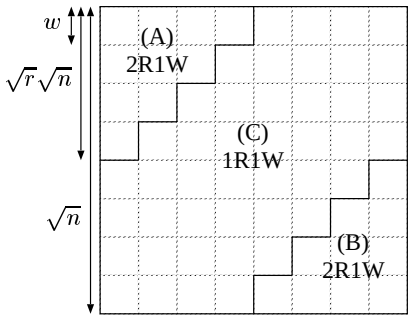


Figure 12. Partition of a matrix for $(1+r)$R1W SAT algorithm

When $r = 0.25$, the global memory access cost of 1.25R1W SAT algorithm is $2.5\frac{n}{w} + (\frac{\sqrt{n}}{w} + 5 + 4d)l$ time units. Since 2R1W and 1R1W SAT algorithms run approximately $3\frac{n}{w} + (3+2d)l$ and $2\frac{n}{w} + 2\frac{\sqrt{n}}{w}l$ time units, respectively, 1.25R1W SAT algorithm may run faster than these algorithms. Further, we can select the best value r that minimize the running time of $(1+r)$R1W SAT algorithm.

VIII. Experimental results

We have implemented all SAT algorithms presented so far in this paper on GeForce GTX 780 Ti. Since the number of memory banks and the number of threads in a warp is 32 [3], we have implemented SAT algorithms with $w = 32$. Barrier synchronization of all threads is implemented by invoking separated CUDA kernel calls. For example, one CUDA kernel call is invoked for each of $2\frac{\sqrt{n}}{w} - 1$ stages of 4R1W SAT algorithm. We have tested several configuration in terms of the number of threads in a CUDA block, and selected the best configuration. For example, in 2R2W, 4R4W, and 4R1W SAT algorithms, CUDA blocks with 64 threads each are invoked. Since each Stage i and each Stage $2\frac{\sqrt{n}}{w} - 1 - i$ $(0 \leq i \leq \frac{\sqrt{n}}{w} - 1)$ of 4R1W SAT algorithm computes $i+1$ values of the SAT, it uses $i+1$ threads in $\frac{i+1}{64}$ CUDA blocks. In 2R1W and 1R1W SAT algorithm, a 32×32 block in a matrix is copied to the shared memory in a streaming multiprocessor and the column-wise sums, the row-wise sums, and/or the SAT of it is computed. After that, the resulting values are copied to the global memory. For this operation, we use one CUDA block with 128 threads for each 32×32 block. All 128 threads in a CUDA block are used to copy a 32×32 block in the shared memory and 32 threads out of 128 threads are used to compute the column-wise sums, the row-wise sums, and/or the SAT.

Table II shows the running time of SAT algorithms for a double (64-bit) matrix of size from 1K×1K ($= 1024 \times 1024$) to 18K×18K ($= 18432 \times 18432$). Since a 18K×18K 64-bit matrix uses 2.53GBytes, it is hard to store a matrix larger than it in the global memory of GeForce GTX 780 Ti of size 3GBytes. The running time of the best SAT algorithm for each value of $\sqrt{n}$ is highlighted in boldface. Since 4R1W SAT algorithm performs a lot of kernel calls and stride memory access, and has large memory access latency overhead, it needs much more computing time than the other algorithms. Recall that 4R4W SAT algorithm corresponds to 2R2W SAT algorithm with transpose and 4R4W SAT algorithm performs much more memory access operations than 2R2W SAT algorithm. Since 2R2W SAT algorithm performs stride memory access, it is much slower than 4R4W SAT algorithm. These experimental results imply that stride memory access imposes a large penalty on the computing time.

Recall that 2R1W and 1R1W SAT algorithms are block-based algorithms, that perform $3n + O(\frac{n}{w})$ and $2n + O(\frac{n}{w})$ global memory access operations, respectively. Hence, they are faster than 4R4W SAT algorithm, which performs approximately $8n$ global memory access operations. Although 1R1W SAT algorithm performs fewer global memory access operations than 2R1W SAT algorithm, it runs slower when $\sqrt{n} \leq$6K. The reason is that 1R1W SAT algorithm has a larger latency overhead than 2R1W SAT algorithm and the latency overhead dominates the bandwidth overhead when the size of input is small. 1.25R1W SAT algorithm runs faster than both 2R1W and 1R1W SAT algorithms whenever $\sqrt{n} \geq$5K. We have evaluated the computing time for all possible values of r to find the best value r that minimize the running time of $(1+r)$R1W. Table II also shows the values of r $(0 < r < 1)$ that minimize the running time of $(1+r)$R1W SAT algorithm. From the table, we can see that $(1+r)$R1W SAT algorithm attain the best performance when $\sqrt{n} \geq$5K. Also, the value of r that gives the best performance decreases as the size of a matrix increases. This is because the memory bandwidth overhead of 1R1W SAT algorithm dominates the latency overhead for larger matrices and 1R1W SAT algorithm has better performance than 2R1W algorithm. We can conjecture that 1R1W SAT algorithm could be the best if an input matrix was much

Table II
THE RUNNING TIME OF SAT ALGORITHM (IN MILLISECONDS) AND THE VALUE OF r THAT MINIMIZE THE RUNNING TIME OF $(1+r)$R1W SAT ALGORITHM FOR MATRICES OF SIZES FROM 1K×1K TO 18K×18K

SAT Algorithms	1K	2K	3K	4K	5K	6K	7K	8K	10K	12K	14K	16K	18K
2R2W	1.47	3.28	5.71	9.53	13.6	23.9	27.1	47.8	90.8	163	160	234	401
4R4W	1.07	2.52	4.48	6.77	9.67	13.7	17.2	22.2	33.9	50.4	64.2	83.1	117
4R1W	11.5	22.9	36.4	50.1	113	104	173	252	315	597	437	742	1600
2R1W	**0.332**	**0.850**	**1.83**	**3.09**	4.79	6.78	9.25	12.3	18.9	27.2	36.8	48.7	61
1R1W	0.902	1.46	2.43	3.65	5.05	6.81	8.71	10.9	16.2	22.6	29.7	38	53.8
1.25R1W	0.453	1.05	1.96	3.25	4.71	6.41	8.47	10.8	16.5	23	31.2	40.7	57.6
fastest $(1+r)$R1W	0.365	0.958	1.94	3.16	**4.58**	**6.32**	**8.25**	**10.5**	**15.7**	**22.0**	**29.1**	**37.5**	**53.1**
r $(0<r<1)$	0.168	0.174	0.172	0.159	0.136	0.123	0.0876	0.103	0.0963	0.0710	0.0835	0.0694	0.0725
2R2W(CPU)	25.9	107	241	427	670	966	1310	1690	2670	3850	5250	6760	8670
4R1W(CPU)	18.0	73.2	165	293	459	660	904	1160	1830	2660	3600	4590	5950

larger than 18K×18K.

To see a speed-up factor of SAT algorithms running on the GPU over a conventional CPU, we have evaluated the performance of several sequential SAT algorithms on Intel Xeon X7460 (2.66GHz). Table II shows the running time of top two sequential algorithms as follows:

2R2W(CPU): The column-wise prefix-sums are computed in a raster scan order from the top row to the bottom row. More specifically, $a[i+1][j] \leftarrow a[i+1][j]+a[i][j]$ is executed in a raster scan order of (i,j). The row-wise prefix-sums are also computed in a raster scan order, that is, $a[i][j+1] \leftarrow a[i][j+1]+a[i][j]$ is executed in a raster scan order of (i,j).

4R1W(CPU): Formula (1) is evaluated in a raster scan order of (i,j).

From the table, we can see that 4R1W(CPU) SAT algorithm runs faster than 2R2W(CPU) SAT algorithm, because of the memory access locality. Also, $(1+r)$R1W SAT algorithm runs more than 100 times faster than 4R1W(CPU) SAT algorithm when $\sqrt{n} \geq$5K.

IX. CONCLUSION

The main contribution of this paper is to introduce the asynchronous Hierarchical Memory Machine, which capture the essence of CUDA-enabled GPUs. We have also presented a global-memory-access-optimal parallel algorithm for computing the summed area table on the asynchronous HMM. The experimental results on GeForce GTX 780 Ti show that our best algorithm, $(1+r)$R1W SAT algorithm, runs faster than any other algorithms for an input matrix of size 5K×5K or larger. It also runs at least 100 times faster than the best sequential algorithm running on a single CPU.

REFERENCES

[1] W. W. Hwu, *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, 2011.

[2] A. Uchida, Y. Ito, and K. Nakano, "Fast and accurate template matching using pixel rearrangement on the GPU," in *Proc. of International Conference on Networking and Computing*, Dec. 2011, pp. 153–159.

[3] NVIDIA Corporation, "NVIDIA CUDA C programming guide version 5.0," 2012.

[4] D. Man, K. Uda, H. Ueyama, Y. Ito, and K. Nakano, "Implementations of a parallel algorithm for computing euclidean distance map in multicore processors and GPUs," *International Journal of Networking and Computing*, vol. 1, no. 2, pp. 260–276, July 2011.

[5] NVIDIA Corporation, "NVIDIA CUDA C best practice guide version 3.1," 2010.

[6] K. Nakano, "Simple memory machine models for GPUs," in *Proc. of International Parallel and Distributed Processing Symposium Workshops*, May 2012, pp. 788–797.

[7] A. V. Aho, J. D. Ullman, and J. E. Hopcroft, *Data Structures and Algorithms*. Addison Wesley, 1983.

[8] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. C-21, pp. 948–960, 1972.

[9] K. Nakano, "The hierarchical memory machine model for GPUs," in *Proc. of International Parallel and Distributed Processing Symposium Workshops*, May 2013, pp. 591–600.

[10] F. Crow, "Summed-area tables for texture mapping," in *Proc. of the 11th annual conference on Computer graphics and interactive techniques*, 1984, pp. 207–212.

[11] A. Lauritzen, "Chapter 8: Summed-area variance shadow maps," in *GPU Gems 3*. Addison-Wesley, 2007.

[12] K. Nakano, "Optimal parallel algorithms for computing the sum, the prefix-sums, and the summed area table on the memory machine models," *IEICE Trans. on Information and Systems*, vol. E96-D, no. 12, pp. 2626–2634, 2013.

[13] D. Nehab, A. Maximo, R. S. Lima, and H. Hoppe, "GPU-efficient recursive filtering and summed-area tables," *ACM Trans. Graph.*, vol. 30, no. 6, p. 176, 2011.

[14] A. Kasagi, K. Nakano, and Y. Ito, "An optimal offline permutation algorithm on the hierarchical memory machine, with the GPU implementation," in *Proc. of International Conference on Parallel Processing*, Oct. 2013, pp. pp. 1–10.

編集距離空間における1-挿入被覆符号のサイズ限界式

田中 俊介　　泉 泰介

1　導入

類似ジョインとは入力集合から類似ペアを全列挙する問題である．本研究は等長文字列からなる編集距離空間上の類似ジョイン，すなわちアルファベット Σ 上の長さ n の文字列の集合 $S \subseteq \Sigma^n$ を入力とし，$s_1, s_2 \in S$，$ed(s_1, s_2) \leq \epsilon$ となる対 (s_1, s_2) を全出力する問題を，並列処理フレームワークの代表的なプラットフォームである MapReduce 上で処理することを考える [1]．ここで ϵ は閾値パラメータであり，$ed(s_1, s_2)$ は s_1 と s_2 の間の編集距離とする。

MapReduce は mapper と reducer と呼ばれる 2 種類のプロセスにより構成される。mapper は入力データより複数の key-value ペアを生成する。生成された key-value に従って入力データを各 reducer に送信し，reducer は並列処理を行い出力を返す。この処理において複数の mapper および reducer が複平行動作するが，通常，同一キー値を持つ key-value ペアは単一の reducer に集約されることが保障される。本研究では MapReduce 上で類似ジョインを計算するためのアルゴリズムの一つであるアンカーポイントアルゴリズムに注目する。アンカーポイントアルゴリズムでは入力文字列集合 Σ^n を被覆する被覆符号 C（アンカーポイント集合）を事前に用意する．ここで，被覆符号 C とは文字列の集合（長さ n とは限らない）で，任意の文字列 $s \in \Sigma^n$ に対して $ed(c, s) \leq \delta$ を満たす文字列 $c \in C$ が存在するようなものである．アンカーポイントアルゴリズムでは，各入力文字列 $s \in S$ について，自身からの距離が $\delta + \epsilon/2$ 以内に存在する（すなわち，自身の被覆している）C 中のすべての語 $c_1, c_2, \cdots, c_k$ に対して，$(c_1, s), (c_2, s), \cdots (c_k, s)$ を生成する．その後，共通のキー値を持つすべての文字列対にのみ全対比較を行うことで類似ペアを列挙する．アンカーポイントアルゴリズムにおいて，生成される key-value ペアの個数を出来るだけ小さくするためには，被覆符号 C が重複して被覆する領域を出来るだけ小さくすることが望ましい．そのため，被覆効率のよい符号 C の構成可能性，およびその限界を知ることは重要な問題である．しかしながら，被覆の重複量を正確に評価することは容易ではない．そこで，本研究では単純に可能な限り語数の少ない被覆符号を構成することに注目し，特に $\Sigma = 0, 1$ における 1-挿入被覆符号のサイズ限界式について考察する．

2　1-挿入被覆集合

Σ^n 上の 1-挿入被覆集合 C とは，Σ^{n+1} の部分集合であり，任意の $s \in \Sigma^n$ に対して適切な位置に Σ 中の 1 文字を挿入することで C 中にある語に変形可能であるものである．1-挿入被覆符号は Σ^n 上の編集距離空間における $\epsilon = 2$ の被覆符号となる．また，符号語 $c \in C$ が被覆する Σ^n 中の領域は，c から任意の 1 文字を削除して得られる Σ^n 中の語の数に等しいため，その領域サイズは高々$(n+1)$ である．このことから，1-挿入被覆符号のサイズに対する自明な下界は $|\Sigma|^n/(n+1)$ となる．一方で，既知の結果として，サイズ $|\Sigma|^{n+1}/(n+1)$ の 1-挿入被覆符号が存在することが知られている [2].

2.1　1-挿入被覆集合のサイズ

以下に，今回新たに得られた 1-挿入被覆符号のサイズ限界式を示す.

Theorem 2.1 k を以下の不等式を満たす最大の自然数とする．

$$\sum_{i=0}^{n-k} 2\binom{n}{n-i}(n+1-i) \leq 2^n$$

また，上記の条件を満たす k の値に対して決まる左辺の値を v とする．このとき，任意の *1*-挿入被覆符号 C に対して，

$$|C| \geq \sum_{i=0}^{n-k} 2\binom{n}{n-i} + \lceil \frac{2^n - v}{k} \rceil$$

が成立する．

2.2 既存の結果との比較

既存の上下界と今回の限界式を数値計算を用いて比較した．結果を図 1 に示す．(横軸：文字列の長さ n，縦軸：被覆符号の個数)

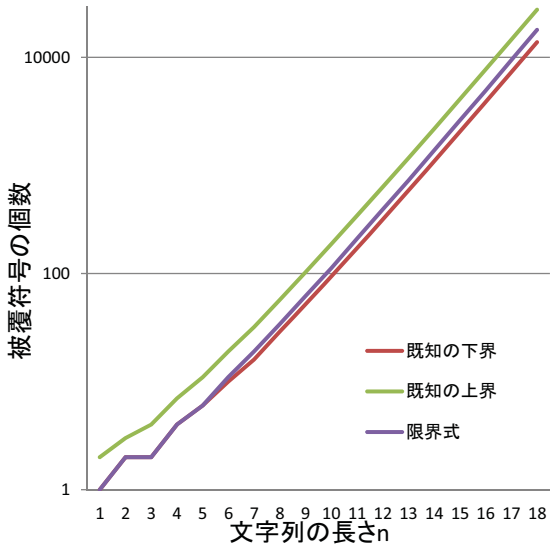


図 1: 既存の上下界と今回の限界式比較

参考文献

[1] F .N. Afrati, A. D. Sarma, D. Menestrina, A. G. parameswaran, and J. D. Ullman. Fuzzy Joins Using MapReduce. In ICDE,2012.

[2] F. Afrati, A. Das Sarma, A. Rajaraman, P. Rule, S. Salihoglu, and J. Ullman. Anchor points algorithms for hamming and edit distance.In Proceedings of ICDT, 2014.

GPGPUを用いた連結センサカバーに対する蜂群最適化

(A bee colony optimization algorithm for a connected sensor cover on GPGPU)

Yukihide Sasamura and Akihiro Fujiwara
Graduate School of Computer Science and Systems Engineering,
Kyushu Institute of Technology
Iizuka, Fukuoka, 820-8502, Japan
Email: o676113y@mail.kyutech.jp, fujiwara@cse.kyutech.ac.jp

***Abstract*—In the sensor network, a set of connected sensors that covers all discrete targets is called the connected sensor cover (CSC). The CSC with a small number of sensors is desirable because the small CSC can reduce network energy and communication costs.**

In the present paper, we propose an algorithms for CSC using an artificial bee colony optimization, which is an optimization technique based on behaviors of honey bees, on GPGPU. The experimental result shows that the execution on GPGPU is 5 times faster than the execution on CPU in case that the number of bees in the optimization algorithm is enough large.

I. Introduction

Sensor network is a wireless communication network, which is composed of small autonomous sensors. Each sensor can sense given targets if the target is in the sensing area, and communicate with the other sensors if the sensor is in the communication area. In the sensor network, a set of sensors is called *connected* if each sensor in the set can communicate with any sensor in the set using the others sensors as relays. A set of connected sensors can obtain a vast range of information across the network by communicating messages with the other sensors. In addition, a set of sensors is called *cover* if all discrete targets are covered with sensors in the set. Thus, the connected sensor cover (CSC) is defined as a set of connected sensors that cover all discrete targets.

Although construction of the CSC with the minimum number of sensors is NP-hard [1], the CSC with a small number of sensors is desirable because construction of the small CSC can reduce network energy and communication costs. Therefore, a number of algorithms [1], [2], [3], [4], [5], [6], [7] have been proposed for constructing CSC with a small number of sensors. For example, Jaggi et al. [2] proposed centralized algorithms using linear programming and a greedy method. Cardei et al. [4] proposed another centralized approximation algorithms, and showed that their algorithms can be easily applied to a distributed environment.

In addition, some algorithms have been proposed for CSC using particle swarm optimization techniques. Begum et al. [3] proposed algorithms based on ant colony optimization, and Shimokawa and Fujiwara [7] proposed algorithms based on artificial bee colony optimization. Although the above algorithms with the swarm optimization obtain CSC with a small numbers of sensors, the algorithms are time-consuming, and huge computational powers are needed to execute the proposed algorithms.

In this paper, we propose an algorithms for CSC using an artificial bee colony optimization on GPGPU. GPU (Graphics Processing Unit) is a hardware device equipped with a number of small processors specialized in graphics processing, and GPGPU (General Purpose computation on GPU) is general parallel computation on GPU. Since recent GPU is developed according to CUDA (Compute Unified Device Architecture), we can implement proposed algorithms on GPGPU as programs for parallel processing.

We implement our proposed algorithm using CPU and GPU, and evaluate validity of the proposed algorithm. The experimental result shows that the execution on GPGPU is 5 times faster than the execution on CPU in case that the number of bees in the optimization algorithm is enough large.

II. Preliminaries

A. Sensor Model

In this paper, the sensor network $G = (V, E)$ is defined by a set of sensors $V = \{s_1, s_2, \ldots, s_n\}$, where n is the number of sensors, and the set of links E that is a set of communication links between sensors. The set of discrete targets is represented by $T = \{t_1, t_2, \ldots, t_m\}$, where m is the number of discrete targets. Each sensor s_i has an unique identification number ID_i. The sensors are deployed on two-dimensional plane R, and each sensor knows the geographical location of itself.

Figure 1 is the sensor model used in this paper. Each sensor s_i has communicating area C and can communicate with other sensors in the communication area. In case that sensors s_i and s_j can communicate each other, a communication link $e_{ij} \in E$ exists between the sensors s_i and s_j on a graph G, and the sensors are called adjacent. We assume that direct communication of messages is possible for only between adjacent sensors without collision. In addition, two sensors are called connected if there is a path of adjacent sensors between two sensors in G.

We also assume that each sensor s_i has a circular sensing area $S \subseteq C$, and can sense a target in the sensing area. We

call that a sensor s_i covers a target t if t is in sensing area of s_i.

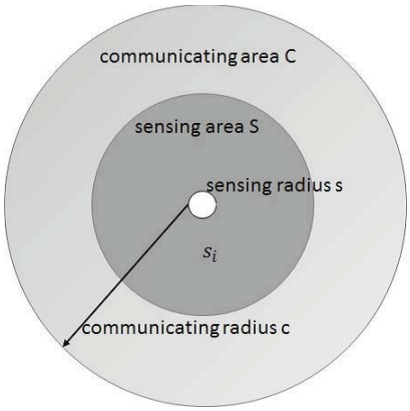


Fig. 1. A sensor model

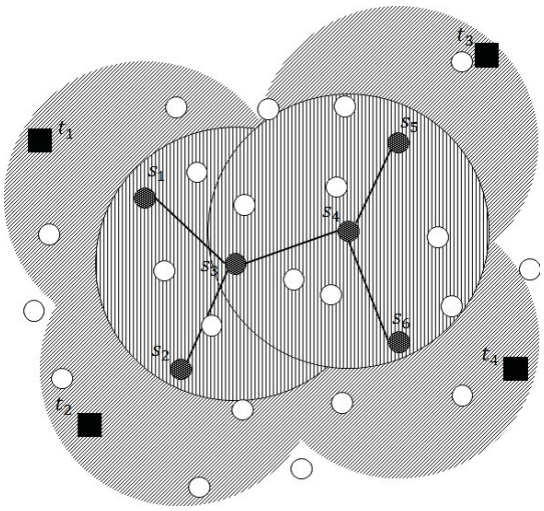


Fig. 2. An example of CSCDT

B. Connected Sensor Cover for discrete targets

Since each sensor equips a limited battery, coverage with a fewer number of sensors should be desired for discrete targets. However, connectivity of sensors is needed to communicate data of targets in the sensor network. As a set of sensors that satisfies these conditions, the connected sensor cover is defined as follows.

Definition 1 (Connected sensor cover for discrete targets): Given a sensor network $G = (V, E)$ and a set of discrete targets $T = \{t_1, t_2, \ldots, t_m\}$, the subset of sensors $M = \{s_{i_1}, s_{i_2}, \ldots, s_{i_n}\}$ $(M \subseteq V)$, which satisfies the following condition, is called the connected sensor cover for the discrete targets (CSCDT).

1) M is connected.
2) Any target in T is in at least one of sensing areas of sensors in M. □

The targets covering problem using line segments is known as NP-hard, and CSCDT with the minimum number of sensors is also known as NP-hard by reducing from the problem [4]. Therefore, a smaller number of sensors should be desired for CSCDT.

In this paper, we assume that the number of sensors n are enough large to construct CSCDT. In other words, CSCDT, which covers a set of input discrete targets, always exits for an input set of sensors.

Figure 2 is an example of CSCDT. In this example, the set of sensors $\{s_1, s_2, \ldots, s_6\}$ is CSCDT of the input sensors. The set of sensors $\{s_1, s_2, s_5, s_6\}$ covers a set of discrete targets $\{t_1, t_2, t_3, t_4\}$ with an union of sensing areas, and maintain the connectivity of sensors by including a set of sensors $\{s_3, s_4\}$ as relays.

III. A PROCEDURE FOR THE MINIMAL CSCDT

In this section, we explain a procedure for constructing a minimal CSCDT. We first define redundant sensors for CSCDT, and next propose a procedure, which deletes the redundant sensor, as a basic operation in the proposed algorithm.

Definition 2 (A redundant sensor for CSCDT): Let S_i be sensing area of sensor s_i in the sensor network $G = (V, E)$, and also let $M \subseteq V$ be CSCDT for the sensor network. We also assume that T is a set of discrete targets. Then, a sensor $s_d \in M$ is a redundant sensor for M if the sensor satisfies the following two conditions.

1) $M - \{s_d\}$ is connected.
2) Any target in T is in at least one of sensing areas of sensors in $M - \{s_d\}$. □

Using the above definition, we propose a procedure, which is called *Repeated deletion*, for deleting sensors from CSCDT.

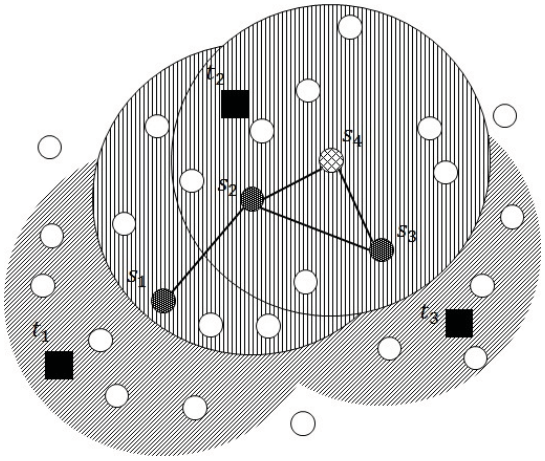
Repeated deletion

Repeat the following two steps for each sensor in an input set of sensors until no sensor is deleted in the second step.

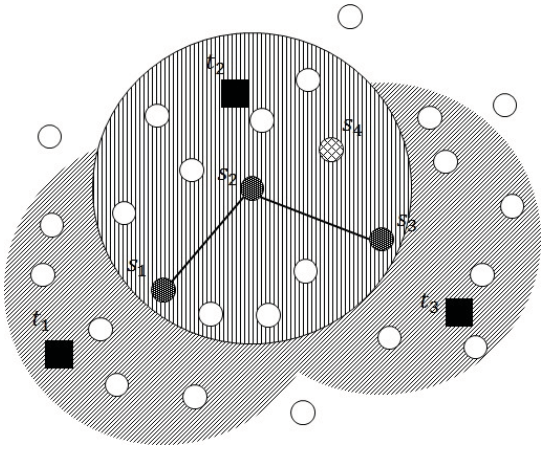
Step 1: Check whether the sensor is a redundant sensor or not for CSCDT.
Step 2: Delete the sensor in case that the sensor is redundant. □

The procedure, *Repeated deletion*, ensures connectivity and coverage of the sensor network. In addition, the obtained CSCDT is apparently minimal due to the end condition of the procedure.

Figure 3 shows an example for a redundant sensor and the procedure. Figure 3 (a) shows an input CSCDT, and the set of sensors $\{s_1, s_2, s_3, s_4\}$ is in CSCDT. An union of sensing areas of the set of sensors $\{s_1, s_3\}$ covers the set of discrete targets $\{t_1, t_2, t_3\}$, and the sensors are connected. In this case, sensor s_4 is redundant, and is deleted if two steps in *Repeated deletion* is executed for s_4. Then, we obtain CSCDT in Figure 3 (b) after deletion of s_4.



(a) An input CSCDT



(b) A CSCDT after deletion of a redundant sensor

Fig. 3. An example of *Sensor reduction*

IV. An algorithm using bee colony optimization on GPGPU

A. Artificial bee colony optimization for CSCDT

In this section, we explain an algorithm using artificial bee colony optimization algorithm for CSCDT. The algorithm is a revised version of an algorithm proposed in [7].

We first explain an outline of the artificial bee colony optimization. The artificial bee colony (ABC) optimization [8] is an optimization technique based on the following habit of honey bees. The bee gathers honey outside region, and shares information of gathered honey with other bees when bee arrives at comb. Then, each bee decides a next way of exploration using exchanged informations. As a result, the honey bees can collect high-quality honey.

In our algorithm for CSCDT, ABC optimization is used for reducing the number of sensors. We assume m bees, and B_j $(1 \leq j \leq m)$ denotes j-th bee used for optimizing CSCDT.

We next introduce an operation, which is called *Sensor reduction*. Let $G = (V, E)$ be an input sensor network, and $G' = (V', E')$ be an CSCDT for G. The following *Sensor reduction* is executed by each bee for reducing the number of sensors in CSCDT.

Sensor reduction

Step 1: Select NC sensors in V' randomly. (The NC is a parameter for optimization. and we assume V_c denotes a set of selected sensors.)

Step 2: For each sensor $s_i \in V_c$, execute the following sub-steps.

(2-1) For each adjacent sensor s_j of s_i, compute V'' such that $V'' = V' - \{s_i, s_j\}$, and then, check whether there exists a sensor $s_k \in V - V'$ such that $V'' \cup \{s_k\}$ is CSCDT. If the sensor s_k exists, the two sensors, s_i and s_j, are removed from V' and s_k is added to V'.

Using the above *Sensor reduction* and *Repeated deletion*, the algorithm for CSCDT using ABC optimization is given below.

An algorithm for CSCDT using ABC optimization

Step 1: Construct the minimal CSCDT V' for $G = (V, E)$ using *Repeated deletion*. The obtained CSCDT is copied to all m bees. (We assume that bee B_j stores the CSCDT as $CSCDT_j$.)

Step 2: The following 3 sub-steps are repeated by a given number of trials u, and output a CSCDT with the minimum number of sensors among all bees.

(2-1) Each bee B_j executes *Sensor reduction* for $CSCDT_j$, and set evaluation value O_j to the number of deleted sensors by *Sensor reduction*.

(2-2) Each bee B_j decides whether to become a recruiter or a follower according to the following probability F_j, where $O_{\max} = \max\{O_k \mid 1 \leq k \leq n\}$ and t is the number of repetition in Step 2.

$$F_i = e^{-\frac{O_{max} - O_i}{t}}$$

(2-3) Each bee B_j maintains $CSCDT_j$ in case of the recruiter. On the other hand, in case of the follower, B_j selects one of CSCDTs owned by recruiters according to following probability R_i, where R is a set of recruiters.

$$R_i = \frac{O_i}{\sum_{O_k \in R} O_k}$$

B. Implementation on GPGPU

In this section, we explain an implementation of our algorithm for CSCDT using ABC optimization on GPGPU. For parallel processing on GPGPU, sub-step (2-1) in the algorithm is executed in parallel on GPGPU, and the other steps are executed serially.

Figure 4 illustrates parallel execution of the proposed algorithm. Each bee in (2-1) is assigned a single thread and executed on each core. Informations of sensors and targets are stored in constant memory, and variables of the kernel function are stored in a register. Informations for the other steps are stored in the global memory, and all informations are exchanged throughout the global memory.

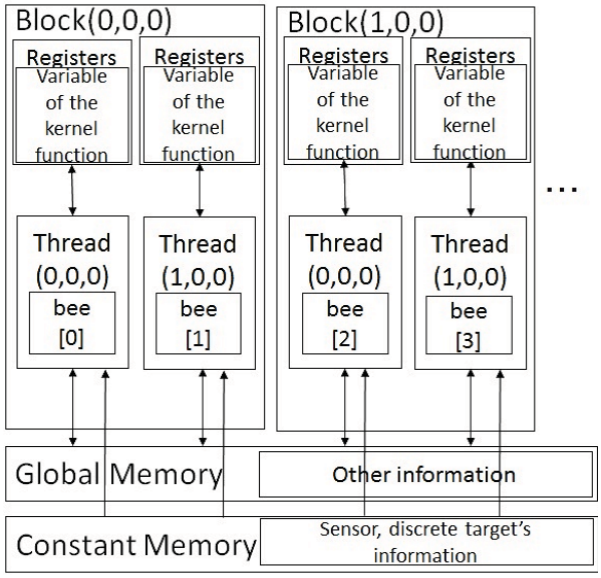


Fig. 4. a model of parallel processing of runtime

TABLE I
EXPERIMENTAL ENVIRONMENT

CPU	Intel Core i3-4130 (3.4GHz)
Main memory for CPU	8GB
GPU	NVIDIA GeForce GTX 760
The number of CUDA cores on GPU	1152
Memory on GPU	4GB
OS	CentOS release 6.5
Development Environment	CUDA 5.5

V. EXPERIMENTAL RESULTS

Our proposed algorithm is implemented on CPU on GPGPU, we compare execution times between the implementations.

We assume that an input region is a 100×100 square area, and sensors and targets are randomly located in the region. The numbers of sensors are 1000, and the number of targets is 20. Sensing radius and communication radius of each sensor are both 10.

In addition, the following parameters are set for proposed algorithms according to execution times.

- The number of bees used in ABC optimization: 32 to 65536
- The number of repetition in ABC optimization: $u = 16$
- The number of sensors that each bee selects randomly: $NC = 4$

We also show our experimental environment for CPU and GPU on Table V.

Figure 5 shows execution times for CPU and GPGPU in the simulation environment. The result shows that the execution on GPGPU is 5 times faster than the execution on CPU in case that the number of bees in the optimization algorithm is enough large.

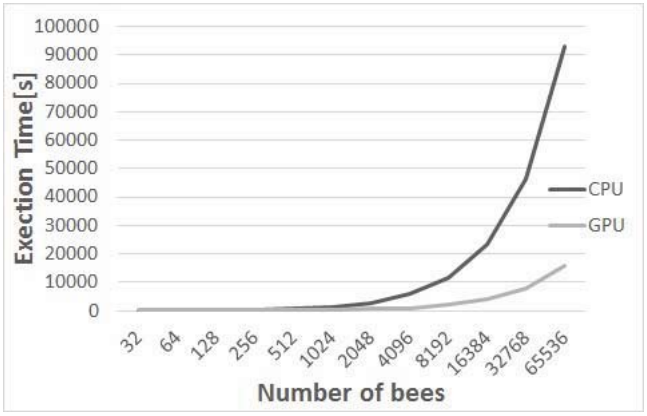


Fig. 5. Execution times on CPU and GPGPU

VI. CONCLUSIONS

In this paper, we proposed an algorithms for CSC using an artificial bee colony optimization on GPGPU. The experimental result shows that the execution on GPGPU is 5 times faster than the execution on CPU in case that the number of bees in the optimization algorithm is enough large.

In our future work, we are considering further improvement of the proposed algorithm using another optimization techniques, and we are also considering to simulate distributed algorithms on GPGPU.

REFERENCES

[1] H. Gupta, Z. Zhou, S. Das, and Q. Gu, "Connected sensor cover:self-organization of sensor networks for efficient query execution," *ACM/IEEE Transactions on Networking*, 14(1), 2006.
[2] N. Jaggi and A. Abouzeid, "Energy-efficient connected coverage in wireless sensor networks," *4th Asian International Mobile Computing Conference*, 2006.
[3] S. Begum, N. Tera, and S. Sultana, "Energy-efficient target coverage in wireless sensor networks based on modified ant colony," *International Journal of Ad hoc, Sensor & Ubiquitous Computing*, vol. 1, no. 4, 2010.
[4] I. Cardei and M. Cardei, "Energy-efficient connected-coverage in wireless sensor networks," *International Journal of Sensor Networks*, vol. 3, no. 3, 2008.
[5] M. Lu, J. Wu, M. Cardei, and M. Li, "Energy-efficient connected coverage of discrete targets in wireless sensor networks," *International Journal of Ad Hoc and Ubiquitous Computing*, 2009.
[6] K. Nakamoto and A. Fujiwara, "Distributed algorithms for 2-connected sensor cover in sensor network," in *Proceedings of the International Conference on Wireless Networks*, 2010.
[7] T. Shimokawa and A. Fujiwara, "Centralized algorithms for the connected target coverage in wireless sensor networks," in *Proceedings of 3rd International Workshop on Advances in Networking and Computing*, 2012, pp. 307–310.
[8] D. Karaboga, "An idea based on honey bee sarm for numerical optimization," Erciyes University, Tech. Rep. TR06, 2005.

GPUを用いた並列ソートアルゴリズム

小池 敦[1,a)]　定兼 邦彦[2,b)]

概要：GPU を用いた並列比較ソートアルゴリズムを扱う．GPU 向けの高速なソートアルゴリズムとして，Merrill らの提案する高速 Radix ソート [14] が知られている．本論文では，まずその計算量について解析したのち，本アルゴリズムを改良することで新しいアルゴリズムを二つ提案する．一つは MSD Radix ソートであり，これは分散システム等におけるアルゴリズムを設計する際に有効である．もう一つは Splitter-based ソートである．これは比較ソートであるため，キーの性質に依らずに使用することができる．

1. はじめに

プロセッサの動作クロック周波数の向上は限界を迎えており，周波数向上に代わるパフォーマンス向上の手段として並列アーキテクチャが注目されている．GPU (Graphics Processing Unit) は元々はグラフィック処理のための専用プロセッサとして開発された．しかし，非常に高い並列性を持っていることから，グラフィック処理以外にも GPU が使われ始めている．汎用の処理に GPU を使用することは GPGPU (general-purpose GPU) と呼ばれており，安価に超並列環境が構築できることから注目されている．

GPU は多数のコアを用いて効率よく処理を行うため，特殊なアーキテクチャとなっている．GPU プログラミングにおいては，このアーキテクチャを適切に考慮する必要がある．NVIDIA 社は GPGPU のための開発環境として，CUDA[15] を提供しており，CUDA 上で開発することにより，様々な GPU モデル上で動作するプログラムを実装することができる．しかし，最適なパフォーマンスを得るためには，GPU アーキテクチャを適切に考慮してアルゴリズムを設計する必要がある．

逐次アルゴリズムの評価では，RAM(Random Access Machine) モデル上での漸近解析が一般的に行われている．RAM モデルはすべての逐次実行マシンに対する抽象化となっており，RAM モデルを用いて漸近解析を行うことで，デバイスの仕様や入力データの値に依らない汎用的なアルゴリズムの性能を知ることができる．一方，並列実行マシンには，RAM モデルのような共通の抽象化が存在しない．並列アルゴリズムの漸近解析に一般的に使用されているモデルに PRAM モデル [6] があるが，PRAM モデルは GPU アーキテクチャとは大きく異なっており，GPU 向けアルゴリズムの性能を正しく評価できない．[13] では GPU における実際の計算実行時間を精度よくシミュレートすることについて検討されているが，計算実行時間は GPU のモデルに大きく依存するため，GPU 向けアルゴリズムの汎用的な性能評価とならない．筆者らは GPU 向けアルゴリズムを漸近解析するための並列計算モデルとして AGPU モデルを提案している [11]．アルゴリズムの正確な計算量はデバイス仕様に依存するが，AGPU モデル上で解析された計算量の高々定数倍である．AGPU モデルにより，GPU デバイスの仕様や入力データの値に依らない汎用的なアルゴリズムの性能を知ることができる．

本報告では，GPU 上でのソートアルゴリズムを扱う．これまでにも多くの GPU 向けソートアルゴリズムが提案されている [2], [7], [8], [10], [12], [14], [16], [17], [18], [19], [21]. その中で最も高速なもののとして，Merrill らの提案する高速 Radix ソート [14] が挙げられる．本アルゴリズムは LSD Radix ソートをベースとしているが，グローバルメモリアクセス回数の削減およびグローバルメモリアクセスのレイテンシ隠ぺいの両方が適切に考慮されており，従来アルゴリズムを大きく上回るパフォーマンスが得られている．本報告では，まず，AGPU モデルを改良することにより，上記アルゴリズムの計算量の漸近解析を行う．その後，上記アルゴリズムの変形版について 2 つ提案する．一つ目は MSD Radix ソートである．MSD Radix ソートはメモリアクセスが局所的になりやすい事が特徴である．よって分散環境におけるソートアルゴリズムを設計する場合への応用が容易である．二つ目は Spliter-based ソーティング

1　国立情報学研究所アーキテクチャ科学研究系
2　東京大学大学院情報理工学系研究科
a)　koike@nii.ac.jp
b)　sada@mist.i.u-tokyo.ac.jp

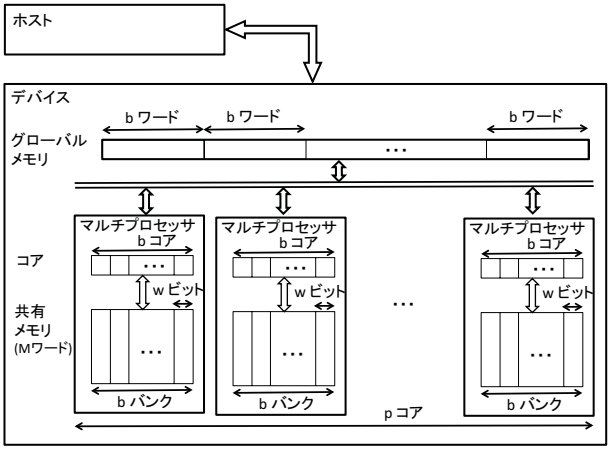


図 1　AGPU モデルのアーキテクチャ

である．Merrill らの高速 Radix ソートは比較ソートでないため，キーの性質によっては，利用できないことがある．そこで，上記の MSD ソートを改良することで比較ソートのアルゴリズムを提案する．提案アルゴリズムは，共に Merrill らの高速 Radix ソートと同様の効率的なメモリアクセスが可能である．

本論文の構成は以下の通りである．2 章で マルチスレッド対応 AGPU モデルについて説明する．次に，3 章において，Merrill らの高速 Radix ソートを紹介し，AGPU モデルでの計算量解析について述べる．4 章では，筆者らの提案する MSD Radix ソートについてアルゴリズムと計算量を説明する．5 章では，筆者らの提案する比較ソートアルゴリズムについて，詳細と計算量を説明する．6 章で結論を述べる．

2. マルチスレッド対応 AGPU モデル

AGPU モデル [11] は，GPU 向けアルゴリズムの設計と評価を行うための並列計算モデルである．AGPU モデルを用いることで，GPU デバイスの詳細仕様に依らない汎用的なアルゴリズム設計と評価を行うことができる．従来の AGPU モデルでは，並行して実行されるスレッド (ワープ) が共有メモリを共有することについて，考慮していなかった．そこで，本論文では AGPU モデルを改良する．まず，AGPU モデルのアーキテクチャを説明した後，GPU 向けアルゴリズムの評価基準について説明する．

2.1 アーキテクチャ

AGPU モデルのアーキテクチャを図 1 に示す．AGPU モデルのアーキテクチャは並列計算を行うためのデバイス (GPU) とデバイスを制御するためのホスト (CPU) の異種混載システムとなっている．デバイスは p 個のコアを備えている．コアのワード長は w ビットであり，コアはワード単位でデータにアクセスする．また，デバイスは k 個のマルチプロセッサで構成されており，各マルチプロセッサは

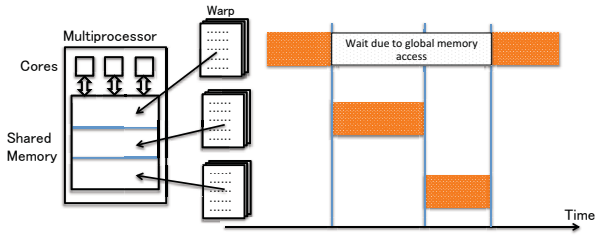


図 2　マルチスレッディングによるレイテンシ隠ぺいの例

b 個のコアを備えている．すなわち $p = kb$ である．マルチプロセッサはホストから起動されたプログラムを個別に実行する．すなわち，マルチプロセッサは他のマルチプロセッサとの通信手段および同期手段を持たない．ホストはすべてのマルチプロセッサの処理完了を待つことにより，マルチプロセッサ間の同期を行うことができる．しかし，マルチプロセッサの処理完了時，共有メモリのデータはすべて削除される．後で参照する必要があるデータはマルチプロセッサの処理終了時にすべてグローバルメモリに書き込む必要がある．

マルチプロセッサ内の b 個のコアは b 個のスレッドに対し，常に同一の命令を実行する．この時，各コアは同一命令を並列に実行するという．1 つのマルチプロセッサ内で並列に処理されるスレッドの集合をワープと呼ぶ．ただし，オペランドに指定されるデータアドレスについてはコアごとに指定することができる．また，命令には実行条件を含めることができ，条件を満たすコアのみ命令を実行させることができる．一方，各コアは複数のスレッドを時分割で切り替えながら同時実行することができる．この時，各コアは複数スレッドを並行に実行するという．言い換えれば，マルチプロセッサは複数ワープを並行に実行することができる．GPU のこの機能をマルチスレッディングと呼ぶ．各マルチプロセッサが並行に実行可能なワープの最大数を C とする．

マルチスレッディングにはグローバルメモリアクセスのレイテンシ（待ち時間）を隠ぺいする効果がある．すなわち，あるワープがグローバルメモリアクセスにより，待ち状態になっている場合に，マルチプロセッサは他のワープを実行することによりコアの使用率を高めることができる．図 2 に具体例を示す．マルチスレッディングは GPU における効率的なメモリアクセスのキーとなる技術である．

デバイスは 2 種類のメモリを備えている．1 つ目はグローバルメモリである．これは低速であるが大容量であり，すべてのマルチプロセッサおよびホストからアクセス可能である．グローバルメモリは b ワードごとのブロックに分割されている．マルチプロセッサ内の全コアが同一ブロックにアクセスする時，1 回のメモリアクセスで全コア分のデータにアクセスすることができる．これはコアレッシングと呼ばれており，処理時間に大きな影響を与える．

一方，コアが複数の異なるブロックにアクセスする時は，各ブロックのに対して 1 回のアクセスが必要となる．2 つ目は共有メモリである．各マルチプロセッサは内部に容量 M ワード ($b \leq M$) の共有メモリを備えている．これは高速であるが低容量である．また，マルチプロセッサ内部のコアからのみアクセス可能である．共有メモリは b 個のバンクから構成されており，マルチプロセッサ内の b 個のコアのそれぞれが異なるバンクにアクセスする時，単位時間でデータにアクセスできる．一方，複数のコアが同一のバンクにアクセスする時は，処理がシリアライズされる．これはバンクコンフリクトと呼ばれており，これも処理時間に大きな影響を与える．

以上で定義される計算モデルを AGPU(p, b, M, C, w) と記載する．ただし M, C, w については，省略される場合がある．

2.2 アルゴリズムの評価基準

AGPU モデルではアルゴリズムを計算量，メモリ使用量，多重度を使用して評価する．以下では，計算量とメモリ使用量について説明し，多重度については，次節で説明する．

まず，アルゴリズムの計算量を評価する基準として，時間計算量と I/O 計算量を使用する．時間計算量は，各マルチプロセッサで実行されるプログラムの命令発行数である．マルチプロセッサが複数のワープを並行に実行する場合は，すべてのワープの命令発行数の合計値となる．共有メモリへのアクセスでバンクコンフリクトが発生する場合，コンフリクト数に応じた時間が時間計算量に加算される．また，グローバルメモリへのアクセスについては，b ワードのブロックに対する書き込みまたは読み込みの時間計算量を 1 とする．マルチプロセッサごとに命令発行数が異なる場合には，最も多い発行数を時間計算量とする．I/O 計算量については，上記で説明したグローバルメモリアクセス回数のすべてのマルチプロセッサでの合計値とする．I/O 計算量を時間計算量とは別に評価する理由は，グローバルメモリアクセス処理に要する時間が他の処理に比べて大きくなるためである．また，グローバルメモリに対しては，同時にアクセスできるマルチプロセッサの数が限られているため，アクセス回数については，すべてのマルチプロセッサでの合計値とする．

次に，メモリ使用量を評価する基準として，グローバルメモリ使用量と共有メモリ使用量を使用する．共有メモリ使用量は各マルチプロセッサで使用されるメモリ使用量の最大値とする．大規模データを扱う場合，グローバルメモリ使用量を少なくすることは特に重要である．また，共有メモリ使用量は M ワード以下にする必要がある．また，共有メモリ使用量は次節で説明する多重度にも影響する．

2.3 マルチスレッディングの効果

2.1 節で述べた通り，マルチスレッディングは GPU におけるメモリアクセスのキーとなる技術である．しかし，I/O 計算量の値はマルチスレッディングの効果とは無関係であるため，マルチスレッディングの効果を I/O 計算量を用いて評価することはできない．本節ではマルチスレッディングの効果を評価する値として多重度を導入する．

マルチスレッディングの効率を上げるためには，マルチプロセッサに割り当てるワープ数を増やせば良い．1 つのマルチプロセッサに C 個のワープを割り当てる時，マルチスレッディングの効果は最も高くなる．

また，十分な数のワープが生成されている時，GPU ではユーザの設定に関わらず，1 つのマルチプロセッサにより多くのワープを割り当てようとする．これにより，マルチスレッディングの効果を高めることができる．しかし，マルチプロセッサに常に C 個のワープが割り当てできるとは限らず，割当数は共有メモリ使用量に依存する．マルチプロセッサ内のすべてのワープは同一の共有メモリを使用するため，全ワープでの共有メモリ使用量の合計値が共有メモリサイズを超えることはできない．

多重度はこれらの効果を見積もるために導入される．AGPU(p, b, M, C) 上で設計されたアルゴリズムについて，共有メモリ使用量を m，マルチプロセッサごとの使用ワープ数を c とすると，多重度 $\mathcal{M}$ は $\mathcal{M} := Mc/m$ と定義される．これは CUDA のオキュパンシに対応する値であるが，多重度は AGPU モデルのパラメータを使用して計算することができる．多重度の値が C 以下の時，値が大きいほどマルチスレッディングの効果が大きくなるが，C より大きくしても効果は大きくならない．共有メモリ使用量が大きく，かつ，マルチプロセッサへの割当ワープ数が小さい場合に多重度の値は小さくなり，マルチスレッディングの効果が小さくなる．

3. 既存の高速 Radix ソートの解析

本章では，Merrill らの高速 Radix ソート [14] について，AGPU モデルを用いて，アルゴリズムの概要説明と計算量の漸近解析を行う．

3.1 アルゴリズムの概要

Merrill らの高速 Radix ソート [14] は LSD Radix ソート [3] に分類される．すなわち，最下位桁から最上位桁の方向に順に各桁の値のみを用いてソートを行う．各桁のソートが安定の時，本ソートアルゴリズムは正しく動作する．彼らのアルゴリズムでは，各桁は $r = 2^d$ 個の数字（基数）で表現されるものとする．

次に各桁のソート処理について説明する．マルチプロセッサ数 ($k = p/b$) が 4，$r = 4$ の場合の例を，図 3 に示す．図 3 において，基数 r_1, r_2, r_3, r_4 は $r_1 < r_2 < r_3 < r_4$

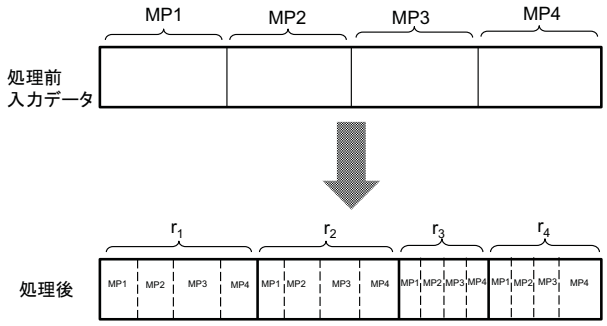


図 3　各桁のソート処理

を満たすものとする．各マルチプロセッサは入力データの連続する要素を均等に割り振られ，担当するデータを処理する．出力は図 3 の下図の様になる．まず，最小の基数 r_1 について各マルチプロセッサの担当する入力データのうち，r_1 に属するものが出力され，次に r_2, r_3, r_4 の順に同様に出力される．各マルチプロセッサが並列に処理を行うためには，処理したデータをどこに出力するかについて，あらかじめ計算しておく必要がある．そこで，各桁の処理を以下の 3 ステップで構成することにする．

(1) Bottom-level Reduction
(2) Top-level Scan
(3) Bottom-level Scan/Scatter

本節では，図 3 処理後データの 16 個の領域のそれぞれをブロックと呼ぶ事にする．Bottom-level Reduction では，マルチプロセッサごとに各基数に属する要素の数を計算する．これにより，図 3 下図の 16 個のブロックそれぞれのサイズがわかる．この計算は，Harris らの提案する GPU 向け高速 Reduction アルゴリズム（Cascading アルゴリズム）[9] を用いて行うことができる．次に，Top-level Scan では，各ブロックの先頭アドレスを計算する．これは Bottom-level Reduction で得られた各ブロックの要素数を格納した配列に対して Prefix Scan の処理を行うことで得る事ができる．GPU 向けの Prefix Scan アルゴリズムとして Tree-based アルゴリズム [20] が知られている．次に，Bottom-level Scan/Scatter では，入力のそれぞれの要素について適切なアドレスへのデータコピーを行う．前のフェーズにおいて，各ブロックの先頭アドレスが分かっているため，各マルチプロセッサは並列に処理を行うことができ，マルチプロセッサ間の情報交換は不要である．以下，各マルチプロセッサの処理を説明する．この処理を効率良く行うため，彼らは *“multi-scan”* という方法を提案している．

multi-scan では Dotsenko らの提案する高速 prefix scan アルゴリズム（Matrix-based アルゴリズム） [4] を使用する．Matrix-based アルゴリズムでは各マルチプロセッサは担当する入力データをサイズ ab の小ブロックに分割し (b はマルチプロセッサ内のコア数，a はチューニングパラメータ)，各小ブロックをシーケンシャルに処理する．a の値については，大きいほど時間計算量が小さくなるものの，マルチスレッディングの効率が下がる事が知られている．筆者らはこの事について，AGPU モデルを用いて解析を行っている [22].

multi-scan では入力の各データの出力アドレスを計算するため，基数ごとにワープを生成する．各ワープは担当する基数に属する入力データをスキャンすることにより，それらの出力先アドレスを計算する．この計算を行うために Dotsenko らの提案する高速 prefix scan アルゴリズムを使用する．最後にデータを指定のアドレスに出力するが，グローバルメモリアクセス回数を減らすため，連続するアドレスに出力されるデータを一旦共有メモリの連続する領域に書き込んだのち，出力される．これにより，グローバルメモリへのコアレスアクセスが行われやすくなる．

3.2　計算量の解析

本アルゴリズムの各桁の処理は基数ごとの Prefix Scan の処理に Scatter（データ出力）処理を追加したものとなっている．Scatter 処理で共有メモリにデータをコピーする際にバンクコンフリクトが発生するため，時間計算量は Prefix Scan よりも大きくなる．しかし，$r = a$ とすると，I/O 計算量は Prefix Scan と同様になる．また，入力をビット長を w とすると，各桁の処理は $w/\log r$ 回繰り返される．

入力要素数 N がコア数 p よりも十分大きい時アルゴリズム全体の計算量はアルゴリズム全体の計算量は，表 1 のようになる．

4.　MSD Radix ソート

本章では 3 章で紹介した Merrill らの高速 Radix ソート [14] を変更することにより，高速な MSD Radix ソートアルゴリズムを提案する．MSD Radix ソートは，最上位桁から最下位桁の方向に順に各桁の値のみを用いてソートを行う．MSD Radix ソートは複数 GPU による Radix ソートを設計する場合などへの応用が容易である．また，5 章の Splitter-based ソートは本章のアルゴリズムを変更したものである．

4.1　アルゴリズムの概要

本章以降では，各桁のソート処理をフェーズと呼ぶ．最初のフェーズは最上位桁に対して Merrill らの高速 Radix ソート [14] と同様の処理を行う．次のフェーズに関しては，Merrill らの Radix ソートを修正する必要がある．なぜならば，MSD から LSD 方向への Radix Sort では，前フェーズで基数により区切られた領域を別々に処理する必要があるためである．

2 フェーズ目以降では，各領域へのマルチプロセッサの割当は図 4 のように行う．まず，各マルチプロセッサに対

表 1　入力をビット長 w の整数としたときの，アルゴリズム全体の計算量

	I/O 計算量	時間計算量	多重度
LSD Radix ソート	$\mathcal{O}\left(\frac{Nw}{b\log r}\left(1+\frac{a}{r}\right)\right)$	$\mathcal{O}\left(\frac{nw}{p\log r}\left(a+\frac{r\log b}{a}\right)\right)$	$\mathcal{O}\left(\frac{M}{ab}\right)$

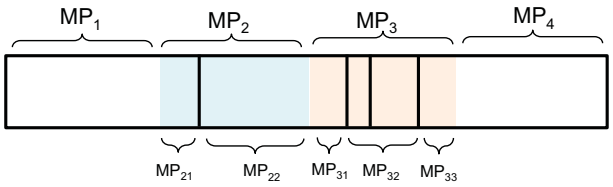
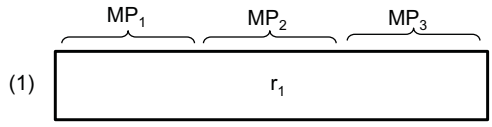


図 4　マルチプロセッサへの要素の割当



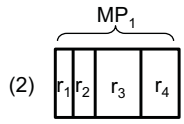


図 5　基数領域と分割領域の関係

し，均等に領域を割り当てる．そして，割当領域に複数の基数領域がある場合には，以下のようにさらに領域を分割する．

- 複数領域のうち最も左の基数領域
- 複数領域のうち最も右の基数領域
- それ以外の領域

このようにしても分割された領域の数は高々 3 倍にしかならない．

次に，分割された領域を各マルチプロセッサに同じ数だけ分配する．この時，各マルチプロセッサに割り当てられる要素数は均等に分配した場合の高々 3 倍となる．

また，分割された領域と基数領域の関係は必ず以下のどちらかになる．図 5 に具体例を示す．

(1) 一つの基数領域が複数の分割領域で構成される
(2) 複数の基数領域で一つ分割領域が構成される

すなわち，(2) の場合の除けば，基数領域の区切りは必ず分割領域の区切りとなっている．

図 5 の (1) の場合，マルチプロセッサは最初のフェーズと同様の各桁のソート処理を行う．(2) の場合は，マルチプロセッサはシーケンシャルに各基数領域を処理する．

以上を LSD まで繰り返す事で，Radix sort を行うことができる．

計算量は Merrill らの LSD Radix ソートと同様になる．

5. Splitter-based ソート

4 章の入力データ振り分け処理を，基数を用いずにピボット集合を用いて行うように変更する．ピボット集合を用いる場合，ピボット集合の選び方が問題となる．ピボット集合により振り分けられた各集合のサイズに偏りが有る場合，より多くのフェーズが必要となり，計算量が大きくなる．

本章では，Aggarwal ら [1] の I/O 計算量最適な Distribution ソートのアイデアを用いて適切にピボット集合を選択することについて検討する．ただし，このアルゴリズムをそのまま適用すると，多重度が小さくなり．マルチスレッディングの効果が下がるので，多重度を大きく保てるように工夫する．

以下では，まず Aggarwal らの Distribution ソートについて説明した後，筆者らの提案するアルゴリズムについて説明する．

5.1 Aggarwal らの Distribution ソートについて

Aggarwal ら [1] の Distribution ソートも，1 フェーズの処理を繰り返すことでソートを行う．ただし，各フェーズは以下の 2 ステップからなる．

(1) ピボット集合を算出する
(2) ピボット集合を用いて，データを分割する

Radix ソートにおける各桁の処理と比較すると，ピボット集合を算出する処理が追加されている．(2) のデータ振り分け処理については，MSD Radix ソートと同様のアルゴリズムを用いることができるので，以降では (1) について説明する．

Aggarwal らの Distribution ソートは I/O モデルと呼ばれる計算モデル上で設計されている．標準的な I/O モデルは 1 つのプロセッサ，M 要素を格納できる 1 つの内部メモリおよび 1 つの外部メモリ (ディスク) から構成される．プロセッサは単位時間あたりに外部メモリの連続した b レコードからなるブロックにアクセスすることができる．アルゴリズムはブロックの転送回数 (I/O 計算量) で評価される．本モデルを I/O(b, M) と記述する．

Aggarwal らの Distribution ソートは，外部メモリに保存された要素数 n の入力データに対し，最適な I/O 計算量でソートを行う．図 6 に処理の流れを示す．まず，入力を要素数 M のメモリロードに分割する．そして，各メモリロードをソートする．次に，各メモリロードから間隔 $S/4$ ごとにピボット集合を取り出す（S の値は最後に決める）．本論文ではこのピボットをローカルピボットと呼ぶことにする．次に全メモリロードからのローカルピボットを結合する．これを本論文ではローカルピボット集合と呼ぶことにする．最後にこの集合から等間隔に S 個のピボットを取り出す．すなわち，i 番目のグローバルピボットはローカルピボット集合の中で $4iN/S^2$ 番目に小さい要素となる．グ

ローバルピボット集合の抽出は線形時間セレクション [5] を S 回行うことでできる．この時，I/O 計算量も入力サイズに線形となるようにできる．

次に index i のグローバルピボットの入力データ中での rank (何番目に小さいか) の値を検討する．ローカルピボット集合の中で，このグローバルピボット以下の値を持つものは（自身も含めて）i 個であり，ローカルピボットのピボット間隔が $S/4$ なので，取りうるランク値の最小値は $rank(i) \geq \frac{4iN}{S^2} \cdot \frac{S}{4} = \frac{iN}{S}$ となる．また，ランク値は以下の値よりは小さくなる．$rank(i) < \frac{iN}{S} + \frac{N}{M}\frac{S}{4} < \frac{N}{S}\left(i + \frac{1}{4}\right)$

これより，グローバルピボットによる振り分け後の各領域のサイズは $\frac{5}{4}\frac{N}{S}$ 未満となる．ここで，$S = \sqrt{\frac{M}{b}}$ とすると，フェーズの回数は高々 $\mathcal{O}\left(\log_{\frac{4}{5}S} \frac{N}{b}\right) = \mathcal{O}\left(\frac{\log_{\frac{M}{b}} \frac{N}{b}}{\log_{\frac{M}{b}} \frac{4}{5}\sqrt{\frac{M}{b}}}\right) = \mathcal{O}\left(\log_{\frac{M}{b}} \frac{N}{b}\right)$ となる．各フェーズでの I/O 計算量は $\mathcal{O}\left(\frac{N}{b}\right)$ なので，合計の I/O 計算量は $\mathcal{O}\left(\frac{N}{b}\log_{\frac{M}{b}} \frac{N}{b}\right)$ となり，これは下界 [1] と一致する．

5.2 提案アルゴリズムの概要

$I/O(b, M)$ 上で設計された任意のアルゴリズムに対して，同じ I/O 計算量を持つ AGPU(p, b, M) 上のアルゴリズムが存在する [11]．しかし，前節のアルゴリズムを AGPU モデル上で実装する場合，多重度が 1（最小値）になってしまう．そこで，アルゴリズムを改良し，多重度を大きくすることを考える．

前節のアルゴリズムを AGPU(p, b, M, C) 上で動作させる際に多重度が最小値 1 になる原因は，ローカルピボット抽出時に行うメモリロードのソート処理である．そこでメモリロード全体に対するソート処理を行う事無しにローカルピボットの抽出処理を行うようにする．

基本的なアイデアは，ピボット集合を用いてメモリロードの領域分割処理を繰り返すことで，メモリロードをサイズ b 以下のチャンクに分割することである．

一つのメモリロードは一つのマルチプロセッサによって処理される．まず，メモリロードを b ワードからなる基本ブロックに分割し，マルチプロセッサはすべての基本ブロックに対し共有メモリを用いてソートを行う．次に，各基本ブロックにおいて，$S/4$ 要素ごとにローカルピボットを抽出する．すると合計で $4M/S = 4b \cdot S$ 個のピボットが取り出せる．ここから，S 個のグローバルピボットを取り出す（i 番目に抽出されるグローバルピボットは全ピボットの中で $4bi$ 番目に小さい要素となる）．これは，線形時間セレクション処理 [5] を S 回することで実現できる．1 回のセレクション処理での I/O 計算量は $\mathcal{O}(S)$ なので，S 個のセレクション算出処理での合計 I/O 計算量は $S \cdot S = M/b$ となる．このグローバルピボット集合を用いて，メモリロードの分割を行う（4 章と同様の方法を用いる）と，各領域のサイズは高々 $\frac{5}{4}\frac{M}{S}$ となる．上記の処理を $\log_{4S/5} \frac{M}{b} = \mathcal{O}(1)$ 回行うことで，メモリロードを $\mathcal{O}(b)$ のチャンクに分割することができる．最後に分割されたチャンクをシーケンシャルにチェックしていくことで，メモリロードから $4bS$ 個のローカルピボットを抽出する．

I/O 計算量について考察する．1 回の分割での I/O 計算量はメモリロードにつき $\mathcal{O}(M/b)$ なので，入力全体では $\mathcal{O}(N/b)$ である．よって全フェーズ合計の I/O 計算量は $\mathcal{O}\left(\frac{N}{b}\log_{\frac{M}{b}} \frac{N}{b}\right)$ となり，下界と一致する．また，アルゴリズム全体の計算量は表 2 のようになる．

6. 結論

本論文では，まず，Merrill らの高速 Radix ソートの計算量を AGPU モデルを用いて解析した後，それを変更することで 2 つのアルゴリズムを提案した．一つ目は MSD Radix ソートであり，二つ目は Splitter-based ソートである．MSD Radix ソートは Merrill らの Radix ソートと同様の計算量を持っている．Splitter-baed ソートは漸近的な計算量がキーのサイズ等に依存しないため，キーのサイズが大きいような場合にも適している．

今後は，提案アルゴリズムを実装し，実計算時間を評価したい．また，複数 GPU デバイスを備えた環境において，本アルゴリズムを用いた高速化を試みたい．また，Merrill らの高速 Radix ソート [14] についても更なる高速化を検討したい．

参考文献

[1] Aggarwal, A. and Vitter, Jeffrey, S.: The input/output complexity of sorting and related problems, *Commun. ACM*, Vol. 31, No. 9, pp. 1116–1127 (online), DOI: 10.1145/48529.48535 (1988).

[2] Capannini, G., Silvestri, F., Baraglia, R. and Nardini, F.: Sorting using bitonic network with CUDA, *Proceedings of the 7th Workshop on LSDS-IR* (2009).

[3] Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C.: *Introduction to Algorithms, Third Edition*, The MIT Press, 3rd edition (2009).

[4] Dotsenko, Y., Govindaraju, N. K., Sloan, P.-P., Boyd, C. and Manferdelli, J.: Fast scan algorithms on graphics processors, *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, New York, NY, USA, ACM, pp. 205–213 (online), DOI: 10.1145/1375527.1375559 (2008).

[5] Floyd, R.: Permuting Information in Idealized Two-Level Storage, *Complexity of Computer Computations* (Miller, R., Thatcher, J. and Bohlinger, J., eds.), The IBM Research Symposia Series, Springer US, pp. 105–109 (1972).

[6] Fortune, S. and Wyllie, J.: Parallelism in random access machines, *Proceedings of the tenth annual ACM symposium on Theory of computing*, STOC '78, New York, NY, USA, ACM, pp. 114–118 (online), DOI: 10.1145/800133.804339 (1978).

[7] Govindaraju, N., Gray, J., Kumar, R. and Manocha, D.: GPUTeraSort: high performance graphics co-

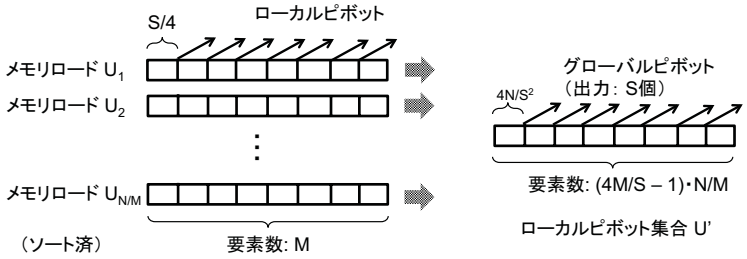


図 **6** Distribution ソートのピボット算出処理の流れ

表 **2** 提案アルゴリズム（Splitter-based ソート）の計算量

	I/O 計算量	時間計算量	多重度
(下界)	$\Omega\left(\frac{N}{b}\log_{\frac{M}{b}}\frac{N}{b}\right)$	$\Omega\left(\frac{N}{p}\log N\right)$	-
Splitter-based ソート	$\mathcal{O}\left(\frac{N}{b}\log_{\frac{M}{b}}\frac{N}{b}\right)$	$\mathcal{O}\left(\frac{N}{p}\left(\sqrt{\frac{M}{b}}+\log b\right)\log_{\frac{M}{b}}\frac{N}{b}\right)$	$\mathcal{O}\left(\sqrt{\frac{M}{b}}\right)$
従来の I/O 最適アルゴリズム [11]	$\mathcal{O}\left(\frac{N}{b}\log_{\frac{M}{b}}\frac{N}{b}\right)$	$\mathcal{O}\left(\frac{N}{p}\log\frac{N}{b}\log b\right)$	$O(1)$

processor sorting for large database management, *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, New York, NY, USA, ACM, pp. 325–336 (online), DOI: 10.1145/1142473.1142511 (2006).

[8] Greß, A. and Zachmann, G.: GPU-ABiSort: optimal parallel sorting on stream architectures, *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, Washington, DC, USA, IEEE Computer Society, pp. 45–45 (online), available from ⟨http://dl.acm.org/citation.cfm?id=1898953.1898980⟩ (2006).

[9] Harris, M.: Optimizing Parallel Reduction in CUDA (2008).

[10] Khorasani, E., Paulovicks, B. D., Sheinin, V. and Yeo, H.: Parallel implementation of external sort and join operations on a multi-core network-optimized system on a chip, *Proceedings of the 11th international conference on Algorithms and architectures for parallel processing - Volume Part I*, ICA3PP'11, Berlin, Heidelberg, Springer-Verlag, pp. 318–325 (online), available from ⟨http://dl.acm.org/citation.cfm?id=2075416.2075446⟩ (2011).

[11] Koike, A. and Sadakane, K.: A Novel Computational Model for GPUs with Application to I/O Optimal Sorting Algorithms, *2014 IEEE 28th International Parallel & Distributed Processing Symposium Workshops*, pp. 614–623 (online), DOI: 10.1109/IPDPSW.2014.72 (2014).

[12] Kolonias, V., Voyiatzis, A. G., Goulas, G. and Housos, E.: Design and implementation of an efficient integer count sort in CUDA GPUs, *Concurr. Comput. : Pract. Exper.*, Vol. 23, No. 18, pp. 2365–2381 (online), DOI: 10.1002/cpe.1776 (2011).

[13] Kothapalli, K., Mukherjee, R., Rehman, M., Patidar, S., Narayanan, P. and Srinathan, K.: A performance prediction model for the CUDA GPGPU platform, *High Performance Computing (HiPC), 2009 International Conference on*, pp. 463 –472 (online), DOI: 10.1109/HIPC.2009.5433179 (2009).

[14] Merrill, D. and Grimshaw, A.: High Performance and Scalable Radix Sorting: A case study of implementing dynamic parallelism for GPU computing, *Parallel Processing Letters*, Vol. 21, No. 02, pp. 245–272 (online), DOI: 10.1142/S0129626411000187 (2011).

[15] NVIDIA Corporation: NVIDIA CUDA C Programming Guide version 4.2 (2012).

[16] Peters, H., Schulz-Hildebrandt, O. and Luttenberger, N.: Fast in-place sorting with CUDA based on bitonic sort, *Proceedings of the 8th international conference on Parallel processing and applied mathematics: Part I*, PPAM'09, Berlin, Heidelberg, Springer-Verlag, pp. 403–410 (online), available from ⟨http://dl.acm.org/citation.cfm?id=1882792.1882841⟩ (2010).

[17] Peters, H., Schulz-Hildebrandt, O. and Luttenberger, N.: A Novel Sorting Algorithm for Many-core Architectures Based on Adaptive Bitonic Sort, *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, Washington, DC, USA, IEEE Computer Society, pp. 227–237 (online), DOI: 10.1109/IPDPS.2012.30 (2012).

[18] Satish, N., Harris, M. and Garland, M.: Designing efficient sorting algorithms for manycore GPUs, *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, Washington, DC, USA, IEEE Computer Society, pp. 1–10 (online), DOI: 10.1109/IPDPS.2009.5161005 (2009).

[19] Satish, N., Kim, C., Chhugani, J., Nguyen, A. D., Lee, V. W., Kim, D. and Dubey, P.: Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort, *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, New York, NY, USA, ACM, pp. 351–362 (online), DOI: 10.1145/1807167.1807207 (2010).

[20] Sengupta, S., Harris, M. and Garland, M.: Efficient parallel scan algorithms for GPUs, *Technical Report NVR-2008-003*, NVIDIA (2008).

[21] Ye, X., Fan, D., Lin, W., Yuan, N. and Ienne, P.: High performance comparison-based sorting algorithm on many-core GPUs, *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–10 (online), DOI: 10.1109/IPDPS.2010.5470445 (2010).

[22] 小池 敦, 定兼 邦彦: AGPU モデルにおけるマルチスレッディングの効果, 総合大会 COMP 学生シンポジウム DS-1-13, 電子情報通信学会 (2013).

ハードウェアソーティング アルゴリズムのFPGA実装

広島大学　松本直之

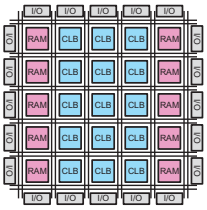
FPGAとは

- FPGA：Field Programmable Gate Array
  - ユーザが任意に回路を書き換えることのできるLSI

CLB (Configurable Logic Block)
ユーザが書き換え可能な論理ブロック
小規模な組み合わせ回路や順序回路を構成

ブロックRAM
CLBのメモリ能力を補う専用回路

➢ これらの素子を接続することで任意の回路を実現



FPGAの構成図

研究概要

- 2種類のソーティングアルゴリズムをFPGAに実装

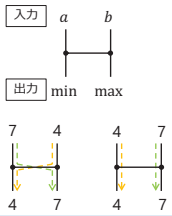
□バイトニックソート
- ソーティングネットワークに基づいた回路
- 任意のデータ幅のデータをソート可能

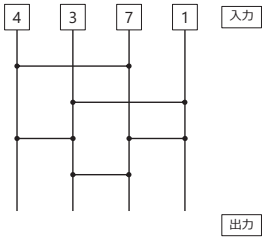
□マージソート
- FIFOを用いた回路
- 最大で524288要素のデータをソート可能

ソーティングネットワークとは

- 入力データをソートして出力するネットワーク
- 複数の**コンパレータ**から構成

コンパレータ
2つの値を入力し、小さい値を一方に、大きい値を他方に出力（比較交換）

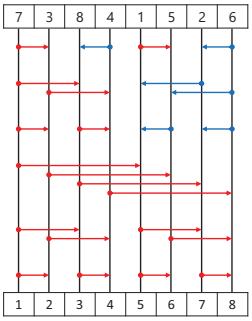




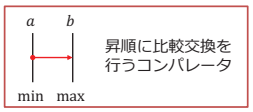
ソーティングネットワークの例

バイトニックソートのソーティングネットワーク

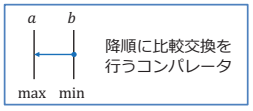
- 要素数8の場合



K.E.Batcher. Sorting networks and their applications. *AFIPS Spring Joint Computer Conference*, pp307-314, 1968

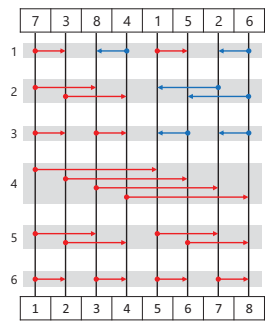


昇順に比較交換を行うコンパレータ



降順に比較交換を行うコンパレータ

ソーティングネットワークの段数



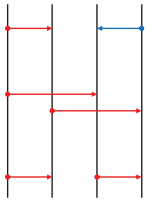
要素数nのバイトニックソートのソーティングネットワークの段数は

$$\frac{\log n\,(\log n+1)}{2}=O(\log^2 n)$$

要素数8の場合、段数は6

バイトニックソート回路

データ列2	01100	11010	10101	00011
データ列1	111	011	001	100



- ビットシリアルを用いた回路を設計

ビットシリアル
データを上位ビットから1ビットずつ入力

最上位ビットから順にソーティングネットワークに入力する

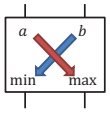
ソートされたデータが最上位ビットから順に出力される

- 1ビットずつ入力するため、任意のデータ幅のソートを行うことができる

ビットシリアルを用いたコンパレータの動作

- 最上位ビットから順に入力し、比較を行う
  - 異なるビットが入力されると、その時点で大小が確定する

101010　100111



入力された2つのビットを比較

↓

同じビットであれば、交換せず出力

↓

異なるビットであれば、1をmaxに、0をminに出力
この時点で2つのデータの大小が確定

↓

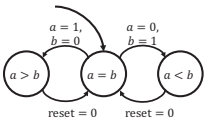
以降のビットは比較せず、決められた通りに出力

ビットシリアルを用いたコンパレータの動作

- コンパレータの動作を3つに分ける
  - ステートマシンを用いて制御
    - ✓初期状態は$a = b$
    - ✓異なるビットが入力されると状態を遷移

各状態でのコンパレータの動作

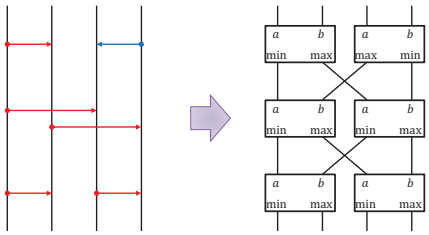
状態$a = b$	状態$a > b$	状態$a < b$
• 比較交換を行う	• 交換を行う	• 交換を行わない
		
a b 比較交換 min max	a b min max	a b min max



コンパレータのステートマシン

バイトニックソート回路の構成

- ビットシリアルを用いたコンパレータでソーティングネットワークを構成



実装結果

- ターゲットFPGA：Xilinx Virtex-7 XC7VX485T

- ソート可能な最大要素数：**1024**
- 動作周波数：**1030.822**[MHz]
- 処理時間(32ビット)：**84.40** [ns]

リソース	使用数
Slice Registers	140863 (23%)
Slice LUTs	141570 (46%)

CPUとの比較

- CPU：Intel Xeon X7460 2.66GHz
- C言語の標準関数qsort()と比較
- 要素数：1024
- データ幅：32ビット

パイプライン化により、実行回数が多くなると高速化率が上がる

実行回数	処理時間[μs] CPU	処理時間[μs] FPGA	高速化率
1	97	0.084	1154.762
10	966	0.364	2653.846
100	9657	3.158	3057.948
1000	96574	31.097	3105.573
10000	965743	310.485	3110.434

研究概要

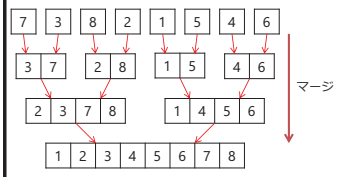
- 2種類のソーティングアルゴリズムをFPGAに実装

□バイトニックソート
- ソーティングネットワークに基づいた回路
- 任意のデータ幅のデータをソート可能

□マージソート
- FIFOを用いた回路
- 最大で524288要素のデータをソート可能

マージソート

- 2つのソート済みのデータ列をマージして、1つのソート済みデータ列を作る動作を繰り返しソートを行う



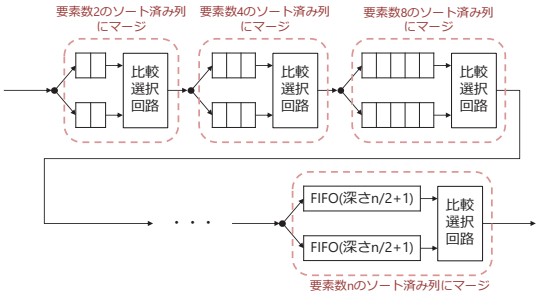
マージ操作

- 2つのソート済みデータ列の先頭要素を比較し小さい方から順に取り出す

マージソート回路

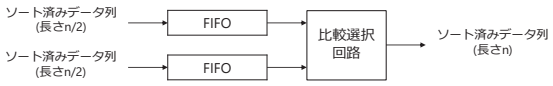
S.Todd. Algorithm and hardware for a merge sort using multiple processors. *IBM Journal of Research and Development*, pp509-517, 1978

- 小さい要素数からマージを行い、大きなソート済み列を作る
- データはパイプライン的に処理される



マージの実装

- 2つのFIFOと比較選択回路から構成
  - ✓FIFO(First-In First-Out)
    - 先に入れたデータが先に取り出されるデータ構造
  - ✓比較選択回路
    - 2つの入力のうち小さい方を出力

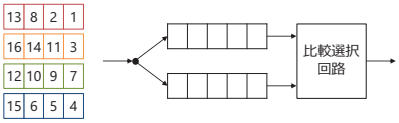


FIFO内の先頭のデータが比較選択回路に入力される

小さい方のデータをFIFOから取り出す

マージの動作

- n/2要素のソート済みデータを逐次的に入力する
- n要素のソート済みデータが逐次的に出力される



実装結果

- ターゲットFPGA：
  Xilinx Virtex-7 XC7VX485T
- データ幅：32ビット

- ソート可能な最大要素数：**524288**
- 動作周波数：**278.339**[MHz]
- 処理時間(524288要素)：**3.767**[ms]

512要素以上のマージ回路でブロックRAMを使用することで、動作周波数の低下を防ぎ、リソースを節約

要素数	Slice Register 使用数	%	Slice LUT 使用数	%	Block RAM 使用数	%	動作周波数 [MHz]
2	136	0	351	0	0	0	414.834
4	220	0	679	0	0	0	370.739
8	325	0	1040	0	0	0	372.731
16	423	0	1471	0	0	0	366.447
32	524	0	1391	0	0	0	326.813
64	643	0	1769	0	0	0	326.179
128	770	0	2249	0	0	0	324.160
256	908	0	2976	0	0	0	314.891
512	1122	0	3450	1	2	0	312.710
1K	1345	0	4006	1	4	0	300.801
2K	1577	0	4146	1	6	0	298.485
4K	1818	0	4610	1	10	0	270.464
8K	2068	0	5110	1	18	1	288.806
16K	2327	0	5765	1	34	3	284.823
32K	2595	0	6412	2	66	6	285.792
64K	2872	0	7101	2	130	12	280.006
128K	3158	0	7648	2	258	25	269.535
256K	3455	0	8412	2	514	49	269.906
512K	3763	0	8729	2	1026	99	278.339

性能比較

- CPU：Intel Xeon X7460 2.66GHz
- C言語の標準関数qsort()と比較
- データ幅：32ビット

524288要素のソートのとき、約25倍の高速化

要素数	処理時間 CPU	FPGA	高速化率
2	109.321[ns]	9.642[ns]	11.338
4	197.351[ns]	24.276[ns]	8.129
8	398.072[ns]	48.292[ns]	8.243
16	868.191[ns]	95.512[ns]	9.090
32	1.961[µs]	0.208[µs]	9.428
64	4.361[µs]	0.408[µs]	10.689
128	9.692[µs]	0.808[µs]	11.995
256	21.148[µs]	1.648[µs]	12.833
512	45.958[µs]	3.300[µs]	13.927
1K	99.786[µs]	6.838[µs]	14.593
2K	215.658[µs]	13.756[µs]	15.677
4K	464.747[µs]	30.329[µs]	15.324
8K	993.881[µs]	56.772[µs]	17.507
16K	2116.377[µs]	115.093[µs]	18.388
32K	4502.533[µs]	229.363[µs]	19.631
64K	9553.16[µs]	468.158[µs]	20.406
128K	20157.653[µs]	972.638[µs]	20.725
256K	43593.288[µs]	1942.547[µs]	22.441
512K	94205.103[µs]	3767.327[µs]	25.006

まとめ

- 本研究ではバイトニックソートとマージソートを実行する回路をFPGAに実装した
- バイトニックソート回路では、ビットシリアルを用いることで任意のデータ幅のデータを最大1024要素ソート可能となった
- マージソート回路では、ブロックRAMを用いることで32ビットのデータで最大524288要素をソート可能となった

生化学反応計算における基本演算およびソートの実現

(Reaction systems for logical operations and sorting)

Akifumi Nakanishi Akihiro Fujiwara
Graduate School of Computer Science and Systems Engineering, Kyushu Institute of Technology
Iizuka, Fukuoka, 820-8502, Japan
Email: o676122a@mail.kyutech.jp, fujiwara@cse.kyutech.ac.jp

***Abstract*—In the present paper, we consider the reaction system, which is a computational model based on biochemical reactions in living cells, and propose reaction systems for logical operations and sorting. We first propose a reaction system that executes a two-input logical operations, such as AND, OR, and XOR, and show that the reaction system works in $O(1)$ steps. We next propose a reaction system for a compare-and-swap operation of two binary numbers of m bits. We show that the reaction system works in $O(m)$ parallel steps using $O(m)$ types of objects and reaction rules. We finally propose a reaction system for sorting of n binary numbers of m bits. The reaction system is based on an idea of the odd-even sort, and we show that the reaction system works in $O(mn)$ parallel steps and using $O(mn)$ types of objects and reaction rules.**

I. Introduction

A number of next-generation computing paradigms have been considered due to limitation of silicon based computation. As an example of the computing paradigms, natural computing, which works using natural materials for computation, has considerable attention. A membrane computing [1], which is a computational model inspired by the structures and behaviors of living cells, is a representative of the natural computing. A computational model of the membrane computing is called P system, and a number of P systems [2], [3], [4], [5], [6], [7] have been proposed for solving NP problems. In addition, a number of P systems [2], [8] are also proposed for basic operations such as logical and arithmetic operations.

On the other hand, a reaction system [9], [10], [11], which is called R system, has been proposed as another computational model of natural computing. The R system is based on biochemical reactions in living cells, and the fundamental idea of the R system is based on interaction between biochemical reactions, which are the mechanisms of facilitation and inhibition. For the reaction system, a number of primitive operations are considered in [9], [10], [11], and no R system that executes basic operations, such as logic or arithmetic operations, has been proposed. However, the reaction system for the basic operation is needed to apply the reaction system on a wide range of problems.

In the present paper, we propose R systems for logical operations and sorting of binary numbers. We first propose a reaction system that executes a two-input logical operation, such as AND, OR, and XOR. We show that the R system works in $O(1)$ parallel steps and using $O(1)$ types of objects and reaction rules.

We next propose an R system for a compare-and-swap operation of two binary numbers of m bits. The R system first computes the most significant bit between the two input values, and then, swap operation is executed according to the result of the most significant bit. We show that the R system works in $O(m)$ parallel steps and using $O(m)$ types of objects and reaction rules.

We finally propose an R system for sorting of n binary numbers of m bits. The R system is based on an idea of the odd-even sort, and the R system employs an object that works as a counter, and executes the sorting for odd and even steps using the counter. We show that the R system works in $O(mn)$ parallel steps and using $O(mn)$ types of objects and reaction rules.

II. Preliminaries

A. Reaction system

A reaction system [9], [10], [11] is a computational model based on biochemical reactions in living cells. In this paper, we first explain definition of a reaction on the reaction system, which is based on [11].

A reaction a is defined by the following equation.

$$a = (R_a, I_a, P_a)$$

R_a, I_a and P_a are sets of reactant, inhibitor and product, respectively, and all of the three sets are finite nonempty sets such that $R_a \cap I_a = \emptyset$, $M = R_a \cup I_a$, and $|M| \geq 2$.

The reaction a is applied if $R_a \subseteq T$ and $I_a \cap T = \emptyset$ for a finite set T. The result of a on T is denoted by $Res_a(T)$, and $Res_a(T) = P_a$ in case that reaction a is applied, and otherwise, $Res_a(T) = \emptyset$.

I now show an example of the reaction. Let $a = (\{3\}, \{1, 2\}, \{1, 2, 4\})$ and $T_1 = \{3, 4\}, T_2 = \{2, 3, 4\}$. In this case, $Res_a(T_1)$ and $Res_a(T_2)$ are sets given below.

$$\begin{aligned} Res_a(T_1) &= P_a = \{1, 2, 4\} \\ Res_a(T_2) &= \emptyset \end{aligned}$$

As shown in the above example, all non-reacted objects, which are not included in P_a, are disappeared after application

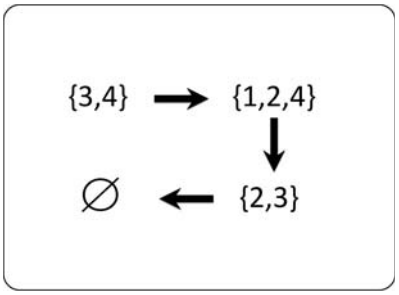


Fig. 1. Operation of R system Λ

of reaction a. The property is called non-persistency of the object.

Next, we explain definition of the R system. R system Λ is defined by the following equation.

$$\Lambda = (S, A)$$

In the above equation, S is a set of all objects, and A is a set of reactions. In addition, the result of a set of reactions A for T is an union of obtained results of all reactions in A. In other words, $Res_A(T)$, which is a set of result of A for a finite set T, is defined as follows.

$$Res_A(T) = \bigcup_{a \in A} res_a(T)$$

In the R system, an application of all reaction is called a transition. In this paper, we assume that all applicable reactions are applied simultaneously in a transition, and also assume that each transition is called one parallel step. The complexity of R system is defined the number of parallel steps executed in the computation.

For example, we show a simple R system Λ, which is defined as follows.

$$\begin{aligned} S &= \{1,2,3,4\} \\ A &= \{a,b,c\} \\ a &= (\{3\},\{1,2\},\{1,2\}) \\ b &= (\{3,4\},\{2\},\{4\}) \\ c &= (\{1,4\},\{3\},\{2,3\}) \end{aligned}$$

Figure1 shows an execution of the R system in case that a set $\{3,4\}$ is given as input. In this case, applicable reactions are a and b for the input set, and these two rules are applied simultaneously. Since a result of the transition is an union of obtained objects, an obtained set of objects is $\{1,2,4\}$ after the first transition.

Next, an applicable reaction for $\{1,2,4\}$ is only c, and an obtained set of objects is $\{2,3\}$ after the second transition. Then, a set of objects becomes empty after the third transition since no reaction is applicable for $\{2,3\}$.

III. Data structure for binary numbers

In this subsection, we describe a unified data structure for a binary number using objects in the R system. Data structure for Boolean values has been proposed in [2], and we improve the data structure in this paper. In the data structure, one object corresponds to one bit of a binary number. Therefore, we use $O(mn)$ objects to denote n binary numbers of m bits. In addition, the data structure enables the addressing feature, that is, each binary number is stored in a given address.

Let $V_{i,j} \in \{0,1\}$ be a j-th Boolean value stored in address i. Then, the value is denoted using the following object on the R system.

$$\langle A_i, B_j, V_{i,j} \rangle$$

We call the above object *a memory object* for Boolean values.

In case of a binary number, let $V_{i,m-1}, V_{i,m-2}, \cdots, V_{i,0}$ be m Boolean values stored in address i. Then, a non-negative integer V_i stored in address i satisfies the following condition.

$$V_i = \sum_{j=1}^{m} V_{i,j} \times 2^{j-1}$$

The above binary number is represented by m memory objects given below.

$$\langle A_i, B_{m-1}, V_{i,m-1} \rangle, \langle A_i, B_{m-2}, V_{i,m-2} \rangle, \cdots, \langle A_i, B_0, V_{i,0} \rangle$$

For example, the binary number 1010 stored in the address 6 is represented by the following four objects.

$$\langle A_6, B_3, 1 \rangle, \langle A_6, B_2, 0 \rangle, \langle A_6, B_1, 1 \rangle, \langle A_6, B_0, 0 \rangle$$

IV. R systems for logical operations

A. Input and output

In this section, we propose a simple R system that executes a two-input logical operation, such as AND, OR, and XOR. We assume that input of the logical operation is a pair of Boolean values x, y. The x, y are denoted by a pair of following two memory objects on the R system.

$$\langle A_0, B_0, V_{0,0} \rangle, \quad \langle A_1, B_0, V_{1,0} \rangle$$

We also assume that an output of the logical operation is a Boolean value z, which is denoted by the following memory object.

$$\langle A_2, B_0, V_{2,0} \rangle$$

B. R system for logical operations

In this subsection, we show an R system for the logical operation. Any two-input logical operation is defined in the truth table in Table I. For example, $z_0 = z_1 = z_2 = 0$ and $z_3 = 1$ in case of AND operation. We propose an R system for any two-input logical operation based on the truth table.

C. Details of the R system for the logical operation

An output of the logical operation is determined according to the truth table. Thus, we encode the truth table into the R system as reactions. We now formally define the R system Λ_{LO} for any logical operations in the following.

$$\Lambda_{LO} = (S, A)$$

TABLE I
A TRUTH TABLE OF A TWO-INPUT LOGICAL OPERATION

Input x	Input y	Output z
0	0	z_0
0	1	z_1
1	0	z_2
1	1	z_3

$$
\begin{aligned}
S &= \{\langle A_0, B_0, V_{0,0}\rangle, \langle A_1, B_0, V_{1,0}\rangle, \langle A_2, B_0, V_{2,0}\rangle, \\
&\quad \langle E\rangle | V_{0,0}, V_{1,0}, V_{2,0} \in \{0,1\}\} \\
A &= \{a_1, a_2, a_3, a_4\} \\
a_1 &= (\{\langle A_0, B_0, 0\rangle, \langle A_1, B_0, 0\rangle\}, \{\langle E\rangle\}, \{\langle A_2, B_0, z_0\rangle\}) \\
a_2 &= (\{\langle A_0, B_0, 0\rangle, \langle A_1, B_0, 1\rangle\}, \{\langle E\rangle\}, \{\langle A_2, B_0, z_1\rangle\}) \\
a_3 &= (\{\langle A_0, B_0, 1\rangle, \langle A_1, B_0, 0\rangle\}, \{\langle E\rangle\}, \{\langle A_2, B_0, z_2\rangle\}) \\
a_4 &= (\{\langle A_0, B_0, 1\rangle, \langle A_1, B_0, 1\rangle\}, \{\langle E\rangle\}, \{\langle A_2, B_0, z_3\rangle\})
\end{aligned}
$$

In the above R system, $\langle E\rangle$ is an object that denotes an empty set, and z_0, z_1, z_2, z_3 are Boolean values given in Table I.

For a simple example of the above R system, we assume that two input objects $\langle A_0, B_0, 0\rangle$ and $\langle A_1, B_0, 1\rangle$ are given to the R system. Then, a reaction a_2 is applied, and an object $\langle A_2, B_0, z_1\rangle$ is obtained as an output of the R system.

Since computation of the above R system Λ_{OL} is completed by only one transaction, we obtain the following theorem for Λ_{OL}.

Theorem 1: The R system Λ_{OL}, which computes any two-input logical operations, works in $O(1)$ parallel steps using $O(1)$ types of objects and reactions. □

V. COMPARE-AND-SWAP

In this section, we present an R system for the compare-and-swap operation of two binary numbers of m bits. The compare-and-swap operation compares two input values x_{in} and y_{in}, and assigns the smaller and larger values to the variable x_{out} and y_{out}, respectively.

A. Input and output

Two input binary numbers are denoted by the following sets of memory objects on the R system.

$$
\begin{aligned}
x_{in}: &\quad \{\langle A_0, B_j, V_{0,j}\rangle \mid 0 \le j \le m-1\} \\
y_{in}: &\quad \{\langle A_1, B_j, V_{1,j}\rangle \mid 0 \le j \le m-1\}
\end{aligned}
$$

In addition to the above input, we assume that the following object is given to the R system. The object, which denotes bit-position of the comparison, starts the R system.

$$\langle C, m-1\rangle$$

In addition, two output binary numbers are also denoted by the following sets of memory objects.

$$
\begin{aligned}
x_{out}: &\quad \{\langle A_2, B_j, V_{2,j}\rangle \mid 0 \le j \le m-1\} \\
y_{out}: &\quad \{\langle A_3, B_j, V_{3,j}\rangle \mid 0 \le j \le m-1\}
\end{aligned}
$$

B. An overview of the R system

Let $x_{in,j}$ and $y_{in,j}$ be j-th bit of two input binary numbers. The R system for the compare-and-swap consists of the following two steps. First, in Step 1, each bit of the two input binary numbers, x_{in} and y_{in}, are compared from a higher bit to a lower bit. We assume that $x_{in,k}$ and $y_{in,k}$ are the first pair of bits such that the bits are different in the comparison. Then, x_{in} is greater than y_{in} if $x_{in,k} = 1, y_{in,k} = 0$, otherwise, x_{in} is less than y_{in}, and $x_{in,k} = 0, y_{in,k} = 1$. An object that denotes a result of the comparison is created after the comparison.

Second, in Step 2, two binary numbers are exchanged and outputted to x_{out} and y_{out} in case of $x_{in} < y_{in}$. Otherwise, x_{in} and y_{in} are copied into x_{out} and y_{out}, respectively.

In the above two steps, it is worth while noticing that memory objects must be copied repeatedly because of non-persistency of the object.

C. Details of the R system

We now explain each step of the R system. In Step 1, each bit of the two input binary numbers, x_{in} and y_{in}, are compared from a higher bit to a lower bit. The comparison is executed using the following sets of reactions.

$$
\begin{aligned}
A_{1,1} = \{ &(\{\langle A_0, B_k, 0\rangle, \langle A_1, B_k, 1\rangle, \langle C, k\rangle\}, \{\langle E\rangle\}, \\
&\{\langle A_0, B_k, 0\rangle, \langle A_1, B_k, 1\rangle, \langle LT\rangle\}) \\
&\mid 0 \le k \le m-1\} \\
\cup \{ &(\{\langle A_0, B_k, 1\rangle, \langle A_1, B_k, 0\rangle, \langle C, k\rangle\}, \{\langle E\rangle\}, \\
&\{\langle A_0, B_k, 1\rangle, \langle A_1, B_k, 0\rangle, \langle GT\rangle\}) \\
&\mid 0 \le k \le m-1\} \\
A_{1,2} = \{ &(\{\langle A_0, B_k, 0\rangle, \langle A_1, B_k, 0\rangle, \langle C, k\rangle\}, \{\langle E\rangle\}, \\
&\{\langle A_0, B_k, 0\rangle, \langle A_1, B_k, 0\rangle, \langle C, k-1\rangle\}) \\
&\mid 1 \le k \le m-1\} \\
\cup \{ &(\{\langle A_0, B_k, 1\rangle, \langle A_1, B_k, 1\rangle, \langle C, k\rangle\}, \{\langle E\rangle\}, \\
&\{\langle A_0, B_k, 1\rangle, \langle A_1, B_k, 1\rangle, \langle C, k-1\rangle\}) \\
&\mid 1 \le k \le m-1\}
\end{aligned}
$$

In case of j-th bit of two binary numbers are different, reactions in $A_{1,1}$ are applied, and one of objects, $\langle LT\rangle$ or $\langle GT\rangle$, which denotes $x_{in} < y_{in}$ or $x_{in} > y_{in}$, is created. Otherwise, two bits are copied, and the comparison is moved to the next bit-position using reactions in $A_{1,2}$.

In addition to the above, the other memory objects are copied repeatedly, due to non-persistency of the object, using the following sets of reactions.

$$
\begin{aligned}
A_{1,3} = \{ &(\{\langle A_0, B_k, V_{0,k}\rangle\}, \{\langle C, k\rangle, \langle LT\rangle, \langle GT\rangle, \langle EQ\rangle\}, \\
&\{\langle A_0, B_k, V_{0,k}\rangle\}) \mid 0 \le k \le m-1, V_{0,k} \in \{0,1\}\} \\
\cup \{ &(\{\langle A_1, B_k, V_{1,k}\rangle\}, \{\langle C, k\rangle, \langle LT\rangle, \langle GT\rangle, \langle EQ\rangle\}, \\
&\{\langle A_1, B_k, V_{1,k}\rangle\}) \mid 0 \le k \le m-1, V_{1,k} \in \{0,1\}\}
\end{aligned}
$$

In case of $x_{in} = y_{in}$, reactions in the following $A_{1,4}$ is applied after comparisons of all bits, and an object $\langle EQ\rangle$ is

created.

$$\begin{aligned} A_{1,4} &= \{\ (\{\langle A_0, B_0, 0\rangle, \langle A_1, B_0, 0\rangle, \langle C, 0\rangle\}, \{\langle E\rangle\}, \\ &\quad \{\langle A_0, B_0, 0\rangle, \langle A_1, B_0, 0\rangle, \langle EQ\rangle\})\ \} \\ &\quad \cup \{\ (\{\langle A_0, B_0, 1\rangle, \langle A_1, B_0, 1\rangle, \langle C, 0\rangle\}, \{\langle E\rangle\}, \\ &\quad \{\langle A_0, B_0, 1\rangle, \langle A_1, B_0, 1\rangle, \langle EQ\rangle\})\ \} \end{aligned}$$

Next, in Step 2, two input binary numbers, x_{in} and y_{in}, are exchanged and outputted to x_{out} and y_{out} in case that there exists an object $\langle GT\rangle$. The swap and copy are executed using the following set of reactions $A_{2,1}$.

$$\begin{aligned} A_{2,1} &= \{\ (\{\langle A_0, B_k, V_{0,k}\rangle, \langle GT\rangle\}, \{\langle E\rangle\}, \\ &\quad \{\langle A_3, B_k, V_{0,k}\rangle\}) \mid 0 \le k \le m-1, V_{0,k} \in \{0,1\}\} \\ &\quad \cup \{\ (\{\langle A_1, B_k, V_{1,k}\rangle, \langle GT\rangle\}, \{\langle E\rangle\}, \\ &\quad \{\langle A_2, B_k, V_{1,k}\rangle\}) \mid 0 \le k \le m-1, V_{1,k} \in \{0,1\}\} \end{aligned}$$

On the other hand, two input binary numbers, x_{in} and y_{in}, are copied to x_{out} and y_{out}, respectively, in case that there exists $\langle LT\rangle$ or $\langle EQ\rangle$. The copy is executed using the following sets of reactions, $A_{2,2}$ and $A_{2,3}$.

$$\begin{aligned} A_{2,2} &= \{\ (\{\langle A_0, B_k, V_{0,k}\rangle, \langle LT\rangle\}, \{\langle E\rangle\}, \\ &\quad \{\langle A_2, B_k, V_{0,k}\rangle\}) \mid 0 \le k \le m-1, V_{0,k} \in \{0,1\}\} \\ &\quad \cup \{\ (\{\langle A_1, B_k, V_{1,k}\rangle, \langle LT\rangle\}, \{\langle E\rangle\}, \\ &\quad \{\langle A_3, B_k, V_{1,k}\rangle\}) \mid 0 \le k \le m-1, V_{1,k} \in \{0,1\}\} \\ A_{2,3} &= \{\ (\{\langle A_0, B_k, V_{0,k}\rangle, \langle EQ\rangle\}, \{\langle E\rangle\}, \\ &\quad \{\langle A_2, B_k, V_{0,k}\rangle\}) \mid 0 \le k \le m-1, V_{0,k} \in \{0,1\}\} \\ &\quad \cup \{\ (\{\langle A_1, B_k, V_{1,k}\rangle, \langle EQ\rangle\}, \{\langle E\rangle\}, \\ &\quad \{\langle A_3, B_k, V_{1,k}\rangle\}) \mid 0 \le k \le m-1, V_{1,k} \in \{0,1\}\} \end{aligned}$$

We now summarize details of the R system Λ_{CS}, which executes the compare-and-swap operation.

$$\Lambda_{CS} = (S, A)$$

$$\begin{aligned} S &= \{\langle A_i, B_j, V_{0,j}\rangle \mid 0 \le i \le 3, 0 \le j \le m-1\} \\ &\quad \cup \{\langle C, k\rangle \mid 0 \le k \le m-1\} \\ &\quad \cup \{\langle GT\rangle, \langle LT\rangle, \langle EQ\rangle, \langle E\rangle\} \\ A &= A_{1,1} \cup A_{1,2} \cup A_{1,3} \cup A_{1,4} \cup A_{2,1} \cup A_{2,2} \cup A_{2,3} \end{aligned}$$

Figure 2 illustrates an execution of the R system Λ_{CS}. In the example, $m = 3$, and an input is a pair of two binary numbers, $x_{in} = 1100$ and $y_{in} = 1011$. In the example, at first, reactions in $A_{1,2}$ and $A_{1,3}$ are applied because the highest bits of the two numbers are same. Next, reactions in $A_{1,1}$ and $A_{1,3}$ are applied because the next bits of the two numbers are different, and an object $\langle GT\rangle$ is created according to the comparison. Finally, the output values are set to x_{out} and y_{out} using reactions in $A_{2,1}$.

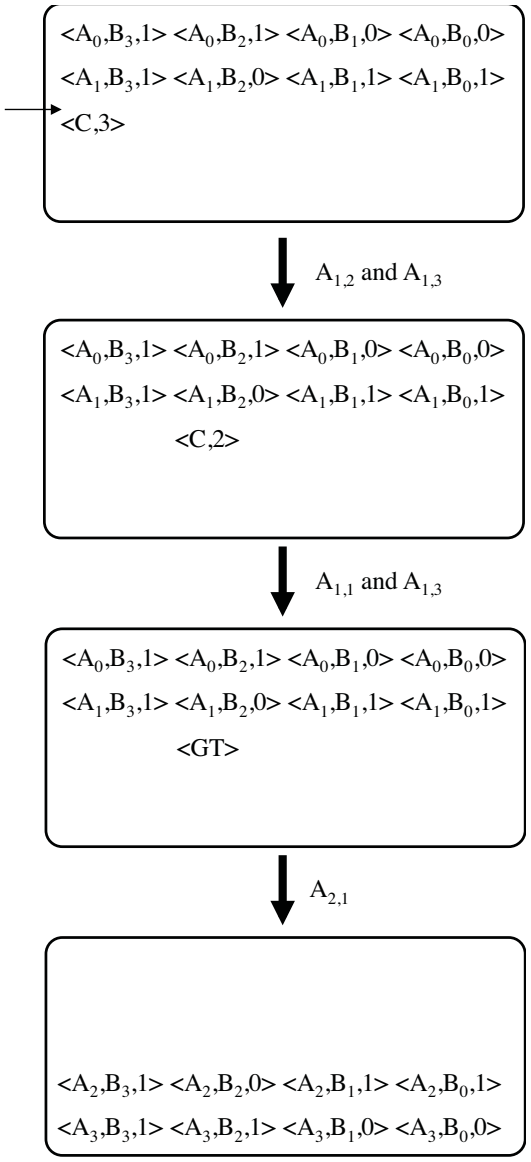


Fig. 2. An example of execution of Λ_{CS}

D. Complexity of the R system

Since complexity of Step 1 in the above R system Λ_{CS} is $O(m)$, we obtain the following theorem for Λ_{CS}.

Theorem 2: The R system Λ_{CS}, which executes the compare-and-swap operation for two binary numbers of m bits, works in $O(m)$ parallel steps using $O(m)$ types of objects and reactions. □

VI. Sorting

A. Input and output

In this section, we present an R system for sorting of n binary numbers of m bits using the R system. An input and an output of the R system is a set of binary numbers that are denoted by the following set of memory objects.

$$\{\langle A_i, B_j, V_{i,j}\rangle \mid 0 \le i \le n-1, 0 \le j \le m-1\}$$

B. An overview and complexity of the R system

The proposed P system is based on odd-even transposition sort [12], which is a well-known parallel sorting algorithm. Let

$(x_0, x_1, \ldots, x_{n-1})$ be an input of the sorting. A basic idea of the odd-even transposition sort is quite simple. At odd phases, we perform the compare-and-swap operations for each pair (x_{2i}, x_{2i+1}) $(0 \leq i \leq \frac{n}{2} - 1)$ in parallel. On the other hand, we perform the same operations for each pair (x_{2i-1}, x_{2i}) $(1 \leq i \leq \frac{n}{2} - 1)$ at even phases. It is proved that the input is sorted after $\frac{n}{2}$ repetition of the above two steps [12].

Using the R system described in Section V, we can realize the odd-even transposition sort on the R system as follows.

A basic idea of R system for sorting

Repeat the following steps $\frac{n}{2}$ times.

Step 1: Execute the compare-and-swap operations, which is described in Section V, for pairs (x_{2i}, x_{2i+1}) $(0 \leq i \leq \frac{n}{2} - 1)$ in parallel.

Step 2: Execute the same compare-and-swap operations for pairs (x_{2i-1}, x_{2i}) $(1 \leq i \leq \frac{n}{2} - 1)$ in parallel.

Since all reactions on the R system can be applied in parallel, the above idea is implemented as an R system Λ_{SORT}, which sorts n binary number of m bits, with a modification of the R system proposed in Section V. Although the main difficulty for the implementation is synchronization of the compare-and-swap operations, we can solve the difficulty by using object that works as a global counter. (The precise description of the R system is omitted because an implementation of reactions is redundant.)

We now consider complexity of the R system Λ_{SORT}. The above two steps are repeated by $\frac{n}{2}$ times, and each compare-and-swap operation is executed in $O(m)$ steps. Then, we obtain the following theorem for the R system Λ_{SORT}.

Theorem 3: The R system Λ_{SORT}, which sorts n binary numbers of m bits, works in $O(mn)$ parallel steps using $O(mn)$ types of objects and reactions. □

VII. Conclusions

In the present paper, we proposed R systems for logical operations and sorting of binary numbers. We first proposed an R system that executes any two-input logical operation, and showed that the R system works in $O(1)$ parallel steps and using $O(1)$ types of objects and reaction rules. We next proposed an R system for a compare-and-swap operation of two binary numbers of m bits, and showed that the R system works in $O(m)$ parallel steps and using $O(m)$ types of objects and reaction rules. We finally proposed an R system for sorting of n binary numbers of m bits, and also showed that the R system works in $O(mn)$ parallel steps and using $O(mn)$ types of objects and reaction rules.

As future work, we are considering reduction of the numbers of types of objects and reactions in the R systems.

Acknowledgments

This research was partially supported by JSPS KAKENHI, Grand-in-Aid for Scientific Research (C), 24500019.

References

[1] G. Păun, "Computing with membranes," *Journal of Computer and System Sciences*, vol. 61, no. 1, pp. 108–143, 2000.

[2] A. Leporati and C. Zandron, "P systems with input in binary form," *International Journal of Foundations of Computer Science*, vol. 17, no. 1, pp. 127–146, 2006.

[3] L. Pan and A. Alhazov, "Solving HPP and SAT by P systems with active membranes and separationrules," *Acta Informatica*, vol. 43, no. 2, pp. 131–145, 2006.

[4] G. Păun, "P system with active membranes: Attacking NP-complete problems," *Journal of Automata, Languages and Combinatorics*, vol. 6, no. 1, pp. 75–95, 2001.

[5] C. Zandron, G. Rozenberg, and G. Mauri, "Solving NP-complete problems using P systems with active membranes," *Proceedings of the Second International Conference on Uncoventional Models of Computation*, pp. 289–301, 2000.

[6] H. Tagawa and A. Fujiwara, "Solving SAT and Hamiltonian cycle problem using asynchronous p systems," *IEICE Transactions on Information and Systems (Special section on Foundations of Computer Science)*, vol. E95-D, no. 3, 2012.

[7] K. Tanaka and A. Fujiwara, "Asynchronous p systems for hard graph problems," *International Journal of Networking and Computing*, vol. 4, no. 1, pp. 2–22, 2014.

[8] A. Fujiwara and T. Tateishi, "Logic and arithmetic operations with a constant number of steps in membrane computing," *International Journal of Foundations of Computer Science*, vol. 22, no. 3, pp. 547–564, 2011.

[9] G. Păun, "Bridging P and R," *Research Topics in Membrane Computing*, vol. 1, no. 1, pp. 53–57, 2012.

[10] ——, "Towards fypercomputations (in membrane computing)," *Lecture Notes in Computer Science*, vol. 7300, pp. 207–220, 2012.

[11] M. M. R. Brijder, A. Ehrenfeucht, "A tour of reaction systems," *International Journal of Foundations of Computer Science*, vol. 22, no. 1, pp. 1499–1518, 2011.

[12] N. Haberman, "Parallel neighbor-sort (or the glory of the induction principle)," AD-759 248, National Technical Information Service, US Department of Commerce, Tech. Rep., 1972.

酵素を用いた数値膜計算における基本演算およびソートの実現

(Enzymatic Numerical P Systems for basic operations and sorting)

Shohei Maeda and Akihiro Fujiwara
Graduate School of Computer Science and Systems Engineering
Kyushu Institute of Technology
Iizuka, Fukuoka, 820-8502, Japan

***Abstract*—Membrane computing, which is a computational model inspired by the structures and behaviors of living cells, has considerable attention as one of non-silicon based computing. In the present paper, we propose EN P systems for basic operations, and sorting.**

We first propose three EN P systems for computing three logic operations, OR, AND, and EX-OR functions. All of the EN P systems work in a constant number of steps.

Next, we propose two EN P systems that operates as a half adder and a full adder, and then, propose two EN P system for additions of two binary numbers and n binary numbers. We show the EN P systems for two additions work in $O(m)$ steps and $O(nm)$ steps, respectively.

Finally, we propose two EN P systems for sorting. We propose an EN P system for compare-and-exchange operation. Then, using the EN P system as sub-systems, we propose an EN P system for sorting n numbers, and show that the EN P system works in $O(n)$ steps.

I. Introduction

Membrane computing, which is a representative example of natural computing, is a computational model inspired by the structures and behaviors of living cells. In the initial study on membrane computing, a basic feature of the membrane computing was introduced by Păun[1] as a P system. In the P system, activities in living cells are considered as parallel computing. In [2], [3], algorithms of P system for logical function and additions have been proposed as basic operations.

As a derived model of the P system, A *Numerical P system*, which is inspired from structures of living cells and economics, has been introduced in [4] by Păun. Each region of numerical P system contains a number of numerical *variables* that are evolved according to a *program*. Each program consists of a *production function* and a *repartition protocol*. The production function calculates an output value from numerical variables of the same region, and the output value is distributed into the region and neighboring regions, which are outside and inside regions, by a repartition protocol. *Enzymatic Numerical P systems* [5] (EN P systems, for short) is also a model such that a number of variables, which is called *enzyme*, is used to promote evolution programs.

In the present paper, we propose EN P systems for logic operation, additions, and sorting. we first propose three EN P systems for computing OR, AND, and EX-OR functions. All of the proposed EN P system work in a constant number of steps by using O(1) variables, O(1) membranes, and a constant number of programs.

Second, we propose four EN P systems for additions. The first and second EN P systems operates as a half adder and a full adder, respectively. Both EN P systems work in a constant number of steps by using O(1) variables, O(1) membranes, and a constant number of programs. The third and fourth EN P systems are for addition of binary numbers of m bits. The third EN P system computes addition of two binary numbers of m bits, and works in $O(m)$ steps by using $O(m)$ kinds of variables, $O(m)$ membranes, and programs of size $O(m)$. The fourth P system computes addition of n binary numbers of m bits by using the above EN P system as a sub-system. The EN P system for the addition works in $O(nm)$ steps by using $O(nm)$ kinds of variables, $O(nm)$ membranes, and programs of size $O(nm)$.

Finally, we propose two EN P systems for sorting. The first EN P system executes a compare-and-exchange operation for two integers in constant number of steps by using $O(1)$ variables, $O(1)$ membrane, and a constant number of programs. The second EN P system executes sorting for n integers. The EN P system is based on an idea of the odd-even sort [6], and works in $O(n)$ steps by using $O(n)$ types of variables, $O(n)$ size of membrane, and programs of size $O(n)$.

II. Preliminaries

A. Enzymatic numerical P systems

In the P system, a membrane is a computing cell, in which independent computation is executed, and may contain objects and other membranes. In other words, the membranes form nested structures. In the present paper, each membrane is denoted by using a pair of square brackets, and the number on the right-hand side of each right-hand bracket denotes the label of the corresponding membrane.

For example, $[\ [\ ]_2\ [\ ]_3\ ]_1$ and Fig. 1 denote the same membrane structure that consists of three membranes. The membrane labeled 1 contains two membranes labeled 2 and 3.

We now describe details of the numerical P system. The Numerical P system and the sets used in the system are defined as follows.

$$\Pi_{NP} = (m, H, \mu, (V_m, P_m, V_m(0)), \ldots, (V_m, P_m, V_m(0)), V_o))$$

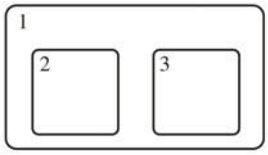


Fig. 1. An example of membrane structure

m: m is the number of membranes in the system.

H: H is a set of labels for membranes. In the present paper, we always use consecutive integer labels such that $H = \{1, 2, \cdots, m\}$. (We assume that a membrane labeled 1, which is called the *skin* membrane, is the outermost membrane, i.e., the *skin* membrane contains all of the other membranes.)

μ: μ is membrane structure that consists of m membranes. Each membrane in the structure is labeled with an element in H.

V_i: V_i is a set of numerical variables in the membrane labeled i.

P_i: P_i is a set of programs in the membrane labeled i.

$V_i(0)$: $V_i(0)$ is a set of initial values of variables in the membrane labeled i.

V_o: V_o is a set of output variables.

The elements of V_i are written in the form $x_{j,i}(t)$, where j is an integer running from 1 to $|V_i|$ at step t. In this paper, V_o is included in the outermost region of the system.

We next formally define a k-th program $pr_{k,i} \in P_i$, which is a program applied in membrane labeled i, as follows.

$$\begin{aligned} pr_{k,i} \quad = \quad & \{F_{k,i}(y_{1,i}, \ldots, y_{l_k,i}) \\ & \rightarrow c_{k,1}|v_1 + c_{k,2}|v_2 + \ldots c_{k,n_i}|v_{n_i})\} \end{aligned}$$

In the above expression, $F_{k,i}(y_{1,i}, \ldots, y_{l_k,i})$ is called a *production function* and computed as arguments $y_{1,i}, \ldots, y_{l_k,i} \subset V_i$. $c_{k,1}|v_1 + c_{k,2}|v_2 + \ldots c_{k,n_i}|v_{n_i}$ is called a *repartition protocol*. $\{v_1, \ldots, v_{n_i}\}$ is a set of variables in the region and in neighboring regions, which are outside and inside regions, and a repartition protocol allocates an output of $F_{k,i}(y_{1,i}, \ldots, y_{l_k,i})$ to the variables according to coefficients $\{c_{k,1}, \ldots, c_{,n_i}\} \subset \mathbf{N}$.

The numerical P system is executed repeatedly according to the following procedure.

(1) Select an applicable program in each region.
(2) Calculate the production functions of the selected programs.
(3) Change the variables used in production functions or repartition protocols to zero.
(4) Allocate output values of (2) according to repartition protocols.

We next describe details of the enzymatic numerical P system. The EN P system is a model such that a number of variables, which are called *enzyme*, in the numerical P systems is used to promote evolution programs. The enzyme in the membrane labeled i is denoted by e_i.

TABLE I
OR FUNCTION

Input 1: $x_{1,1}$	Input 2: $x_{2,1}$	Output: x_{out}
-1	-1	-1
-1	-2	-2
-2	-1	-2
-2	-2	-2

We formally define EN P system Π_{ENP} as follows.

$$\Pi_{ENP} = (m, H, \mu, (V_1, P_1, V_1(0)), \ldots, (V_m, P_m, V_m(0)), V_o)$$

The above definition is the same as the numerical P system because an enzyme is just a numerical variable of the numerical P system. The enzymes e_i are written in the form $e_{j,i}(t)$, where j is an integer running from 1 and $|V_i|$ at step t in the same way as $x_{j,i}(t)$.

We next formally define a k-th program $pr_{k,i}$, which is a program applied in membrane labeled i as follows.

$$\begin{aligned} pr_{k,i} \quad = \quad & \{F_{k,i}(y_{1,i}, \ldots, y_{l_k,i})|_{e_{j,i}} \\ & \rightarrow c_{k,1}|v_1 + c_{k,2}|v_2 + \ldots + c_{k,n_i}|v_{n_i}\} \end{aligned}$$

In case that $e_{j,i}(t) > min\{y_{1,i}(t), \ldots, y_{l_k,i}(t)\}$ at step t, the enzyme works as catalyst, and then the program is applicable.

In this paper, the EN P system is executed repeatedly according to the following procedure.

(1) Select applicable programs in which enzymes are used.
(2) Select an applicable program in each region.
(3) Calculate the production functions of the selected programs.
(4) Change the variables that used in the production functions or repartition protocols to zero.
(5) Allocate output values of (3) by repartition protocols.

The above EN P system has two features, which are maximal parallelism and non-determinism. Maximal parallelism means that all applicable programs are applied in parallel. (However, only one program is applied in a membrane.) On the other hand, non-determinism means that applicable programs are non-deterministically chosen in case that there are several possibilities of the applicable programs.

III. LOGIC FUNCTIONS

A. Input and output for OR function

In this section, we first consider simple EN P system that computes two input OR functions. An input of the function is two binary bits $x_{1,1}$, $x_{2,1}$, and we assume that -1 and -2 denotes general binary numbers 0 (FALSE) and 1 (TRUE), due to computation on the EN P system. Then, outputs of the function on the EN P system are defined in Table I.

B. An overview of EN P system

We now describe an overview of an EN P system for the OR function. The EN P system consists of inner and outer membranes, i.e. membrane structure of the EN P system is $[\ [\ ]_2\ ]_1$.

The computation in the EN P system mainly consists of the following 3 steps.

Step 1: Compute a sum s of two input value. (s is -2 in case of the output is 0, otherwise, the sum is -3 or -4.)

Step 2: Compute two values from the sum of Step 1. The first value is $-s+3$ such that the value is 0 if and only if the output is FALSE. The second value is $s+2$ such that the value is negative if and only if the output is TRUE.

Step 3: Send out an output value depending on the result of Step 2. In case that the result of the first value in Step 2 is a negative value, a value -1 is sent out as FALSE. On the other hand, a value -2 is sent of as as TRUE in case that the result of the second value in Step 2 is negative.

C. Details of EN P system

We now show details of each step of the EN P system for the OR function.

Step 1 is executed by applying the following program $pr_{1,1}$ to two input values $x_{1,1}$ and $x_{2,1}$.
(A programs for the outer membrane)

$$pr_{1,1} \quad = \quad \{2(x_{1,1}+x_{2,1})|_{e_{1,1}} \rightarrow 1|x_{3,1}+1|x_{1,2}\}$$

In this step, a sum s of two input value is computed from two input values, and the sum is copied to two variable $x_{3,1}$ and $x_{1,2}$. ($x_{3,1}$ is a variable in the outer membrane, and $x_{1,2}$ is a variable in the inner membrane.)

Step 2 is executed by applying the following two programs $pr_{2,1}$ and $pr_{1,2}$ in the outer and inner membranes, respectively.
(A program for the outer membrane)

$$pr_{2,1} \quad = \quad \{-x_{3,1}-3|_{e_{1,1}} \rightarrow 1|x_{4,1}\}$$

(A program for the inner membrane)

$$pr_{1,2} \quad = \quad \{x_{1,2}+2|_{e_{1,2}} \rightarrow 1|x_{5,1}\}$$

In this step, two values are computed in the outer and inner membranes in parallel. The first value $-s-3$ is computed according to $pr_{2,1}$ in the outer membrane, and the value is copied to $x_{4,1}$ in the same membrane. The second value $s+2$ is computed according to $pr_{1,2}$ in the inner membrane, and the value is copied to $x_{5,1}$ in the outer membrane.

Step 3 is executed by applying one of the following two programs $pr_{3,1}$ and $pr_{4,1}$ according to two values $x_{4,1}$ and $x_{5,1}$.
(A program for the outer membrane)

$$\begin{aligned} pr_{3,1} \quad &= \quad \{x_{4,1}|_{e_{1,1}} \rightarrow 1|x_{out}\} \\ pr_{4,1} \quad &= \quad \{-2+0x_{5,1}|_{e_{1,1}} \rightarrow 1|x_{out}\} \end{aligned}$$

In this step, a value x_{out} is determined by two values $x_{4,1}$ and $x_{5,1}$. In case that $x_{4,1}$ is negative, an output is FALSE, and a variable x_{out} is set to -1. Otherwise, $x_{5,1}$ is negative, and a variable x_{out} is set to -2 as TRUE.

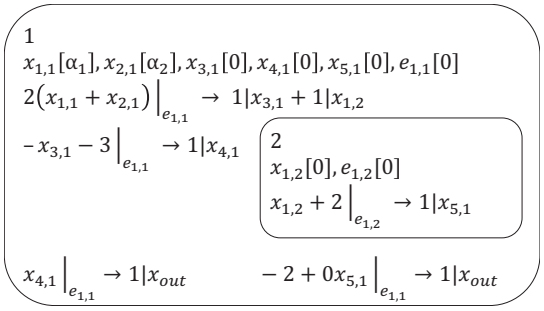


Fig. 2. EN P system for the OR function

TABLE II
AN EXECUTION OF Π_{OR}

	Initial value	Step 1	Step 2	Step 3
Input 1: $x_{1,1}$	-1	0	0	0
Input 2: $x_{2,1}$	-2	0	0	0
Variable $x_{3,1}$	0	-3	0	0
Variable $x_{4,1}$	0	0	0	0
Variable $x_{5,1}$	0	0	-1	0
Enzyme $e_{1,1}$	0	0	0	0
Variable $x_{1,2}$	0	-3	0	0
Enzyme $e_{1,2}$	0	0	0	0
Output x_{out}	-	-	-	-2

We summarize the EN P system for the OR function Π_{OR} and the sets used in the system as follows.

$$\Pi_{OR} = (2, H, \mu, (V_1, P_1, V_1(0)), (V_2, P_2, V_2(0)), V_o)$$

- $H = \{1, 2\}$
- $\mu = [\ [\ ]_2]_1$
- $V_1 = \{x_{1,1},\ x_{2,1},\ x_{3,1},\ x_{4,1},\ x_{5,1},\ e_{1,1}\}$
- $P_1 = pr_{1,1} \cup pr_{2,1} \cup pr_{3,1} \cup pr_{4,1}$
- $V_1(0) = (\alpha_1,\ \alpha_2,\ 0,\ 0,\ 0,\ 0)$
- $V_2 = \{x_{1,2},\ e_{1,2}\}$
- $P_2 = pr_{1,2}$
- $V_2(0) = (0,\ 0)$
- $V_o = \{x_{out}\}$

The above system Π_{OR} is also illustrated in Fig. 2.

In addition, Table II shows an example of variables in execution on Π_{OR} in case of $x_{1,1} = -1$ and $x_{2,1} = -2$.

D. Complexity and other logic operations

Since the number of types of variable in the EN P system Π_{OR} is $O(1)$, and $O(1)$ kinds of programs are used, we obtain the following theorem for Π_{OR}.

Theorem 1: The EN P system Π_{OR}, which computes OR function, works in $O(1)$ steps by using $O(1)$ types of variables, a constant number of membranes, and programs of size $O(1)$. □

We can also propose two EN P systems Π_{AND} and $\Pi_{\text{EX-OR}}$ for other two logic operations, AND and EX-OR, using a similar idea, and obtain the following theorems for Π_{AND} and $\Pi_{\text{EX-OR}}$. (We omit details of the EN P systems due to space limitation.)

Theorem 2: The EN P systems Π_{AND}, which computes AND function, works in $O(1)$ steps by using $O(1)$ types of

TABLE III
HALF ADDER

x_{in_1}	x_{in_2}	$x_{1,1}$	Carry: x_{carry}	Sum: x_{sum}
-1	-1	-2	-1	-1
-1	-2	-3	-1	-2
-2	-1	-3	-1	-2
-2	-2	-4	-2	-1

variables, a constant number of membranes, and programs of size $O(1)$. □

Theorem 3: The EN P systems $\Pi_{EX\text{-}OR}$, which computes EX-OR function, works in $O(1)$ steps by using $O(1)$ types of variables, a constant number of membranes, and programs of size $O(1)$. □

IV. ADDITION

A. Input and output for a half adder

In this section, we first consider an EN P system that executes computation of a half adder of two inputs. We assume that -1 and -2 denotes general binary numbers 0 (FALSE) and 1 (TRUE), due to computation on the EN P system. In addition, we assume that an input of the function is $x_{1,1}$, which is a sum of two input binary bits x_{in_1}, x_{in_2} because the sum is easily computed on the EN P system. Outputs of the function on the EN P system are a carry bit and a sum, which are defined in Table III.

B. An overview of EN P system

We now describe an overview of an EN P system for the half adder. The EN P system consists of a main membrane, and the main membrane includes two EN P systems Π_{AND} and $\Pi_{EX\text{-}OR}$, which are described in the previous section, as sub-system. We define that an output of $\Pi_{EX\text{-}OR}$ is $x_{2,1}$ and an output of Π_{AND} is $x_{3,1}$.

The computation on the EN P system mainly consists of the following 2 steps.

Step 1: Move input values in the main membrane into Π_{AND} and $\Pi_{EX\text{-}OR}$.

Step 2: Compute a carry bit and a sum in Π_{AND} and $\Pi_{EX\text{-}OR}$, respectively, and send out the two values to the main membrane.

Step 3: Send out an output value of Π_{AND} as a carry bit of the half adder, and send out an output value of $\Pi_{EX\text{-}OR}$ as a sum of the half adder.

C. Details of EN P system

We now show details of each step of the EN P system for the half adder.

Step 1 is executed by applying the following program $pr_{1,1}$ to input values $x_{1,1}$.
(A programs for membrane 1)

$$pr_{1,1} = \{2x_{1,1}|_{e_{1,1}} \rightarrow 1|x_{1,EX\text{-}OR} + 1|x_{1,AND}\}$$

In this step, an input variable $x_{1,1}$ is copied to a variable $x_{1,EX\text{-}OR}$ in $\Pi_{EX\text{-}OR}$ and a variable $x_{1,AND}$ in Π_{AND}.

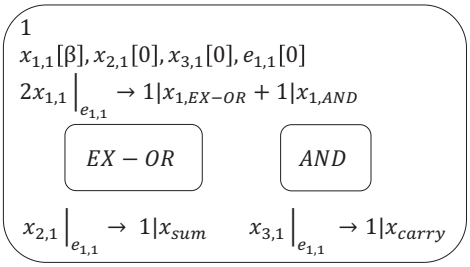


Fig. 3. EN P system for the half adder

In Step 2, two values are computed from EN P systems $\Pi_{EX\text{-}OR}$ and Π_{AND}, and the two output value are sent out from the two sub-systems to $x_{2,1}$ and $x_{3,1}$, respectively.

Step 3 is executed by applying one of the two programs $pr_{2,1}$ or $pr_{3,1}$.
(A program for membrane 1)

$$pr_{2,1} = \{x_{2,1}|_{e_{1,1}} \rightarrow 1|x_{sum}\}$$
$$pr_{3,1} = \{x_{3,h}|_{e_{1,h}} \rightarrow 1|x_{carry}\}$$

After application of the above programs, the output values x_{carry} and x_{sum} are sent out as a carry and a sum of half adder, respectively.

We summarize the EN P system for the half adder Π_{HADD} and the sets used in the system as follows.

$$\Pi_{HADD} = (6, H, \mu, (V_1, P_1, V_1(0)), VP_{AND}, VP_{EX\text{-}OR}, V_{out})$$

- $H = \{h, H_{EX\text{-}OR}, H_{AND}\}$
  - $H_{EX\text{-}OR}$: a label of $\Pi_{EX\text{-}OR}$
  - H_{AND}: a label of Π_{AND}
- $\mu = [\ \mu_{EX\text{-}OR}\ \mu_{AND}\ ]_h$
  - $\mu_{EX\text{-}OR}$: $\mu_{EX\text{-}OR}$ is the membrane structure of Π_{EXOR}
  - μ_{AND}: μ_{AND} is the membrane structure of Π_{AND}
- $Var_1 = \{x_{1,1}, x_{2,1}, x_{3,1}, e_{1,1}\}$
- $Par_1 = pr_{1,1} \cup pr_{2,1} \cup pr_{3,1}$
- $Var_1(0) = (\beta, 0, 0, 0)$
- VP_{AND}: VP_{AND} is the set of $(V_i, P_i, V_i(0))$ in Π_{AND}.
- $VP_{EX\text{-}OR}$: $VP_{EX\text{-}OR}$ is the set of $(V_i, P_i, V_i(0))$ in $\Pi_{EX\text{-}OR}$.
- $V_{out} = \{x_{sum}, x_{carry}\}$

The above system Π_{HADD} is also illustrated in Fig. 3.

In addition, Table IV shows an example of variables in execution on Π_{HADD} in case of $x_{1,1} = -3$. In the example, we assume that program $pr_{2,1}$ is applied before application of program $pr_{3,1}$.

D. Complexity and other operations

Since the number of types of variable in the EN P system Π_{HADD} is $O(1)$, and $O(1)$ kinds of programs are used, we obtain the following theorem for Π_{HADD}.

Theorem 4: The EN P system Π_{HADD}, which executes computation of a half adder of two inputs, works in $O(1)$

TABLE IV
AN EXECUTION OF Π_{HADD}

	Initial value	Step 1	$\cdots$	Step 4	Step 5	Step 6
Input: $x_{1,1}$	-3	0		0	0	0
Variable $x_{2,1}$	0	0		-2	0	0
Variable $x_{3,1}$	0	0		-1	-1	0
Enzyme $e_{1,1}$	0	0		0	0	0
Variable x_{sum}	-	-		-	-2	-2
Output x_{carry}	-	-		-	-	-1

steps by using $O(1)$ types of variables, a constant number of membranes, and programs of size $O(1)$. □

We can also propose three EN P systems Π_{FADD}, Π_{2ADD} and Π_{nADD} for other three operations, which are a full adder, addition of two binary numbers of m bits, and addition of n binary numbers of m bits, using a similar idea. We obtain the following theorems for Π_{FADD}, Π_{2ADD}, and Π_{nADD}. (We omit details of the EN P systems due to space limitation.)

Theorem 5: The EN P system Π_{FADD}, which executes computation of a full adder, works in $O(1)$ steps by using $O(1)$ types of variables, a constant number of membranes, and programs of size $O(1)$. □

Theorem 6: The EN P system Π_{2ADD}, which computes addition of two binary numbers of m bits, works in $O(m)$ steps by using $O(m)$ types of variables, $O(m)$ kinds of membranes, and programs of size $O(m)$. □

Theorem 7: The EN P system Π_{nADD}, which computes addition of n binary numbers of m bits, works in $O(nm)$ steps by using $O(nm)$ types of variables, $O(nm)$ kinds of membranes, and programs of size $O(nm)$. □

V. COMPARE-AND-EXCHANGE AND SORTING

A. Input and output for compare-and-exchange

In this section, we first consider an EN P system that executes compare-and-exchange operation for two input values. Inputs are two negative integers, $x_{1,1}$, $x_{2,1}$, and we also assume that $x_{1,1}$ and $x_{2,1}$ are values between -1 to $n_{\min -1}+1$, where $n_{\min}$ is the minimum values on the EN P system. (The assumption is considered due to computation on the EN P system.)

The result of the computation is outputted to x_{large} and x_{small}. x_{large} and x_{small} are larger and smaller values of the inputs, respectively.

B. An overview of EN P system

We now describe an overview of an EN P system for the compare-and-exchange. The membrane structure of the EN P system is [[[ ]$_3$]$_2$ [ ]$_4$]$_1$.

The computation on the EN P system mainly consists of the following 5 steps.

Step 1: Compute $x_{1,1} - 0.1$.

Step 2: Copy the value of Step 1 to variable v_1 and enzyme e_1 and copy the inputs $x_{2,1}$ to variable v_2 and enzyme e_2.

Step 3: Compare v_1 and e_2, and compare v_2 and e_1.

Step 4: Send out a smaller value according to a result of Step 3. In case that the result of the first comparison in Step 3 is $v_1 < e_2$, send out v_1 as a smaller value, otherwise, send out v_2 as a small value.

Step 5: Send out a larger value according to a result of Step 3. In case that the result of the first comparison in Step 3 is $v_1 < e_2$, send out e_2 as a larger value, otherwise send out e_1 as a larger value.

C. Details of EN P system

We now show details of each step of the EN P system for the compare-and-exchange.

Step 1 is executed by applying a program $pr_{1,1}$ to an input value $x_{1,1}$.
(A programs for membrane 1)

$$pr_{1,1} = \{4(x_{1,1} - 0.1)|_{e_{1,1}} \rightarrow 1|x_{3,1} + 1|x_{5,1} + 1|x_{1,2} + 1|x_{3,2}\}$$

In this step, $x_{1,1} - 0.1$ is computed, and the result is copied to four variable $x_{3,1}$, $x_{5,1}$, $x_{1,2}$, and $x_{3,2}$. ($x_{3,1}$ and $x_{5,1}$ are variables in membrane 1, and $x_{1,2}$ and $x_{3,2}$ are variables in membrane 2.)

Step 2 is executed by applying the following two programs $pr_{2,1}$ and $pr_{1,2}$ in membrane 1 and membrane 2, respectively.
(A program for membrane 1)

$$pr_{2,1} \quad = \quad \{4x_{2,1}|_{e_{1,1}} \rightarrow 1|x_{4,1} + 1|x_{6,1} + 1|x_{2,2} + 1|x_{4,2}\}$$

(A program for membrane 2)

$$pr_{1,2} \quad = \quad \{x_{5,2} - 0.1 + 0x_{1,2}|_{e_{1,2}} \rightarrow 1|x_{5,2}\}$$

In this step, input $x_{2,1}$ is copied to four variable $x_{4,1}$, $x_{6,1}$, $x_{2,2}$, and $x_{4,2}$ by $pr_{2,1}$. ($x_{4,1}$ and $x_{6,1}$ are variables in membrane 1, and $x_{2,2}$ and $x_{4,2}$ are variables in membrane 2.) A formula $x_{5,2} - 0.1$ is computed by program $pr_{1,2}$, and the result is copied to $x_{5,2}$.

Step 3 is executed by applying the following program $pr_{2,2}$.
(A program for membrane 2)

$$pr_{2,2} \quad = \quad \{x_{5,2} - 0.1 + 0x_{2,2}|_{e_{1,2}} \rightarrow 1|x_{5,2}\}$$

In this step, $x_{5,2} - 0.1$ is computed by program $pr_{2,2}$, and the result is copied to $x_{5,2}$. Then, $x_{5,2} = -0.2$, and the value denotes a finish of comparison of the two values $x_{1,1}$ and $x_{2,1}$.

In addition, a result of the comparison is set to an enzyme by applying the following program $pr_{3,2}$ and $pr_{4,2}$.
(A program for membrane 2)

$$\begin{aligned} pr_{3,2} \quad &= \quad \{x_{5,2}|_{e_{2,2}} \rightarrow 1|e_{3,2}\} \\ pr_{4,2} \quad &= \quad \{2x_{3,2}|_{e_{3,2}} \rightarrow 1|e_{2,1} + 1|x_{1,3}\} \end{aligned}$$

In this step, $x_{5.2}$ is copied to $e_{3,2}$ by $pr_{3,2}$, and then, $x_{3.2}$ is copied to $e_{2,1}$ and $x_{1,3}$ by $pr_{4,2}$.

Step 4 is executed by applying the following programs $pr_{3,1}$, $pr_{5,1}$, $pr_{5,2}$, and $pr_{1,3}$. (At first, programs $pr_{3,1}$ and $pr_{5,2}$ are applied, and then program $pr_{1,3}$ is applied.)
(A program for membrane 1)

$$pr_{3,1} \quad = \quad \{2(x_{3,1} + 0.1 + 0x_{4,1})|_{e_{2,1}} \rightarrow 1|x_{large} + 1|x_{1,4}\}$$

(A program for membrane 2)

$$pr_{5,2} \quad = \quad \{x_{4,2}|_{e_{3,2}} \rightarrow 1|e_{3,1}\}$$

(A program for membrane 3)

$$pr_{1,3} \quad = \quad \{n_{min-1} + 0x_{1,3}|_{e_{1,3}} \rightarrow 1|e_{3,2}\}$$

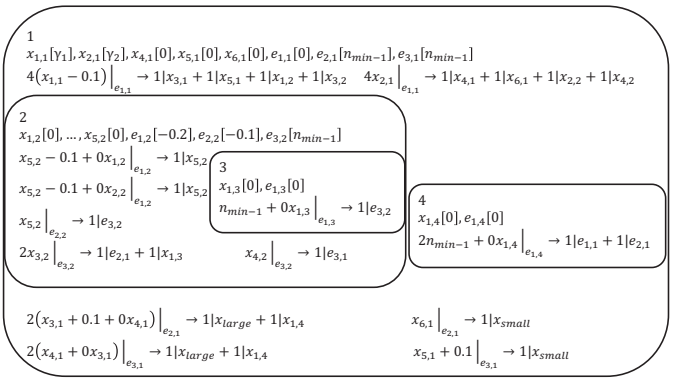


Fig. 4. EN P system for the compare and exchange

In this step, a larger value $x_{1,1}$ is sent out to x_{large} by $pr_{3,1}$, and the value is copied to $x_{1,4}$. A value $x_{4,2}$ is also copied to $e_{3,1}$ by $pr_{5,2}$. In addition, $e_{3,2}$ is reset to n_{min-1} by $pr_{1,3}$.

Finally, Step 5 is executed by applying the following two programs, $pr_{5,1}$ and $pr_{1,4}$.

(A program for membrane 1)

$$pr_{5,1} \quad = \quad \{x_{5,1}+0.1|_{e_{3,1}} \to 1|x_{small}\}$$

(A program for membrane 4)

$$pr_{1,4} \quad = \quad \{2n_{min-1}+0x_{1,4}|_{e_{1,4}} \to 1|e_{1,1}+1|e_{2,1}\}$$

In this step, a smaller value $x_{2,1}$ is sent out to x_{small} by $pr_{5,1}$, and $e_{1,1}$ and $e_{2,1}$ are reset to n_{min-1} by $pr_{5,2}$.

We summarize the EN P system for the compare-and-exchange function Π_{CMPEX} and the sets used in the system as follows.

$$\Pi_{CMPEX} \quad = \quad (4, H, \mu, (V_1, P_1, V_1(0)), (V_2, P_2, V_2(0)), (V_3, P_3, V_3(0)), (V_4, P_4, V_4(0)), V_{out})$$

- $H = \{1, 2, 3, 4\}$
- $\mu = [\ [\ [\ ]_3\ ]_2\ [\ ]_4\ ]_1$
- $V_1 = \{x_{1,1},\ x_{2,1},\ x_{3,1},\ x_{4,1},\ x_{5,1},\ x_{6,1},\ e_{1,1},\ e_{2,1},\ e_{3,1}\}$
- $P_1 = pr_{1,1} \cup pr_{2,1}$
- $V_1(0) = (\gamma_1,\ \gamma_2,\ 0,\ 0,\ 0,\ 0,\ 0,\ n_{min-1},\ n_{min-1})$
- $V_2 = \{x_{1,2},\ x_{2,2},\ x_{3,2},\ x_{4,2},\ x_{5,2},\ e_{1,2},\ e_{2,2},\ e_{3,2}\}$
- $P_2 = pr_{1,2} \cup pr_{2,2} \cup pr_{3,2} \cup pr_{4,2} \cup pr_{5,2}$
- $V_2(0) = (0,\ 0,\ 0,\ 0,\ 0,\ -0.2,\ -0.1,\ n_{min-1})$
- $V_3 = \{x_{1,3},\ e_{1,3}\}$
- $P_3 = pr_{1,3}$
- $V_3(0) = (0,\ 0)$
- $V_4 = \{x_{1,4},\ e_{1,4}\}$
- $P_4 = pr_{1,4}$
- $V_4(0) = (0,\ 0)$
- $V_{out} = \{x_{large},\ x_{small}\}$

The above system Π_{CMPEX} is also illustrated in Fig. 4.

In addition, Table V shows an example of variables in execution on Π_{CMPEX} in case of $x_{1,1} = -3$ and $x_{2,1} = -8$.

D. Complexity and sorting

Since the number of types of variable in the EN P system Π_{CMPEX} is $O(1)$, and $O(1)$ kinds of programs are used, we obtain the following theorem for Π_{CMPEX}.

Theorem 8: The EN P system Π_{CMPEX}, which computes the compare-exchange operation, works in $O(1)$ steps by using $O(1)$ types of variables, a constant number of membranes, and programs of size $O(1)$. □

We can also propose an EN P system $\Pi_{OddEven}$ for soring. The EN P system is based on the odd-even sort [6], which is a well-known parallel sorting algorithm. For $\Pi_{OddEven}$, we obtain the following theorem. (We omit details of the EN P systems due to space limitation.)

Theorem 9: The EN P system $\Pi_{OddEven}$, which performs the odd-even sort for n inputs, works in $O(n)$ steps by using $O(n)$ types of variables, $O(n)$ kinds of membranes, and programs of size $O(n)$. □

TABLE V
AN EXECUTION OF Π_{CMPEX}

	Initial value	Step 1	Step 2	Step 3	Step 4	Step 5
$x_{1,1}$	-3	0	0	0	0	0
$x_{2,1}$	-8	-8	0	0	0	0
$x_{3,1}$	0	-3.1	-3.1	-3.1	0	0
$x_{4,1}$	0	0	-8	-8	0	0
$x_{5,1}$	0	-3.1	-3.1	-3.1	-3.1	0
$x_{5,1}$	0	0	-8	-8	-8	0
$e_{0,1}$	0	0	0	0	0	0
$e_{1,1}$	-10	-10	-10	-3.1	-3.1	-10
$e_{2,1}$	-10	-10	-10	-10	-8	-10
$x_{1,2}$	0	-3.1	0	0	0	0
$x_{2,2}$	0	0	-8	0	0	0
$x_{3,2}$	0	-3.1	-3.1	0	0	0
$x_{4,2}$	0	0	-8	0	0	0
$x_{5,2}$	0	0	-0.1	0	0	0
$e_{0,2}$	0	0	0	0	0	0
$e_{1,2}$	-10	-10	-10	-3.1	-3.1	-10
$e_{2,2}$	0	0	0	0	-3.1	0
$x_{1,3}$	0	0	0	0	0	0
$e_{1,4}$	0	0	0	0	0	0
$x_{1,4}$	0	0	0	0	-3	0
x_{large}	-	-	-	-	-3	-3
x_{small}	-	-	-	-	-	-8

VI. CONCLUSIONS

In the present paper, we proposed EN P systems for logic operations, additions and a sorting. As future work, we are considering an EN P system using the fewer number of membranes and programs.

REFERENCES

[1] G. Păun, "Computing with membranes," *Journal of Computer and System Sciences*, vol. 61, no. 1, pp. 108–143, 2000.

[2] A. Leporati and C. Zandron, "P systems with input in binary form," *International Journal of Foundations of Computer Science*, vol. 17, pp. 127–146, 2006.

[3] A. Fujiwara and T. Tateishi, "Logic and arithmetic operations with a constant number of steps in membrane computing," *International Journal of Foundations of Computer Science*, vol. 22, no. 3, pp. 547–564, 2011.

[4] G. Păun and R. Păun, "Membrane computing and economics: Numerical p systems," *Fundamenta Informaticae*, vol. 73, pp. 213–227, 2006.

[5] A. Pavel, O. Arsene, and C. Buiu, "Enzymatic numerical p systems - a new class of membrane computing systems," *IEEE Fifth International Conference on BioInspired Computing: Theories and Applications (BIC-TA)*, pp. 1331–1336, 2010.

[6] N. Haberman, "Parallel neighbor-sort(or the glory of the induction principle)," *CMU Computer Science Report*, 1972.

MapReduceによる列挙木探索手法の提案と評価

中川　遼
大阪大学基礎工学部情報科学科ソフトウェア科学コース

概要　列挙木探索は，木で表せるような状態空間の中にある「解」を探すことを目的とした探索手法である．しかしながら，状態空間探索においては状態空間は入力に対して爆発的に増加する場合が多く，探索には膨大な時間的・空間的資源が必要となる．本報告ではこの探索を，大規模なデータを分散処理することの出来る MapReduce を用いて行うことを考える．MapReduce は，BigData の重要性が注目を浴びている現在，代表的な分散処理フレームワークとして注目されている．しかし MapReduce を素朴に状態空間の探索に適応すると，MapReduce は並列分散実行中はデータのリアルタイムでの受け渡しが困難なため，マシン間での限定操作が困難である．本報告ではこのように MapReduce には不向きだと考えられる状態空間の探索を，できるだけ手軽で少しでも効率的に MapReduce で行う手法を提案し，具体的な問題に対して提案手法のシステムを実装した上で提案手法の評価を行う．

1　はじめに

状態空間の探索 [?] は，ある問題において“条件を満たす様々な状態 (State) の中から最適な解を探索する手法”であり，組み合わせ最適化問題や人工知能などの分野で幅広く用いられている．しかし多くの場合，問題から生成される様々な状態の数は入力に対して爆発的に増加する [?] ため，全ての状態を探索することは膨大な時間的・空間的資源を必要とする．したがって，状態空間を効率的に探索することは重要な課題である．

本報告では状態空間の探索の中でも最も一般的である，状態をツリー状に列挙した列挙木の探索問題について議論する．

列挙木の探索は，一般的に木の大きさ（状態数）が大きくなるとその分探索にも時間がかかるので，莫大な数の状態を素早く探索するため，多数のマシンを用いた並列分散処理が考えられる．本報告で利用する分散処理フレームワークである MapReduce [?] は，通信量の増加や通信されるデータの大規模化が進む近年において，大規模なデータの分散処理のモデルに幅広く利用されている．

MapReduce はユーザーが Map フェーズと Reduce フェーズの処理内容を記述するだけで簡単に分散処理を行うことができ，記述内容次第で様々な種類の問題に適応可能である．また，多数のマシンで構成されているクラスタ上で動作するように実装されており，大規模データを効率良く分散処理するために Map フェーズと Reduce フェーズに記述された処理を行う．Map フェーズではマスターノードによって分割された入力データそれぞれに対し処理を行い，Reduce フェーズでは Map フェーズの出力を集約する．基本的な流れを（図 1）に示す．

また，MapReduce では，マスターノードが入力データを分割して割り当てる Map タスクはワーカーノードの数に比べて多い方がよい．そして，マスターノードによって分割されたそれぞれのタスクは均等な処理時間で行われるものである方がよい [?].

例えば，ある組み合わせ最適化問題を仮定する．組み合わせの制約を一つ一つ追加していくことによる状態の遷移を，問題の初期状態やすべての制約が追加さ

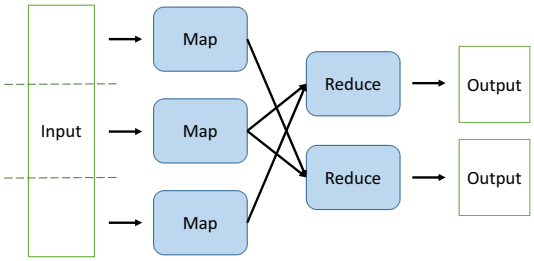


図 1: MapReduce における基本的な動作の流れ

れた状態以外の，まだ追加されていない制約のある状態も含めた状態の存在範囲を状態空間とする．そうすると状態空間のそれぞれの状態と制約の追加による遷移で論理的な木を生成することができる．それが列挙木の生成であり，その列挙木の中に存在する特定の最適解を探すことが列挙木の探索である．

MapReduce を列挙木の探索問題に適応するには，（図 2）のように探索問題の最初に根から生成されるいくつかの子問題（子である状態が持つ問題）を，それぞれ Map フェーズの入力タスク（Map タスク）として割り当てる手法が素朴である．

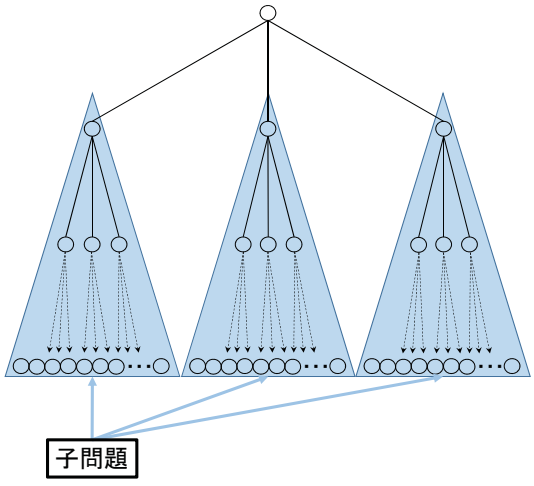


図 2: MapReduce における列挙木探索の素朴なタスク割り当て例

しかしながら，列挙木の探索においては，この手法をそのまま適応するのはあまり効率的ではない．なぜ

なら状態空間の性質上，入力に対してその先に莫大な状態空間が広がる場合が多いため，最初に生成される子問題の数は全体の状態数に対して少なく，それぞれの子問題の木の大きさに非常に大きな差が生まれる可能性がある．したがって Map を行う多数のマシンに割り当てるにはタスクの数が少なく，その上それぞれの Map フェーズの実行時間に大きな差ができる [?]．また，MapReduce の性質上，Map タスクの最中に他の Map タスクや Reduce タスク内で得られた情報をやり取りすることが難しい．

今日，BigData の重要性が注目を浴び，効率的にデータを管理・処理するフレームワークが必要とされている．その中で Hadoop を始めとする BigData 用システムが幅広く使われ，代表的な分散処理フレームワークとして MapReduce が注目されている [?]．列挙木探索を分枝限定法を用いて行うフレームワークはいくつか開発されているが，汎用性の高い MapReduce が多くの企業で導入されている中，列挙木の探索において他の専用のフレームワークを使用するのは不便である．本報告では，本来 MapReduce での計算に不向きだと考えられる列挙木の探索問題を，主にノードへの Map タスクの割り当て方と，Map タスクでの計算内容を操作することにより，総探索時間の短縮を図る．また，提案手法では逐次実行時ほど正確にではないが，限定操作を行うことも可能にする．本稿では提案手法の概要を説明し，提案手法を用いて実装した特定の問題（0-1 ナップサック問題）を解くシステムを Hadoop [?] 上で動作させた実験結果を元に提案手法を評価する．

2 関連研究

2.1 MapReduce

MapReduce は大規模なデータを分散処理するためのプログラミングモデルである．1 台のマスターノードと複数台のワーカーノードで処理を行う．マスターノードは分散処理の指示・管轄を行い，ワーカーノードは実際に処理を行う．マスターノードはユーザーから受け取ったジョブをタスクと呼ばれる小さい単位の処理に分割し，複数のワーカーノードにタスクの処理を要求する．その後，マスターノードはユーザーからのジョブが全て完了されるまで，ジョブやタスクの処理の進捗・タスクのワーカーノードへの割り当て・ワーカーノードの死活などの状況を監視し，管理する．基本的な処理の流れは以下である．

1. マスターノードは入力データを分割しファイルシステムに保持して，複数のワーカーノードに割り当ての指示を出す．
2. マスターノードから割り当てられたデータをファイルシステムから受け取ったワーカーノードは，Map クラスに記述された処理を行い，何らかの中間（Key,Value）ペアを生成し，出力する．（Map フェーズ）
3. Map フェーズから出力された中間（Key,Value）ペアを Key の内容でソートする．このとき同一の Key はまとめられる．（Shuffle フェーズ）
4. Shuffle フェーズによってまとめられた（Key,Value）ペアに対し，Reduce クラスに記述された処理を行い，結果を最終出力として出力する．（Reduce フェーズ）

MapReduce の特徴としては処理の継続性や拡張性，データの局所性などがある．

処理の継続性 MapReduce では，マスターノードでジョブとタスクの進捗を管理している．また，Map 処理や Reduce 処理で扱うデータは基本的には他の Map 処理や Reduce 処理とは関連がなく独立している．したがって，あるタスクがワーカーノードの故障によって中断された場合に，他のワーカーノードに同じタスクを再度割り当てることにより処理を継続することが可能である．

処理の拡張性 Map 処理や Reduce 処理は，処理するデータを分割することにより複数のワーカーノードで並列処理が可能である．したがって，データノードの台数を増やすことで処理性能の向上を図ることができる．

データの局所性 Map 処理はできる限り処理をしているワーカーノードと同じノード上で起動しているファイルシステムのデータを利用しようとする．これによりデータの通信を抑え，ネットワーク帯域の節約や通信コストを抑えることに繋がる．

基本的な MapReduce の実装でユーザーが記述するのは Map フェーズと Reduce フェーズの処理内容だけである．

2.2 Hadoop

本研究では，Hadoop により開発・公開されている Hadoop MapReduce を使用する．

Hadoop は Apache Software Foundation が開発・公開している，大規模データの分散処理や管理を行うためのソフトウェア基盤である．オープンソースソフトウェアとして公開されており，誰でも自由に利用することが可能である [?]．Hadoop は多くの要素で構成されるが，共通の基盤システムである“Hadoop Common”をベースに，前述の Hadoop MapReduce や HDFS，HBase などが主に開発されている．

2.2.1 HDFS

HDFS（Hadoop Distributed File System）は，Hadoop で提供している分散ファイルシステムであり，大きな

ファイルを複数の計算機にまたがって格納することが出来る．HDFS はデータの複製を複数のホストに格納することにより信頼性を確保している．

2.3 Solving Hard Ploblems with Lots of Computers

Solving Hard Ploblems with Lots of Computers は，組み合わせ最適化問題をクラスタなどを用いて並列化し解くことに対する課題を探ることを目的とした研究と，得られた課題を改善できるような動的フレームワークを提案している．この研究では次のように述べられている．「分岐限定法は探索木の力まかせ探索に基づく方法であり，この方法では洗練された限定技術により最適解を含まない部分問題の切り捨てを行う．しかしながら，以降で述べる理由により分岐限定法は MapReduce にうまくマッチしない.」とあり，その理由部分が「分岐限定法は，部分木の計算にどれくらいの時間 (仕事量がいかほどか) がかかるか実行してみるまでわからないため，MapReduce などのフレームワークで実装されている静的な分割では効率はあまり良くならない.」である．これらの事実を踏まえて，本報告では MapReduce のフレームワークをそのままに少しでも効率的に分枝限定法を行う手法を提案する．

3 研究背景・諸定義

3.1 状態と状態空間

組み合わせ最適化問題等において，解を探す各段階における対象の有様や制約を状態（State）と呼び，全ての状態の集合からなる論理的な仮想の空間を状態空間 (State Space) と呼ぶ．状態空間の例を（図 3）に示す．

3.1.1 初期状態

状態空間の状態のうち，入力直後の一番初めの状態を初期状態と呼ぶ．

3.1.2 許容解

状態空間の状態のうち，すべての制約を決定し，もうそれ以上有様の変わらない状態を許容解と呼ぶ．許容解は最終的な最適解になる可能性がある．

3.1.3 目的状態

許容解のうち，求める最適解である状態を目的状態と呼ぶ．

3.1.4 中間状態

状態空間の状態のうち，初期状態でも許容解でもない状態を中間状態と呼ぶ．

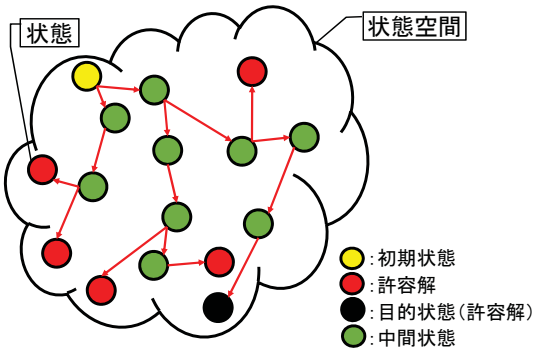


図 3: 状態空間の例

3.2 列挙木

状態空間を表した木のことを列挙木と呼ぶ．状態空間を木で表すには，問題の初期状態を根として表し，ある状態での条件の追加や分岐から他の状態が生成される様子が木における親から子が生成される様子で表される．すなわち，目的状態は列挙木においては葉になる．解も含めて，木における葉が許容解になる．初期状態や各中間状態から生成される次の状態をそのノードの子として記述し，子の問題を子問題と呼ぶ．

例として，各状態において次に生成される状態が高々二つであるような状態空間の列挙木は（図 4）のような二分木になる．

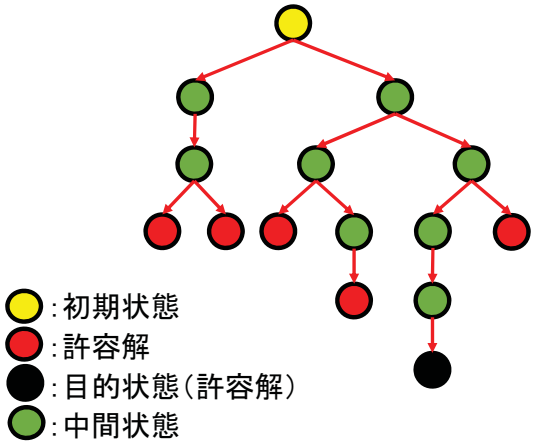


図 4: 列挙木の構築例

3.3 ステップ

状態空間探索の列挙木において，ある中間状態が 1 つ先の状態（子）を探索することを 1 ステップと定義する．（図 5）に例を示す．

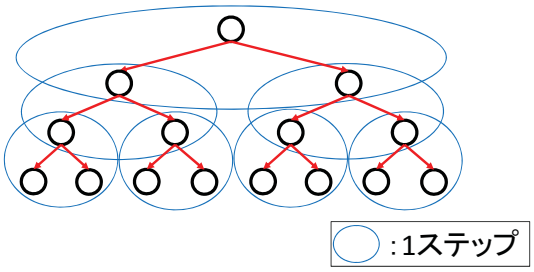


図 5: 1 ステップの定義

3.4 ポテンシャル

各状態が持つ，最適解になり得る期待値のこと．ナップサック問題のような最大化問題のある状態においては，その状態から進んでいくと得られる可能性のある値の上界のこと．

3.5 分枝限定法

分枝限定法（branch and bound）は，組み合わせ最適化問題の最適解を求める汎用アルゴリズムである．分枝操作と限定操作から構成される．以下では，$f(x)$ の最大値を求める最適化問題を例に説明する．$x \in S$ とする．

3.5.1 分枝操作

分枝操作は，解く問題を場合分けにより部分問題に分割することである．つまり，S を $S_1 \cup S_2 \cup S_3 ... = S$ となるような複数の集合 $S_1, S_2, , ,$ に分割する．S_i における $f(x)$ の最大値を v_i とすると，S における $f(x)$ の最大値は $max\{v_1, v_2, ...\}$ である．なお，$S_i \cap S_j \neq \phi$ でもよい．

3.5.2 限定操作

限定操作は，S_i の最大値の上界や下界を計算する手続きである．求めた上界と下界を用いて，以下に述べる枝刈りを行うことが可能になる．

3.5.3 枝刈り

例えば S_i の最大値の上界が S_j の下界より小さければ，S_i には $f(x)$ を最大にする x は存在しないので，それ以上探索をしないようにする．それが枝刈りである．枝刈りにより総探索数は大幅に減少する場合も多い．

3.6 対象問題

本報告では，提案手法を評価するために具体的な問題を解くシステムを実装する．対象とする問題に“0-1 ナップサック問題”を設定する．

3.6.1 0-1 ナップサック問題

0-1 ナップサック問題とは，分枝限定法が有効な代表的な組み合わせ最適化問題であり，NP 困難であることが知られている．

「入る重さの容量（capacity）が C のナップサックと n 個の種類の荷物（item）があり，荷物には各々重さ（weight）と価値（value）がある．ナップサックの容量 C を超えない範囲で荷物をナップサックに詰めていき，ナップサックに入れた荷物の価値の和を最大にするにはどの荷物を入れればよいか.」という問いに対して最大の価値の和とそのときの荷物の中身を求めればよい．一つ一つの荷物については，ナップサックに入れるか入れないかの二択である．n 個の荷物を $x_0, x_1, x_2, ..., x_{n-1}$ とし，荷物 X_i がナップサックに入っているときを $X_i = 1$，入っていないときを $X_i = 0$ と表記すると，0-1 ナップサック問題の解の探索は (図 6) のような列挙木で表すことができる．(図の簡略のため $n = 3$ としている)

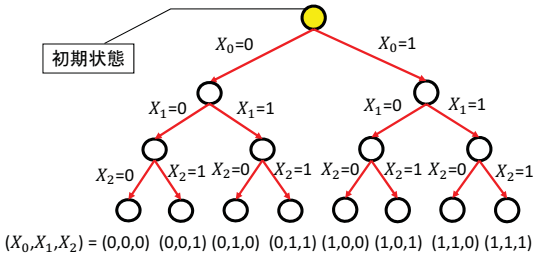


図 6: 0-1 ナップサック問題の列挙木

3.6.2 問題設定

提案システムへの入力として荷物の数 n が，10 個,15 個,20 個,25 個,30 個,35 個,40 個,50 個,100 個の 9 通りの入力サイズを用意する．対象とするのは二分木なので，木の深さは n であり，探索数の上界は $O(2^n)$（枝刈りが起こらずに全探索した場合）である．

4 提案手法

本報告では，MapReduce による列挙木探索の手法を提案する．提案手法の目的は，MapReduce のフレームワークをそのまま利用した上で，本来 MapReduce に不向きな状態空間の列挙木探索を効率的に行うことである．

4.1 概要

提案手法の基本的な流れを記述する．

1. 入力が初期状態，つまり入力用の中間データを保持していなければ，使用するワーカーノード全てに最初から仕事が渡るようにマスターノードは状態数がワーカーの数になるまで探索する．そしてその探索結果の状態を Map タスクとして各ワーカーノードに割り当てる．入力用の中間データがあれば，それをポテンシャルの高い順にワーカーノードに割り当てる．
2. Map タスクでは，ワーカーは受け取った状態から，ユーザーに指定されたステップ数だけ探索を行う．複数状態を保持しているときは，各状態の中で一番ポテンシャルの高い状態を探索する．指定されたステップ数の探索が終わると，辿り着いた全ての状態を Reduce タスクに受け渡す．
3. Reduce タスクでは受け取った状態がまだ探索中の中間状態か，許容解（最終的な解とは限らない）かどうかを確かめる．解ならば現在の最適解（以後，暫定解と呼ぶ）と比較してより良い方を保持し，中間状態ならば次のように限定操作を行う．

   中間状態のポテンシャルが暫定解以下のとき
   その中間状態を探索し続けても暫定解より良い解は得られないので，その中間状態を探索対象から削除する．

   中間状態の現在のポテンシャルが暫定より大きいとき
   その中間状態を次の MapReduce の入力とする．

4. 全ての Map,Reduce タスクの終了後，次の MapReduce への入力があればもう一度 MapReduce を実行する（1. へ戻る）．

   なければ暫定解を最終的な最適解として出力する．

以上の提案手法の流れを（図 7）に示す．

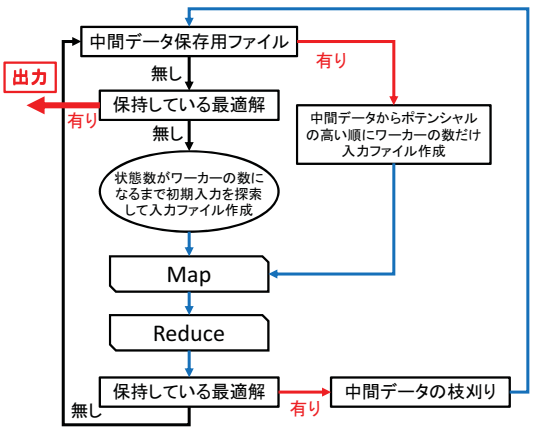


図 7: 提案手法の流れ

4.2 有用性

本報告の主な利点を説明する．状態空間の探索を分枝限定法で高速で行う研究はいくつかあるが，新たなフレームワークを用意しなければならないか，もしくはユーザーに高いプログラミング能力を求められるものばかりである．本報告は，状態空間の列挙木探索が MapReduce フレームワークの処理の手順そのままに，Mapper と Reducer を定義するだけで，高い可能性を持った状態から探索する Best-First Search を実現することができる．また，割り当てる Map タスクの数と 1 回の実行での探索ステップ数をユーザーが指定できるので，実行時のクラスタ環境や取り組む問題の規模に合わせて指定するだけで提案手法を問題により効率良く適応することが可能である．

4.3 提案手法の設定問題への適応

本報告では，探索の高速化のため，n 個の入力荷物は単位重さあたりの価値の大きさ（以降，比率と呼ぶ）が高い順に並んでいるものとする．n 個のデータのソートは $O(nlogn)$ の時間複雑度で可能であり，これは総探索時間に比べて極めて小さく，全体計算量に影響を及ばさないため一般生を失わない．

探索はナップサックに何も入っていない状態を初期状態とし，荷物 X_0 から順に入れるか入れないかで分岐しながら次の状態に進んでいく．初めてある中間状態が n 個目の荷物について入れるか入れないかの探索を終えると，そのナップサックに入っている荷物の価値の和をそのときの最適解として保持する．2 回目以降はそのとき保持している最適解と得られた解を比較し，大きい方を最適解として保持する．一度の MapReduce で進むステップ数はユーザーが指定し，最終的な解が得られるまで複数回 MapReduce を実行する．MapReduce の実行が終わる度に中間状態に対して限定操作を行う．つまり，指定したステップが終わった段階で最適解を保持していれば，その最適解以下のポテンシャルをを持つ中間状態は削除し，その先の探索を行わない．

5 実装システム

5.1 実行環境

本システムの実行環境は以下の表の通りである．

CPU	Intel Core i3 (2cores/4threads)
RAM	2GB(16 台)/4GB(20 台)
HDD	500GB
OS	CentOS 5.8 (Linux Kernel 2.6.x)
Framework	Hadoop 1.2.1 stable

マスターノードを 1 台，ワーカーノードを 35 台の計 36 台で実験を行った．なお，マスターノードには RAM が 4GB のマシンを使用している．

5.2 引数と入力データ

実装システムはマスターノードで実行する．実行時の引数は以下の二つである．

第一引数 使用するワーカーノードの数（以下，nMaps）

第二引数 探索で進むステップ数（以下，nSteps）

ステップ数の大小で探索終了までの実行回数が変わる．探索終了まで手動で繰り返し実行する必要がある．また，問題の入力データはファイルに入れてシステムに受け渡す．入力ファイルの中身は，1 行目にナップサックの容量と荷物の数，2 行目以降に各行 1 つずつ荷物の重さと価値を効率の良い順に記述する．

5.3 Map 処理

Map タスクを行うワーカーノードは，入力の中間状態に対して，まだナップサックに入れるか入れないか決まっていない荷物 X_i のうち i が最少のものについて入るか入らないかを確かめる．ナップサックに荷物 X_i が入るときは入れるパターンと（$X_i = 1$）入れないパターン（$X_i = 0$）の状態を，入らないときは入れないパターンの状態のみを生成する．生成したのが中間状態なら保持，許容解なら Reduce に出力する．ここまでが 1 ステップの動作である．

ユーザーが指定したステップ数が 2 以上のときは指定されたステップ数まで引き続き，前述の動作を自身が生成した中間状態の中で一番ポテンシャルが高い中間状態を取り出し，その状態に対して行う．自身が生成した中間状態がない場合は，ステップが残っていても処理を終了する．

指定されたステップ数まで処理を終えると，保持している中間状態をすべて Reduce に送り，処理を終了する．

5.4 Reduce 処理

受け取った状態に対して，入れるかどうかの判定済みの荷物の数で中間状態か許容解かを判定する．

中間状態ならば，受け取った状態をすべて中間データを保持するためのファイルに書き込む．

許容解ならば，暫定解として最適解を保持するファイルに書き込む．このとき，すでに何らかの最適解を保持しているならば，新たな許容解と比較してナップサックに入っている荷物の価値の和の大きい方を最適解として保持し直す．

Reduce の終了時に保持している最適解を，暫定解を保持するファイルに書き込む．

5.5 マスターノードの処理

マスターノードが Map タスクを生成する動作は 1 回目の実行と 2 回目以降の実行で大きく異なる．1 回目は初期状態から，状態数が nMaps になるまで逐次探索を行い，それによって得られた中間状態を Map タスクの入力とする．2 回目以降は中間データを保持しているファイルからポテンシャルの高い順に中間状態を nMap 個取り出し，得られた中間状態を Map タスクの入力とする．

Map/Reduce タスクの実行後は，暫定解がファイルに保持されていれば，中間データを保持しているファイルの中の中間状態で暫定解よりポテンシャルの低いものは削除する．このとき中間データを保持するファイルが空になれば，そのとき保持している最適解が最終的な解となるので標準端末に出力してシステムを終了する．

6 実験

実装したシステムの評価を行うために他システムと提案システムとの比較実験を行った．

6.1 評価軸

総探索時間（time） 各 Map タスク内での探索時間の合計．

総探索数（count） 各 Map タスク内での探索回数の合計．新しい状態に辿り着く度にインクリメントする．

6.2 比較対象

本報告の提案手法は探索中の限定操作とアルゴリズムの並列実行を実現している．したがって本実験での比較対象は以下の 4 通りである．

Parallel-BnB Hadoop 上で動作し，並列実行を行う．提案手法のようなステップの指定はなく，最初に割り当てられた Map を行うノードが許容解まで探索を行うため，1 回の実行で解が出力される．また，MapReduce では Map タスク実行中のワーカーノードのデータの受け渡しは不可能なので，各々の Map タスク内での限定操作は行っているが，あくまでローカルでの限定操作のみとなっている．

Parallel-DFS Parallel-BnB から各 Map タスクのローカルな限定操作を除いたものである．Hadoop 上で動作し，並列実行ではあるが限定操作は一切行っていない．

Single-BnB 限定操作ありの逐次実行アルゴリズムである．並列処理は一切行っていない．

Single-DFS 限定操作なしの逐次実行アルゴリズムである．

6.3 実験結果

以下の記載する実験結果のグラフでは，提案システムを OurSystem と表記している．

6.3.1 Parallel-BnB

Parallel-BnB システムと提案システムの比較グラフは（図 8）である．

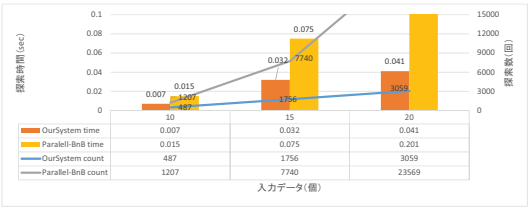


図 8: Parallel-BnB との比較結果

グラフを見ると，どちらのシステムも入力データ数の増加とともに time,count ともに増加しているが，増加の割合が圧倒的に Parallel-BnB の方が大きい．入力データが 20 個の時点で提案システムは time,count ともに Parallel-BnB より明らかに小さく，このまま入力を増加させるとこの差はさらに広がっていくと考えられる．

6.3.2 Parallel-DFS

Parallel-DFS システムと提案システムの比較グラフは（図 9）である．

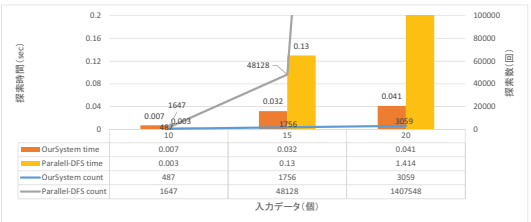


図 9: Parallel-DFS との比較結果

グラフを見ると，どちらのシステムも入力データ数の増加とともに time,count ともに増加している．入力データが 10 個のときは，両システムともに time はとても小さくあまり差はないが，count は分枝限定法を実装している分提案システムが Parallel-DFS の 4 分の 1 程度になっている．入力データが 15 個になると time も提案システムが Parallel-DFS の 3 分の 1 程度になり，count の差はさらに広がっている．入力データ 20 個でも提案システムと Parallel-DFS の time,count の差はさらに大きく広がっているので，このまま入力を増加させても差は広がり続けると考えられる．

6.3.3 Single-BnB

Single-BnB システムと提案システムの比較グラフは（図 10）である．本グラフは差の見やすさの都合上,（図 8）などとは違い，縦軸 time の単位を (msec) にして log スケールで表示した．

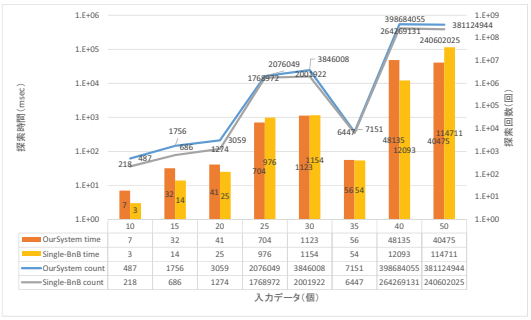


図 10: Single-BnB との比較結果

グラフの下のデータテーブルを見ると，入力データが 10〜30 個まではデータの増加とともに両システムにおいて time,count ともに増加傾向にあるが，入力が 35 のところで両システムともに 30 の場合に比べ count,time ともに大幅に減少している．40 から 50 にかけても少しではあるが同様の現象が見られるが，これらは両システムともに起こっていることから，入力データに原因があると考えられる．基本的にナップサック問題は入力データが増えると探索する状態数も増えるが，入力データを効率順にしている本実験では稀に探索初期の段階でナップサックの容量が満たされる最適解が発見されるようなケースが起こり得る．したがって，35 と 50 はこのような入力データであったことが予想される．

Single-BnB は逐次的に限定操作を行いながら探索するので，count は必ずその入力データに対して最小となる．提案システムは並列処理中のマシン間での限定操作を行えないため，count は必ず Single-BnB の方が小さくなる．しかし，状態空間探索は入力が増えていくと探索する状態数が爆発的に増えるため，逐次実行では入力データ数がある程度大きくなると実行時間もとても増えるはずである．したがって time に着目して比較を行う．入力データサイズが小さいと，逐次で行っても短い時間で終わるため，並列処理の利点を生かせないので，入力データが 20 個までは time はわずかに Single-BnB の方が小さい．これは探索数の差が原因だと考えられる．しかし，入力データが 25 個か

らは提案システムの方が time が小さくなり，その差は 40 のときに一気に広がっている．これは入力データが 30〜40 程度が，並列処理により 1 探索あたりの探索速度は速いが探索数は多くなる提案手法と逐次実行で 1 つずつ無駄なく探索を行う Single-BnB とが総探索時間の点でおよそ釣り合うデータ数であることを表している．したがって，このまま入力を増加させても time の差は広がり続けると考えられる．

6.3.4 Single-DFS

Single-DFS システムと提案システムの比較グラフは（図 11）である．

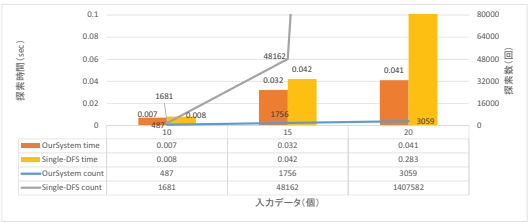


図 11: Single-DFS との比較結果

Single-DFS は初期状態から逐次的に探索を行い，限定操作をしないため，入力データに対する最大の探索数となる．グラフを見ると，入力データが 10 個のときは入力が小さいために両システムの time にはほとんど差がないが，count はすでに提案システムの方が 4 分の 1 で探索している．入力データが 15 個の時点で既に count の差は爆発的に広がっており，time も提案システムの方が小さい．入力データ 20 個でも time,count の差はさらに大きく広がっているので，このまま入力を増加させても差は広がり続けると考えられる．

6.4 考察

4 つの比較グラフを見ると，入力データ数が少ない場合は，提案システムと他のシステムとの間に大差はない．しかしながら，入力データ数が多くなるにつれて大幅に提案システムの探索時間は他のシステムに比べ短くなっている．また，他の並列分散システム（Parallel-BnB,Parallel-DFS）ではメモリ容量を超えて実行できなかった $n = 100$ のときの探索も可能であった点から，提案システムはそれらのシステムに比べて実行時のメモリ使用量が少ないこともわかる．これは処理を複数回の MapReduce に分けて行うため，1 回の実行時に探索する状態数が少ないことで，必要とするメモリが少なくなると考えられる．

以上より，提案システムは 4 つの比較対象システムよりも大きな入力に対応でき，入力が大きくなればなるほど相対的に探索が速くなることがわかる．

7 今後の課題

本報告の提案手法は，MapReduce を複数回実行する性質上，MapReduce の起動時や終了時などにワーカーの起動, 終了やタスクの読み込みなど探索時間以外の冗長な時間が生まれる．本研究における実験では,1 回の実行につき 10〜15 秒の冗長な時間が発生した．そのため，探索時間や探索回数が小さくても全体の実行時間では（図 12）のように 1 回の MapReduce の実行で探索する場合よりも長くなってしまう場合が多い．

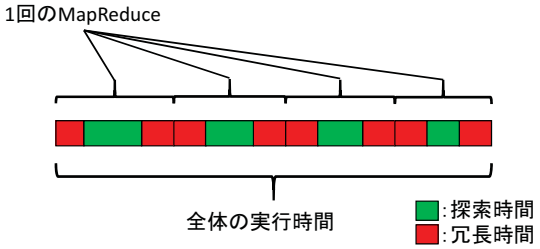


図 12: 提案システムでの全体実行時間例

短い探索時間を生かすために，MapReduce の繰り返し実行をスムーズに行うように改良する必要がある．この繰り返し実行を高速化するフレームワークとして，Apache Spark [?] が登場した．Apache Spark の動作の流れは MapReduce と基本的に同じだが，繰り返し利用するデータを MapReduce はストレージで保存していたのに対して Apache Spark は自身のメモリに保存することで出し入れの高速化を図っている．したがって，提案手法を Apache Spark フレームワークで実装すれば全体の実行時間も短縮できる可能性があるため，現在取り組んでいる．総探索時間の削減が実験により確認できた提案手法に対して，Apache Spark の仕様で全体の実行時間が削減されるのであれば，Apache Spark での提案手法の実装は計算量・時間の両面で有用であると言える．

また，提案システムの汎用化も大きな課題である．本報告での手法では 1 回目の実行の際の初期状態を nMaps 個の状態になるまでの探索や,2 回目以降の入力となる中間データの扱いを問題の内容に合わせてメインクラスに記述しなければならない．これらの部分を汎用化することが出来れば，MapReduce を通して状態空間の探索をより手軽に効率良く行うことが出来るようになるなど期待できる．

8 おわりに

本報告では状態空間におけるポテンシャルを利用し，MapReduce での限定操作の実現による列挙木探索手法の提案と評価と行った．一度の MapReduce で許容解まで探索せずに中間状態を保持することにより，得られたその時の最適解との比較で限定操作を行ってい

る．それにより，少なくとも今回取り上げた 0-1 ナップサック問題については，MapReduce で素朴に探索したときよりも大幅に短い探索時間と少ない探索回数を可能であることを実験により示した．また，現在の提案手法は，探索対象がよほど大きな状態空間でない限り MapReduce を繰り返し実行することによる冗長時間が探索時間に比べて長くなるという現在の一番の課題を踏まえ，今後の展望を述べた．

参考文献

[1] Jeffrey Dean and Sanjay Ghemawat,”MapReduce: Simplified Data Processing on Large Clusters”, USENIX Association OSDI ’04: 6th Symposium on Operating Systems Design and Implementation,2004

[2] Internet Engineering Task Force(Feb. 14 2013 14:40 UTC), ”Combinatorial_explosion”, http://en.wikipedia.org/wiki/Combinatorial_explosion

[3] Sandy Ryza,”Solving Hard Problems with Lots of Computers”, Brown University Department of Computer Science,2012

[4] Tom White,”Hadoop”,O’REILLY,2011

[5] Mihai Budiu and Daniel Delling and Renato F. Wernek,”DryadOpt: Branch-and-Bound on Distributed Data-Parallel Execution Engines”,Microsoft Research Silicon Valley Mountain View, CA, USA,2012

[6] Richard Neapolitan and Kumarss Naimipour,”Foundations of algorithms(4th Edition)”,JONES AND BARTLETT PUBLISHERS,2011

[7] Internet Engineering Task Force(Feb. 14 2014 14:45 UTC), ”hadoop PoweredBy”, http://wiki.apache.org/hadoop/PoweredBy

[8] Internet Engineering Task Force(Feb. 14 2014 14:45 UTC), ”hadoop”, http://hadoop.apache.org/

[9] Internet Engineering Task Force(Feb. 14 2014 14:50 UTC), ”BigTable”, http://ja.wikipedia.org/wiki/BigTable

[10] Internet Engineering Task Force(Sep. 12 2014 11:30 UTC), ”Apache Spark”, https://spark.apache.org/

Topology Control in Cooperative Ad Hoc Wireless Networks

T. F. Neves [1]

Department of Computer Science
University of Brasilia
Brasilia, Brazil

J. L. Bordim[2]

Department of Computer Science
University of Brasilia
Brasilia, Brazil

Abstract

Cooperative communication (CC) is a technique that exploits spatial diversity allowing multiple nodes to cooperatively relay signals to the receiver so that it can combine the received signals to obtain the original message. CC can be combined with topology control to increase connectivity at the cost of a small increase in energy consumption. This work focuses on exploring CC to improve the connectivity with a sink node in ad hoc wireless networks. More precisely, this work proposes a new technique, named CoopSink, that combines CC and topology control techniques to increase connectivity to a sink node while ensuring energy-efficient routes. Simulation results show that connectivity and routing to the sink cost can be improved up to 6.8 and 2.3 times, respectively, when compared with other similar strategies.

Keywords: Topology Control, Wireless Networks, Network Protocols, Cooperative Communication.

1 Introduction

Ad hoc wireless networks are networks where the nodes can communicate with each other without resorting to a centralised infrastructure. These networks have a large number of civil and military applications, ranging from communication support in battlefield, search and rescue operations to object monitoring and tracking. One of the major challenges in ad hoc networks is to reduce energy consumption as these nodes are often powered by batteries [2]. As battery replacement may not be a feasible option during operation, alternatives to improve and optimize energy expenditure in wireless networks are of great interest. This facts have driven the quest for power saving strategies aiming to extend the network lifetime [25,5]. These energy

[1] Email: tfn.thiago@cic.unb.br
[2] Email:bordim@unb.br

saving proposals can be grouped in two main categories: (*i*) techniques that allow the nodes to alternate between active/idle operational modes; and (*ii*) techniques that allow the nodes to adjust their transmission power. Recent works in the first category can be found in [34,9,30]. Topology control strategies fall in the second category and have been largely explored in the literature [22,8]. Topology control consists in allowing wireless nodes to select a subset of neighbouring nodes and/or adjust the transmission power with the objective of reducing energy consumption while maintaining network connectivity [7,3,23].

In traditional multiple-hop wireless networks, intermediate nodes cooperate with each other to assist in the task of relying data packets from a source node to the desired destination. Note that this process occurs at the network layer. Cooperative communication (CC), on the other hand, is a physical layer technique that allows single antenna devices to benefit from some advantages of Multiple-Input Multiple-Output (MIMO) systems by exploring the spatial diversity [28]. This technique allows nodes to improve signal quality and transmission range. In CC, when a source node transmits a packet, a set of *helper nodes* in the vicinity of the source overhear the signal and, simultaneously, relay independent copies of the same signal to the destination node. The destination node then combines the received signals to obtain the original packet.

Recent works have explored CC with topology control techniques to reduce energy consumption [5]. Ways to increase network connectivity and improve network lifetime has been investigated [35,33]. However, to the best of our knowledge, no work so far explored CC to increase the connectivity to the sink in ad hoc wireless networks. Link failure due to battery depletion and node failure may prevent wireless nodes to reach the sink. In this context, CC can be explored to improve network connectivity and to allow the establishment of alternative routes to the sink node.

This work presents a new technique, named CoopSink, that combines CC and topology control in an ad hoc wireless network to increase connectivity to the sink while ensuring energy efficient routes. This proposal could be applied to the environment described in [4], where there is a sink node equipped with a large range radio for query broadcast and the nodes cooperate to overcome link failures and report information to the sink. This scenario is similar to that found in the Amazon Tall Tower Observatory (ATTO) project, where the objective is to position a high central tower in the middle of the Amazon forest and, with the help of smaller and strategically placed sensors, to obtain reliable estimates of sources of greenhouse gases like CO_2, CH_4 and N_20 [24]. The proposed technique has been evaluated through simulation and the results have confirmed that CoopSink is able to improve network connectivity and provide energy-efficient routes to the sink. More precisely, the simulation results show that connectivity and routing to the sink improved up to 6.8 and 2.3 times, respectively, as compared with other similar strategies.

The remaining of this paper is organised as following. Section 2 makes an overview of related works on topology control and cooperative communication. Section 3 describes the communication and network models and formalises the main problem addressed in this work. Section 4 describes the CoopSink protocol and Section 5 presents the simulation results that compares the proposed scheme with other similar and prominent strategies. Section 6 concludes the work.

2 Related Works

This section presents a brief review of the closely related works that explore topology control and cooperative communication in wireless networks.

2.1 Topology Control

Topology Control is a technique that alters the network topology based on some given conditions. For instance, topology control can be used to optimize network power consumption, reduce routing cost and the number of control messages, improve throughput or meet certain QoS requirements [5,15,11].

According to [5], topology control protocols can be classified as: (*i*) centralised; or (*ii*) distributed. Centralised protocols consider that global information is available, such as topological information, routing information, global memory status, and so on. However, even when global information is at hand, it has been proven that finding strongly connected topologies with minimum total energy consumption is a NP-complete problem [6]. Among centralised protocols, Ramanathan et al. [27] proposed alternatives to optimize network connectivity while improving network lifetime. Distributed protocols consider k-hop neighbouring information, where k is typically one or two. Li et al. [20] propose a cone-based algorithm for TC that aims to optimize energy consumption while maintaining network connectivity. To achieve this, each node adjusts its transmitting power to cover a number of neighbouring nodes, under the condition that they lay at most α degrees apart from each other. The authors show that a degree of $\alpha = \frac{5\pi}{6}$ is enough to preserve network connectivity. Several optimized solutions of the basic algorithm are also discussed and a beacon-based protocol is defined for topology maintenance. In a more recent work, Li et al. [21] proposed a Localised Minimum Spanning Tree (LMST) algorithm. The LMST works by having each node building a localised MST based on 1-hop neighbouring information. The final topology is constructed so that the maximum node degree is 6. Comprehensive surveys can be found in [22,14].

2.2 Cooperative Communication (CC)

Cooperative communication (CC) is a technique introduced by [19] and [26] that allow single antenna devices to explore characteristics of MIMO systems. In cooperative communication, a set of nodes transmit independent copies of the original signal. The intended receiver obtain independent versions of the transmitted signal which reduces the fading effect through multi-path propagation. In this communication model, each wireless node is assumed to transmit data and to act as a cooperative agent, relaying data from other users. CC was previously used in energy efficient broadcasting [1], constructing connected dominating sets [32], routing [17], among others applications.

CC techniques can be classified as *amplify-and-forward* and *decode-and-forward* [19]. In the former, a node that receives a noise version of the signal, amplify and relay this noisy version. The receiver then combines the information sent by both sender and relay nodes. When decode-and-forward is employed, a relay node must first decode the signal before retransmitting it. As the cost of a CC-link

area usually higher than conventional links, ways to select suitable nodes is usually employed. Among the techniques used to identify the best set of relay nodes are received Signal-to-Noise Ratio (SNR) and/or remaining battery energy.

This work considers cooperative communication employing *decode-and-forward* approach where the relay nodes are selected based on the SNR of the received signal. This technique requires each node to have a dedicated memory to store data packet and a signal processor that can estimate the SNR of each received packet as in [19].

2.3 Topology Control in Cooperative Ad Hoc Networks to Extend the Link Coverage

Few works in the literature have considered the use of topology control and CC to improve network coverage in ad hoc networks. The possibility of link coverage extension using CC was investigated in [18,31]. In this context, Cardei et al. [5] studied the use of topology control combined with CC with the objective of obtaining strongly connected topologies with minimum energy consumption. The authors showed that this problem is NP-complete and two localised and distributed algorithms were proposed. Both algorithms take as input the result of a traditional topology control algorithm (without CC). The first algorithm uses a distributed decision process where each node uses information of neighbours with at most two hops. The second algorithm iteratively assigns transmission power to the nodes, using one hop neighbour information.

Yu et al. [33] proposed a centralised topology control scheme aiming to increase network connectivity and reduce transmission power. To minimise the number of cooperative communication links (CC-links) and to reduce transmission cost, a polynomial and an exponential (but optimal) helper decision algorithms were proposed. Zhu et al. [35] consider the problem of selecting energy efficient paths when CC-links are used. The authors propose two topology control algorithms to build cooperative energy spanners in which the energy efficiency of individual paths are guaranteed. Both algorithms can be executed in a distributed or localised fashion. The work in [33] focus on maintaining network connectivity with the objective of minimising global energy consumption. Similarly, the work in [35] focused on reducing energy consumption by selecting efficient routes.

In ad hoc settings, link failure due to battery depletion and node failure may prevent wireless nodes to reach the desired destination. In this context, CC can be explored to improve network connectivity and to allow the establishment of more efficient routes. This work addresses this problem. To the best of our knowledge, this is the first work to explore the use of CC-links to improve network connectivity to a sink node in a wireless ad hoc networks.

3 Network Model and Problem Definition

This section first describes the CC model and the corresponding network model that are used in this work. In a second moment, the model is exemplified and the main problem of this work is formalised. The model defined in this section is similar to that used in [5,33,35].

3.1 CC model

Consider a wireless ad hoc network where each node v_i can adjust its transmission power P_i with values within the interval $[0, P_{MAX}]$. When $P_i = 0$, the node's radio is off and, when $P_i = P_{MAX}$, the node's radio operates with maximum power. In traditional cooperative communication models, a sender node v_i can directly communicate with a receiver node v_j only if the transmission power of v_i satisfies Equation 1.

$$P_i(d_{i,j})^{-\alpha} \geq \tau \qquad (0 \leq P_i \leq P_{MAX}), \tag{1}$$

where: α is the path loss exponent, usually between 2 and 4, and represents the rate of signal fading with increasing distance; $d_{i,j}$ is the Euclidian distance between v_i and v_j; and τ is the minimum Signal-to-Noise Ratio (SNR) received by v_j, so that v_j can decode the signal and obtain the original message.

CC takes advance of the physical layer design to combine partial signals to obtain complete information [28]. This way, a communication between the nodes v_i and v_j can be achieved with CC if v_i transmits its signal jointly with a set of *helper* nodes $H_{i,j}$ and the sum of its transmission power satisfies Equation 2. In CC, a helper node is a node that cooperatively retransmit the signal along with the transmitting node.

$$\sum_{v_k \in v_i \cup H_{i,j}} P_k(d_{k,j})^{-\alpha} \geq \tau \qquad (0 \leq P_i \leq P_{MAX}) \tag{2}$$

Figures 1a and 1b exemplify a scenario where CC could be used to increase connectivity in an ad hoc network. In Figure 1a, there are three nodes (v_1, v_2 e v_3), which are close to each other, and one distant node (v_4). The transmission radius, that is, the transmission power of v_1 allows it to reach nodes v_2 and v_3 directly. Node v_2 and v_3 have a single neighbouring node, in this case, node v_1. Node v_4 is outside radio range of the other nodes.

When CC is employed, node v_1 could select the nodes v_2 and v_3 as its helpers to transmit to v_4, that is, $H_{1,4} = \{v_2, v_3\}$. After selecting these nodes as helpers, v_1, in a first moment, transmit its data to v_2 and v_3. In a second moment, v_1 and its helpers transmit the same data to v_4, amplifying the v_1 transmission radius. If the combined SNR received in v_4 is greater than τ, the node is able to decode the signal and obtain the original data from v_1, as illustrated in Figure 1b. When node v_1 transmits its data for its helper nodes v_2 and v_3, in a first moment, v_4 also receives partial data from v_1. In some CC models, such partial data is used by v_4 in the signal decoding process, during the second moment of the CC. In this work, such partial data is ignored for simplicity (as in [33] and [35]). Physical layer techniques to implement CC can be find in [13].

3.2 Network model

Consider an ad hoc network with n nodes that are capable to receive and combine partial, received data, in agreement with the CC model. The network topology is modelled as a planar graph $G = (V, E)$, where $V = \{v_1, v_2, ..., v_n\}$ is a set of wireless nodes and E is the set of communication edges. An edge $v_i v_j \in E$ symbolises that node v_i can transmit data to v_j directly and/or using CC. $N(v_i)$ is the direct neighbour set of v_i within its maximum transmission range R_{MAX}, for every $v_k \in$

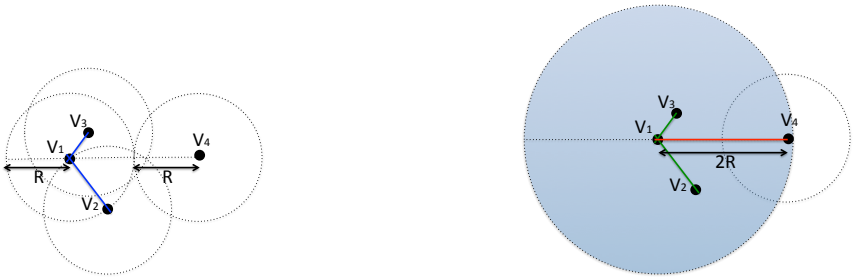


Fig. 1. (a) Scenario with three nodes (v_1, v_2 e v_3) within radio reach of each other and one distant node (v_4). (b) Node v_1 uses nodes v_2 and v_3 as helper nodes to increase radio range thus reaching node v_4.

$N(v_i)$, there is $P_i \leq P_{MAX}$ such that $P_i(d_{i,k})^{-\alpha} \geq \tau$, following Equation 1. In other words, v_i can directly communicate with its neighbours in $N(v_i)$. Each node $v_i \in V$ has an unique ID and knows its own location information. Node IDs and location information are exchanged among the nodes. Each node $v_i \in V$ has an unique radio and runs on battery power. Given the previous information, we define several important concepts, similar to those in [35].

Definition 1 *(Direct edge): A direct edge $\overline{v_iv_j}$ is an edge in E representing that node v_i can transmit data to node v_j directly, that is, P_i is such that v_i can achieve v_j when $P_i \leq P_{MAX}$. A solid horizontal line over the nodes denote a direct edge.*

Definition 2 *(Helper node set): $H_{i,j}$ symbolises the set of helper nodes of v_i in a cooperative communication with v_j. It is assumed that all helper nodes need to be direct neighbours of v_i, that is, $H_{i,j} \subseteq N(v_i)$, where $N(v_i)$ is the set of all direct neighbours of v_i. In other words, all the elements in $N(v_i)$ are helper nodes candidates.*

Definition 3 *(CC edge): A CC edge $\widetilde{v_iv_j}$ is an edge of E that represents that node v_i can transmit data to v_j cooperatively by using a set of helper nodes $H_{i,j}$. A wavy horizontal line is used to denote a CC edge.*

Definition 4 *(Helper edge): A helper edge is an edge from v_i to one of its helper nodes in $H_{i,j}$. For example, in Figure 1b, node v_1 uses the nodes v_2 and v_3 as helper nodes to create a CC edge between v_1 and v_4, that is, $H_{1,4} = \{v_2, v_3\}$, this way, the edges $\overline{v_1v_2}$ and $\overline{v_1v_3}$ are considered helper edges.*

Definition 5 *(Network topology): The union of all direct edges and CC edges are $\overline{E}$ and $\widetilde{E}$, respectively. Similarly, the direct communication graph and the CC communication graph are denoted as $\overline{G} = (V, \overline{E})$ and $\widetilde{G} = (V, \widetilde{E})$, respectively. Note that $E = \overline{E} \bigcup \widetilde{E}$. Following the notation, note that, if $v_iv_j \in E$, then: $v_iv_j = \overline{v_iv_j}$ if v_iv_j is a direct edge; and $v_iv_j = \widetilde{v_iv_j}$ if v_iv_j is a CC edge.*

Definition 6 *(Direct edge weight): The weight of a direct edge $\overline{v_iv_j}$ is defined as:*

$$w(\overline{v_iv_j}) = \tau d_{i,j}^{\alpha}.$$

Definition 7 *(CC edge weight): The weight of a CC edge $\widetilde{v_iv_j}$ is defined as:*

$$w(\widetilde{v_iv_j}) = w_d(H_{i,j}) + (|H_{i,j}| + 1)w_{CC}(H_{i,j}),$$

where:

- $|H_{i,j}|$: *is the number of elements in* $H_{i,j}$;
- $w_d(H_{i,j}) = \left(\frac{\tau}{(\max_{v_k \in H_{i,j}} d_{i,k})^{-\alpha}}\right)$: *is the minimum energy consumption of node* v_i *to communicate with the most distant node in* $H_{i,j}$;
- $w_{CC}(H_{i,j}) = \left(\frac{\tau}{\sum_{v_k \in v_i \bigcup H_{i,j}} (d_{k,j})^{-\alpha}}\right)$: *is the minimum energy consumption of node* v_i *to communicate with* v_j*, jointly transmitting with its helpers in* $H_{i,j}$

Note that, according to Equations 1 and 2, the following relation must be true to exist an CC edge:

$$\max\left(w_d(H_{i,j}), w_{CC}(H_{i,j})\right) \leq P_{MAX}$$

In a CC from v_i to v_j, the node v_i must, in a first moment, send its data to its helper nodes in $H_{i,j}$ and, in a second moment, node v_i and its helpers must simultaneously send the same data to v_j. This way, the weight of the CC edge consists in the sum of the communication costs of these two moments. $w_d(H_{i,j})$ is the cost of the first moment while $w_{CC}(H_{i,j})$ is the individual node cost to transmit a data with CC, that is, it must be multiplied by $(|H_{i,j}| + 1)$, that is, the number of nodes that are involved in the CC between v_i and v_j. In this work, the CC model is simplified assuming that the transmission power of v_i and its helper nodes are the same. Furthermore, it is only considered the power consumption in each sender node.

Definition 8 *(Directional path cost): Given a source node* v_i *and a destination node* v_j *in a graph* $G = (V, E)$*, there is a directional path between* v_i *and* v_j *if, and only if, there is a sequence of vertex:*

$$v_i, v_{i_1}, v_{i_2}, ..., v_{i_{k-1}}, v_{i_k}, v_j \in V,$$

such that:

$$(v_i v_{i_1}), (v_{i_1} v_{i_2}), ..., (v_{i_{k-1}} v_{i_k}), (v_{i_k} v_j) \in E.$$

The directional path cost between v_i *and* v_j *in a graph* G *is defined as:*

$$\pi_G(v_i, v_j) =$$

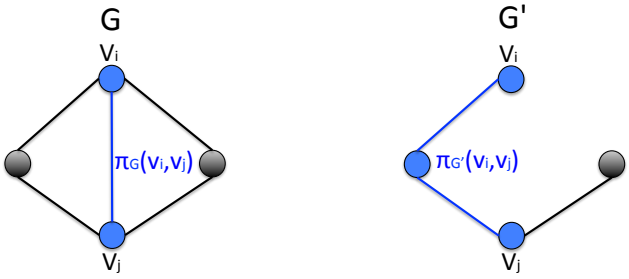
$$w(v_i v_{i_1}) + w(v_{i_1} v_{i_2}) + ... + w(v_{i_{k-1}} v_{i_k}) + w(v_{i_k} v_j).$$

That is, the sum of the weight of all the edges, both direct edges and CC edges, that belong to the path between v_i *and* v_j*. The cost of the shortest path in* G *between* v_i *and* v_j *is defined as:*

$$\min(\pi_G(v_i, v_j)).$$

Definition 9 *(ESF - Energy Stretch Factor): Let* $G' = (V, E')$ *be a subgraph of* $G = (V, E)$*,* $E' \subseteq E$*,* G' *e* G *are connected graphs. The ESF of a pair of nodes* v_i *and* v_j *in* G' *with respect to the same nodes in* G *is defined as:*

$$\rho_G^{G'}(v_i, v_j) = \frac{\min(\pi_{G'}(v_i, v_j))}{\min(\pi_G(v_i, v_j))},$$



(a) (b)

Fig. 2. (a) Example of graph $G = (V, E)$. (b) Example of graph $G' = (V, E')$, $E' \subseteq E$. $\pi_G(v_i, v_j)$ is the path cost from v_i to v_j in graph G. The ESF of a pair of nodes $v_i, v_j \in V$ of G' with respect to G is equal to $\rho_G^{G'}(v_i, v_j) = \frac{\min(\pi_{G'}(v_i,v_j))}{\min(\pi_G(v_i,v_j))}$.

Consider the ESF of G' with respect to G as:

$$\rho_G^{G'} = \max_{v_i, v_j \in V} \rho_G^{G'}(v_i, v_j).$$

That is, the greatest ESF between any pair of nodes in V in G' with respect to G. Figure 2 illustrates this concept.

Definition 10 *(CE-t-S - Cooperative Energy t-Spanner): Let $G' = (V, E')$ be a subgraph of $G = (V, E)$, $E' \subseteq E$, G and G' are connected graphs. G' is a CE-t-S with respect to G if its ESF is less than a constant t, that is:*

$$\rho_G^{G'} \leq t.$$

Definition 11 *(COE-t-S - Cooperative Oriented Energy t-Spanner): COE-t-S is a specific case of the CE-t-S. Let $G' = (V, E')$ be a subgraph of $G = (V, E)$, $E' \subseteq E$, G' and G are not necessarily connected (unlike the COE-t-S). Consider a particular node $v_o \in V$ and constant t. G' is a COE-t-S with respect to G and v_o if:*

$$\max_{v_i \in V} \mu(v_i, v_o) \leq t,$$

where:

$$\mu(v_i, v_o) = \begin{cases} \max \rho_{G'_P}^{G'_P}(v_i, v_o), & \text{if there is a path from } v_i \text{ to } v_o; \\ 0, & \text{otherwise.} \end{cases}$$

3.3 Computing the CC cost

To exemplify the described model, consider the graph represented by Figure 1a. This is a direct graph $\overline{G} = (V, \overline{E})$, where $V = \{v_1, v_2, v_3, v_4\}$ and $\overline{E} = \{\overline{v_1v_2}, \overline{v_1v_3}, \overline{v_2v_1}, \overline{v_3v_1}\}$. Consider, just for exemplification, that $\alpha = 1$, $\tau = 1$ and that $R_{max} = R$. Note that $\alpha = 1$ is not practical in real situations, but it makes the calculation easier in an example. Thus, we can compute the maximum transmission power of each node $v_i \in V$ based on Equation 1:

$$P_{MAX}(d_{MAX})^{-\alpha} = \tau,$$
$$P_{MAX}(R)^{-1} = 1,$$
$$P_{MAX} = R.$$

Note that each node $v_i \in V$ has a maximum transmission power of R with the provided settings. In $\overline{G}$, nodes v_2 and v_3 are neighbours of node v_1, that is, $N(v_1) = \{v_2, v_3\}$. This way, node v_1 could select any non-empty subset of $N(v_1)$ as its helper nodes to create a CC edge from v_1 to v_4, that is $H_{1,4} \subseteq N(v_1)$, following Definition 2. The new graph with the CC edge that comes from v_1 to v_4 is illustrated in Figure 1b.

Consider that, for the proposed example, $d_{12} = R$, $d_{13} = \frac{R}{2}$, $d_{14} = 2R$, $d_{24} = 1.73R$ and $d_{34} = 1.80$. From Equation 2, we can compute the transmission power that node v_1 needs to achieve node v_4 using nodes v_2 and v_3 as helper nodes:

$$\sum_{v_k \in v_1 \cup \{v_2, v_3\}} P_k(d_{k,4})^{-1} \geq 1,$$
$$= P_1(d_{14})^{-1} + P_2(d_{24})^{-1} + P_3(d_{3,4})^{-1} \geq 1,$$
$$= P_1 \geq \frac{1}{\frac{1}{2R} + \frac{1}{1{,}73R} + \frac{1}{1{,}80R}},$$

$$= P_1 \geq 0.612R.$$

Note that all the involved nodes (v_1, v_2 e v_3) operate with the same transmission power, that is, $P_1 = P_2 = P_3$. Thus, the minimum transmission power for v_1 to achieve v_4 with CC is equal to $P_1 = 0.612R$. This transmission power is bellow the previous seen maximum transmission power that is equal to $P_{MAX} = R$. To compute the CC edge weight for the edge $\widetilde{v_1v_4}$ we use Definition 7:

$$w(\widetilde{v_1v_4}) = \frac{1}{R^{-1}} + \frac{(|H_{1,4}| + 1)}{\sum_{v_k \in v_1 \bigcup \{v_2, v_3\}} (d_{k4})^{-1}}$$
$$= R + (|H_{1,4}| + 1) * 0.612R,$$
$$= 2.836R.$$

Note that the weight of the CC edge $\widetilde{v_1v_4}$, using nodes v_2 and v_3 as helpers, is equal to $w(\widetilde{v_1v_4}) = 2.836R$. Likewise the previous example, we use Equation 2 to compute the transmission power that node v_1 needs to achieve v_4 using just node v_2 as a helper node and using just node v_3 as a helper node. Table 1 summarises all the calculus made for this example.

Analysing the $w(\widetilde{v_1v_4})$ column in table, one can verify that the weight of the CC edge $\widetilde{v_1v_4}$ is lower using just node v_3 as a helper node. This happens because, as can be observed from Definition 7, the weight of a CC edge is the sum of the cost to send a data packet from the source node to its helpers and the cost to transmit collaboratively with its helpers. As, in the example, node v_2 is farther away from from v_1 than v_3, respectively distances of R and $\frac{R}{2}$. Hence, node v_1 spends more power to reach node v_2 than to reach v_3. So, using just node v_3 as its helper, allows for grether battery savings. The problem of efficient helper set selection in CC is a challenging issue and has been addressed in other works [12,29].

Table 1
Weight of the CC edge $\widetilde{v_1v_4}$ (graph in Figure 1b) for different helper node sets $H_{1,4}$.

$H_{1,4}$	P_1	$w(\widetilde{v_1v_4})$
$\{v_2, v_3\}$	$0.612R$	$2.836R$
$\{v_2\}$	$0.928R$	$2.855R$
$\{v_3\}$	$0.947R$	$2.395R$

3.4 Problem formulation

Consider an ad hoc network, where the nodes are scattered in a fixed area and there is an unique sink in the network border. The sink node is responsible for collecting and requesting information from the other nodes. The nodes are fixed, that is, no mobility is assumed, and the underline network graph can get disjoint due to link failures, battery depletion and so on. This work's proposal consists in exploring CC communication to improve network connectivity while keeping energy expenditure as low as possible. Using the defined notation, given a network topology $\overline{G} = (V, \overline{E})$, where $v_o \in V$ is a sink node, the goal of this work is to propose a technique that uses CC to create a graph G and a subgraph G', $G' \subseteq G$, such that G' is a COE-t-S with respect to G and the node v_o, following the Definition 11. In the following section, this work's proposal is described.

4 Proposal

In this section, it is described the *CoopSink* (short for Cooperative Sink), a topology control technique for cooperative ad hoc networks. This technique aims to improve node connectivity to the sink while the energy consumption of the routes to the sink are minimised. In a first moment, a greedy heuristic for the helper set selection problem is described, then the CoopSink technique is described as a sequence of four steps. The formulation and algorithms of this section are based on the definitions of Section 3 and it is assumed a central computing unit.

4.1 Greedy Helper Set Selection Algorithm (GHSS)

Algorithm 1 describes a greedy heuristic, proposed by [33] and adapted for this work. This heuristic is used to select the most efficient helper nodes for CC and can be used in both directional and bi-directional graphs. The algorithm's inputs are: a transmitter node v_i; a set $N(v_i)$ of v_i's neighbours; and a receiver node v_j. The output is the set of helper nodes $H_{i,j}$, such that the weigh of the CC-links $\widetilde{v_iv_j}$ is minimised (although optimal solutions are not guaranteed). Consider the function prototype $GreedyHelperSetSelection(v_i, N(v_i), v_j)$ to refer to Algorithm 1. In the following, the algorithm and the gaining function is described.

4.1.1 Algorithm general description

Algorithm 1 has three main steps:

- **Step 1 (lines 1-11)**: this step consists on sorting the neighbour nodes from v_i ($N(v_i)$) according to a gain heuristic. The greater the gain that a node in $N(v_i)$ brings, according to the heuristic, up ahead it will be positioned in the vector B. The following functions are used in this step: the function $head(X)$ returns the first element from a vector or a set X; the function $remove(X, Y)$ removes the element Y from the set X; the function $SortDescending(X)$ receives the vector X and returns the same vector but sorted in descending order; while function $indexTerms(X)$, where $X = [\frac{b_{k_1}}{c_{k_1}}, \frac{b_{k_2}}{c_{k_2}}, ..., \frac{b_{k_{|N(v_i)|}}}{c_{k_{|N(v_i)|}}}]$, returns the vector $Y = [v_{k_1}, v_{k_2}, ..., v_{k_{|N(v_i)|}}]$. The gain function is described in section 4.1.2.
- **Step 2 (lines 12-21)**: This step consists on adding the elements from vector C to the helper set $H_{i,j}$ so that v_i and its helpers should have the minimum transmission power to create the CC-link $\widetilde{v_i v_j}$, according to Equation 2. If, after adding all the elements in C, the coupled power from these nodes is not enough to create the CC-link $\widetilde{v_i v_j}$, an error message is returned (lines 16-18);
- **Step 3 (lines 22-31)**: This step consists on adding the maximum number of helper nodes to the set $H_{i,j}$ so that the inclusion of the next node does not increase the CC-link weight. In the case of the inclusion of a CC-link increases the CC-link weight or all the elements in C were already tested, the function returns the set $H_{i,j}$ (lines 23-24). The function $WeightCC(v_i v_j, \Omega)$, where $v_i v_j \in E_t$ and $\Omega \subseteq V$, returns the CC-link weight of the link $v_i v_j$ using the nodes in Ω as helper nodes.

4.1.2 Gain function

The gain function in Algorithm 1 (Step 1) is used to sort the nodes in $N(v_i)$ in order of greater gain in terms of energy consumption to create a CC-link $\widetilde{v_i v_j}$. Consider, for each node $v_k \in N(v_i)$, the following values:

- $b_k \leftarrow \frac{\tau}{(d_{i,j})^{-\alpha}} - \frac{\tau}{\sum_{v_l \in \{v_i, v_k\}} (d_{l,j})^{-\alpha}}$: the amount of power that node v_i can save if it adds node v_k as a helper. Note that the first element in the subtraction is the cost for v_i directly communicate with v_j and the second element of the subtraction is the power for v_i cooperatively communicate with v_j using node v_k as a helper;
- $c_k = \frac{\tau}{(d_{i,k})^{-\alpha}}$: The energy cost for v_i to directly communicate with the helper v_k.

As the objective here is to reduce the weight of the CC-link, the values for b_k, that represents the gain, should be maximised while the values of c_k, that represents the cost to communicate with the helper node, should be minimised. This way, the gain function considers the ratio $\frac{b_k}{c_k}$ as a metric that indicates which node brings greater gain, that is, if $\frac{b_k}{c_k}$ is maximum for $k = h$, so the inclusion of the helper node v_h will bring a greater gain than other helper nodes.

4.2 CoopSink: Proposal Description

This section describes the steps of CoopSink technique. The CoopSink consists on the execution in sequence of the following four steps:

- **Step 1**: Create a topology graph, where each node creates as much edges as its maximum transmission power allows;

Algorithm 1 : $GreedyHelperSetSelection(v_i, N(v_i), v_j)$

Require: $v_i, N(v_i), v_j$
Ensure: $H_{i,j}$;
1: # (Step 1)
2: Set: $A \leftarrow N(v_i)$;
3: vector: $B \leftarrow []$, $C \leftarrow []$;
4: **while** $(v_k \leftarrow head(A)) \neq \emptyset$ **do**
5: $b_k \leftarrow \frac{\tau}{(d_{i,j})^{-\alpha}} - \frac{\tau}{\sum_{v_l \in \{v_i, v_k\}} (d_{l,j})^{-\alpha}}$;
6: $c_k = \frac{\tau}{(d_{i,k})^{-\alpha}}$;
7: $B \leftarrow [B, \frac{b_k}{c_k}]$;
8: $remove(A, v_k)$;
9: **end while**
10: $B \leftarrow SortDescending(B)$;
11: $C \leftarrow indexTerms(B)$
12: # (Step 2)
13: $k \leftarrow 0$, $\Omega \leftarrow v_i$;
14: **while** $(\sum_{v_k \in \Omega} P_{MAX}(d_{k,j})^{-\alpha}) < \tau$ **do**
15: $k \leftarrow k + 1$;
16: **if** $k > |C|$ **then**
17: return error;
18: **end if**
19: $H_{i,j} \leftarrow H_{i,j} \cup A[k]$;
20: $\Omega \leftarrow \Omega \cup H_{i,j}$
21: **end while**
22: # (Step 3)
23: **while** $(k \leq |N(v_i)|)$ **do**
24: **if** $(k = |N(v_i)|)$ **or**
$(WeightCC(\widetilde{v_i v_j}, \Omega) < WeightCC(\widetilde{v_i v_j}, \Omega \cup C[k+1]))$ **then**
25: return $H_{i,j}$;
26: **else**
27: $k \leftarrow k + 1$;
28: $H_{i,j} \leftarrow H_{i,j} \cup C[k]$;
29: $\Omega \leftarrow \Omega \cup H_{i,j}$;
30: **end if**
31: **end while**
32: return $H_{i,j}$;

- **Step 2**: Use CC to create as much CC-links as possible in the network;
- **Step 3** From previous step's graph, use topology control to cut edges such that energy efficient routes to the sink are kept;
- **Step 4**: Adjust the transmission power of the nodes, to minimise the energy consumptions but maintaining the network connectivity.

The following sub-sections better describe each of previous steps.

4.2.1 Step 1: Construction of graph $\overline{G_P}$

Algorithm 2 describes the CoopSink's first step. This step consists on creating all the possible direct edges in a graph topology, since all nodes operates with its maximum transmission power P_{MAX}. Algorithm 2 takes as input: the node set $V = \{v_1, v_2, ..., v_n\}$ and its information in the map; the maximum transmission power P_{MAX}. As output, we have the direct graph $\overline{G_P}$. Figures 3a and 3b show an input graph and the computed $\overline{G}$ graph. In Figure 3a we have a 500x500m area with $n = 50$ nodes no edges, that is, $\overline{E} = \emptyset$. Figure 3b illustrates the resulting graph when nodes create edges based on its maximum transmission power. In this example, $P_{MAX} = 4900$ and $R_{MAX} = 70$. Note that the resulting graph is not necessarily connected.

4.2.2 Step 2: Construction of graphs $\widetilde{G}$ and G

Algorithm 3 describes the CoopSink's second step. This step consists on creating all the possible CC edges taking as input the graph $\overline{G} = (V, \overline{E})$ from the previous step and returning the graph G, which has both direct and CC edges. As described in Subsection 2.3, the efficient helper set selection in CC is a challenging problem, since it is computationally costly. Therefore, heuristics can be used to handle this problem. In this work we consider the greedy heuristic proposed in [33]. This heuristic, which has the $GreedyHelperSetSelection(v_i, v_j)$ prototype, receive as input: the source node v_i; and the destination node v_j. The output is the helper set $H_{i,j}$. As an example, consider the graphs represented in Figures 3b and 3c. The graph in Figure 3b is the output graph from Step 1 and the input graph of Step 2. Graph in Figure 3c is the output from Step 2. In these graphs, direct edges $\overline{v_i v_j} \in E$ are shown in blue, CC edges $\widetilde{v_i v_j} \in E$ are shown in red and helper edges $H_{i,j}$ are shown in green. Note that, in this example, a large number of CC edges were created, however, these are direct edges, that is, the existence of $\widetilde{v_i v_j} \in E$ does not imply $\widetilde{v_j v_i} \in E$.

4.3 Step 3: Removing edges from G

Algorithm 4 describes the CoopSink's third step. This step consists on creating a graph G' such that the graph is a COE-t-S with respect to G and a node $v_o \in V$. It receives as input: a graph G; a constant t; and a node $v_o \in V$. It returns a graph G'. Figures 3c and 3d illustrate what happens in this step. The graph in Figure 3c is a graph with several direct and CC edges and is one of the inputs of Step 3. The other inputs are, in this example, $t = 1.4$ and v_o on location $(0, 0)$ in the cartesian plane. The resulting graph from Step 3, illustrated in Figure 3d, is a COE-t-S with respect to the graph in Figure 3c. The resulting graph is a direct graph oriented to the node v_o, that is, all the connected nodes have direct routes to v_o. Note that previously unconnected nodes on Figure 3b now have a path to the sink node v_o. One may note that some nodes area still disconnected in the final stage. Figure 3d shows a node that failed in establishing CC links to its neighbouring nodes.

Algorithm 2 : $CoopSinkStep1(V, P_{MAX})$

Require: V e P_{MAX};
Ensure: $\overline{G}$;
1: $\overline{G} = (V, \emptyset)$;
2: **for** $(v_i, v_j \in V)$ **do**
3: **if** $(P_{MAX}(d_{ij})^{-\alpha} \geq \tau)$ **then**
4: Add $\overline{v_i v_j}$ to $\overline{G_P}$;
5: **end if**
6: **end for**

Algorithm 3 : $CoopSinkStep2(\overline{G})$

Require: $\overline{G}$;
Ensure: G;
1: $\widetilde{G} = (V, \emptyset)$;
2: **for** $(v_i, v_j \in V)$ **do**
3: $H_{ij} \leftarrow GreedyHelperSetSelection(v_i, v_j)$;
4: **if** $(w_{CC}(H_{i,j}) \leq P_{MAX})$ **then**
5: Add $\widetilde{v_i v_j}$ to $\widetilde{G}$;
6: **if** $(\overline{v_i v_j} \in \overline{G}$ and $w(\overline{v_i v_j}) > w(\widetilde{v_i v_j}))$ **then**
7: Remove $\overline{v_i v_j}$ from $\overline{G}$;
8: **end if**
9: **end if**
10: **end for**
11: $G = \overline{G} + \widetilde{G}$;

4.3.1 Step 4: Adjusting the transmission power

Algorithm 5 describes the CoopSink's last step. This step consists on adjusting the transmission power of all the nodes in the input graph G' such that the COE-t-S property is maintained and the energy consumption is minimised.

In the following section, the CoopSink technique is compared with other techniques from the literature.

5 Simulation Results

In the previous section, it was proposed a new technique for topology control in cooperative ad hoc networks, which focus on increasing connectivity while maintaining efficient routes to the sink. In this section, it is used simulation to compare the proposal to others in the literature. The following techniques were chosen for comparison: CoopBridges [33] and MST-Kruskal [16]. CoopBridges starts from the direct graph $\overline{G}$ (output from CoopSink Step 1) and splits all connect components into clusters. Creates as much bi-direction CC edges as possible within clusters. Finally, uses a distributed MST algorithm to remove inter-clusters edges and intra-clusters latter. MST-Kruskal grows a minimal spanning tree (MST) one edge at a time by finding an edge that connects two trees in a forest of growing MSTs, receives as input the graph $\overline{G}$.

Initially, it was our intention to compare the proposal also with the Greedy-

Algorithm 4 : *CoopSinkStep*3(G, t, v_o)

Require: $G = (V, E)$;
Ensure: $G' = (V, E')$;
1: $G' \leftarrow G$;
2: $k \leftarrow 1$;
3: vector $A \leftarrow$ receives all the edges in E ordered by weight;
4: **while** $(k \leq |B|)$ **do**
5: $v_i v_j \leftarrow A[k]$;
6: $G'_P \leftarrow G'_P - v_i v_j$;
7: **if** $(\max_{v_i \in V} \mu(v_i, v_o) \leq t)$ **then**
8: # The deletion of edge $v_i v_j$ will harm the ESF;
9: $G'_P \leftarrow G'_P + v_i v_j$;
10: **if** $(v_i v_j = \widetilde{v_i v_j})$ **then**
11: Remove all the edges from v_i to all the nodes in $H_{i,j}$ from B;
12: **end if**
13: **end if**
14: $k \leftarrow k + 1$;
15: **end while**

Algorithm 5 : *CoopSinkStep*4(G')

Require: $G' = (V, E')$;
1: **for** $v_i \in V$ **do**
2: $a \leftarrow \max_{\overline{v_i v_j} \in G'_P} w_d(H_{i,j})$;
3: $b \leftarrow \max_{\widetilde{v_i v_j} \in G'_P} w_{CC}(H_{i,j})$;
4: $P_i = \max\{a, b\}$;
5: **end for**

(Add/Del)-Link technique [35], however, this technique assumes that the topology created, after adding the CC edges, would be necessarily connected. This hypothesis is not considered in both CoopSink and CoopBridges. Here, a resulting topology is the topology created after the application of a topology control algorithm. To evaluate the CoopSink performance, it is considered the following metrics:

- M1 - Connectivity to the sink: The percentage of nodes that have a route to the sink in the resulting topology;
- M2 - Average transmission power: The average of transmission power assigned to each node in the resulting topology;
- M3: ESF to the sink: Consists on creating a subgraph $G' = (V, E')$ from $G = (V, E)$, $E' \subseteq E$, and to verify the value for the constant t when it is considered that G' is a COE-t-S with relation to G;
- M4: Number of CC edges: The number of CC edges in the resulting topology.

The reason to consider M1 is to evaluate the connectivity improvements that CoopSink can bring to the network. M2 is justified once minimising energy consumption is a basic concern in ad hoc networks. M3 is used to verify the goodness of the routes connecting to the sink when compared with another proposals. M4 enumerates the average number of CC edges in the resulting topology. M4 is also

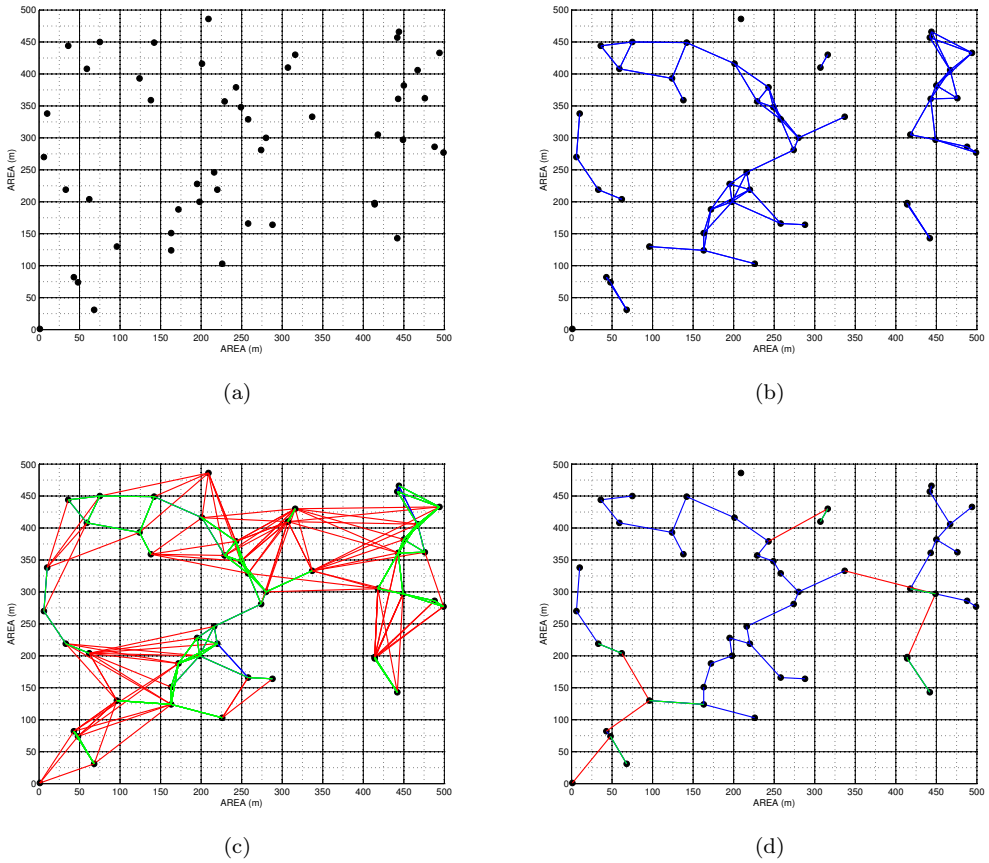


Fig. 3. (*a*) Network topology example with $n = 50$ nodes and $|E| = 0$. (*b*) Direct graph $\overline{G}$ created from (*a*) when nodes operate with maximum power P_{MAX}. (*c*) Graph G, created from the graph in (*b*), by adding all the possible CC edges. (*d*) COE-t-S created from (*c*).

useful to better understand the average energy cost, as CC edges consume more power than direct links.

In order to perform the evaluation, a simulation was developed in Matlab [10], following the steps described in Section 3 for CoopSink and the algorithm described in [33] for CoopBridges. The MST-Kruskal implementation is already present in Matlab. The CoopBridges implementation was validated with the original results. Similarly to other works [33,35], the simulation process takes the following parameters: $n = 10, 20, ..., 100$ nodes are randomly positioned in a 500x500m area; the sink node is always node v_1 at position $(0, 0)$; The PLE is equal to 2 ($\alpha = 2$); $P_{MAX} = 4900$; SNR is equal to 1 ($\tau = 1$); and $t = 1.0, 1.4, 1.8$ are the values for constant t. The simulation results have been drawn from an average of a 100 simulations. The compared proposals are: CoopBridges, MST, CoopSink-1.0, CoopSink-1.4 and CoopSink-1.8, where the last three are the CoopSink technique with t assuming values of 1.0, 1.4 and 1.8, respectively. The following subsections present and analyse the simulation results for each metric considered.

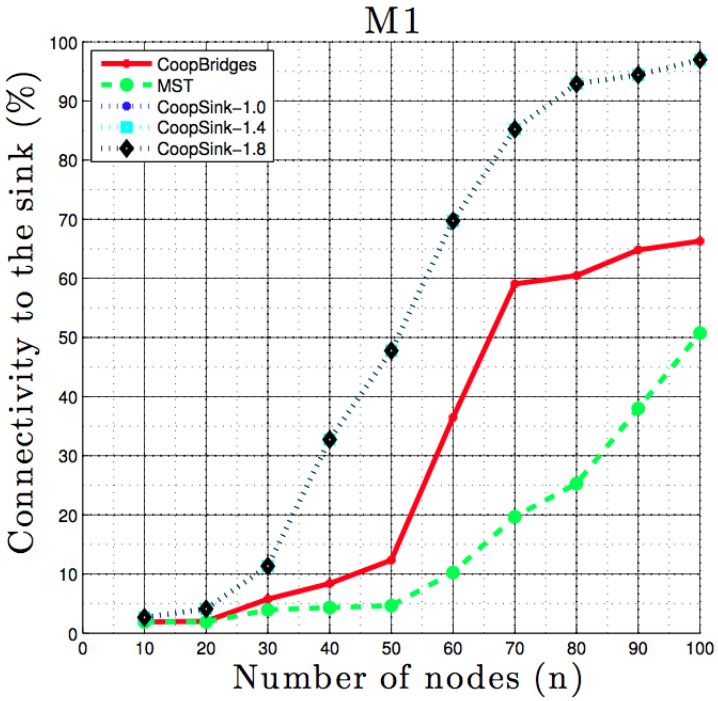


Fig. 4. Simulation results for M1.

5.1 M1: Connectivity to the sink

Figure 4 presents the simulation results for metric M1. The x-axis shows the number of network nodes and the y-axis shows the percentage of the nodes that have a route to the sink. The figure shows that the parameter t had no impact on the CoopSink connectivity to the sink. As can be seen in the figure, CoopSink allows for greater connectivity than CoopBridges and MST. Indeed, with 80 nodes, the connectivity to the sink using CoopSink exceeds 90%. The CoopSink's connectivity to the sink was up to 3.8 times better than CoopBridges and up to 6.8 times better than MST.

5.2 M2: Average transmission power

Figure 5 presents the simulation results for metric M2. The x-axis shows the number of network nodes and the y-axis shows the average transmission power for each node. As the MST does not create CC edges, the energy consumption in this technique is lower that that in CoopSink and CoopBridges. For this reason, the average transmission power of the nodes in CoopSink tend to be greater. However, using CoopSink-1.4 and CoopSink-1.8, the energy consumption is not very far from CoopBridges (up to 30% in the best case scenario). CoopSink, on the other hand, provides better connectivity. For smaller t values, the energy consumption tends to increase as the freedom to remove edges from the graph reduces.

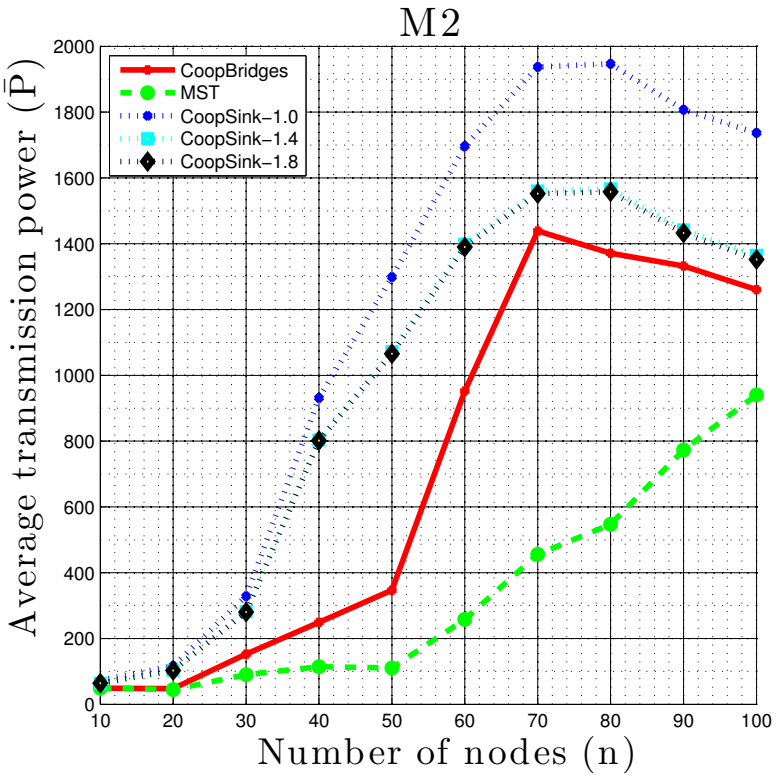


Fig. 5. Simulation results for M2.

5.3 M3: ESF to the sink

Figure 6 presents the simulation results for metric M3. As before, the x-axis shows the number of nodes while the y-axis presented the ESF values to the sink for each of the simulated techniques. As the ESF is measured comparing two graphs, the following graphs are used in the comparison:

- CoopSink-(1.0/1.4/1.8): Graph G' (output of CoopSink's Step 3) with graph G (output of CoopSink's Step 2);
- CoopBridges: Resulting graph from CoopBridges technique with the graph G (output of CoopSink's Step 2);
- MST: Resulting graph from MST with graph $\overline{G}$ (output of CoopSink's Step 1).

The results show that:

- The CoopSink variations obey the ESF to the sink, established during configuration. This means that the observed values for the ESF to the sink must be smaller or equal to 1.0 for CoopSink-1.0, smaller or equal to 1.4 for CoopSink-1.4 and smaller or equal to 1.8 for CoopSink-1.8.
- Both CoopBridges and MST have no commitment with the ESF to the sink. This mean that some of the routes to the sink can be quite inefficient in terms of energy consumption;
- CoopBridges obtained a ESF to the sink up to 2.34 times greater than CoopSink, while the MST obtained a ESF to the sink up to 2.29 greater. The greater the

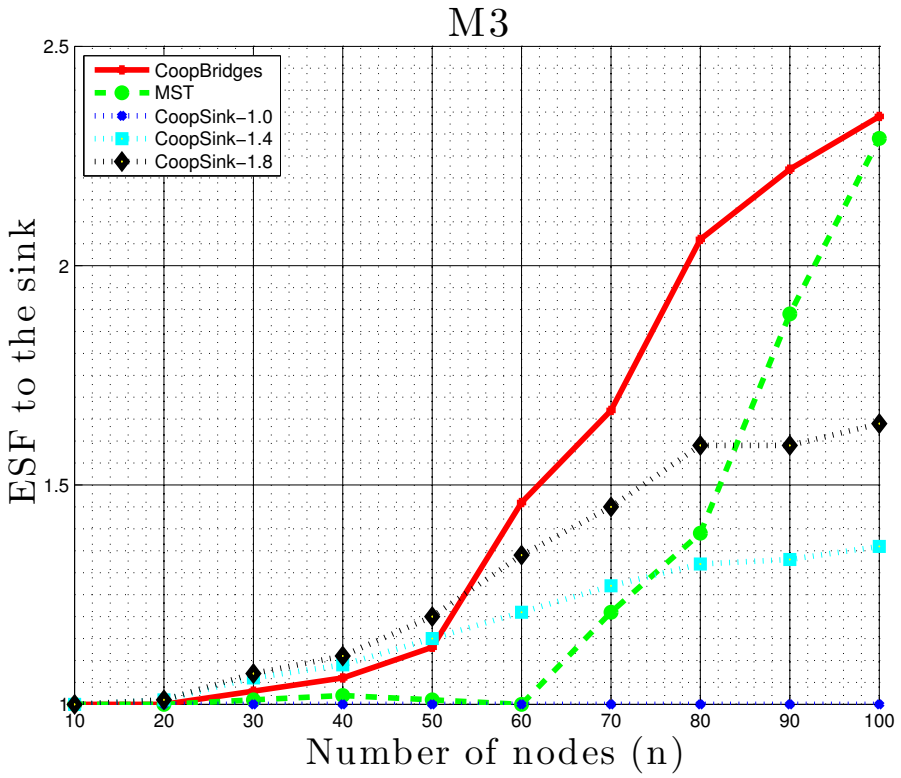


Fig. 6. Simulation results for M3.

ESF difference is, the less efficient the routes are. From the above, it can be verified that CoopSink presented routes up to 2.34 times better with relation to other proposals in the scenario considered in this work;

- Note that, for smaller values to constant t, we have very efficient routes in CoopSink at the price of a greater energy consumption, as can be observed in the results for metric M2. Clearly, striking a sensitive balance between these conflicting parameters is not an easy task and may depend on the application. In high throughput environments without severe energy restrictions, it would be desirable to keep $t \approx 1$, while in environments with energy restrictions, the t values should be higher.

5.4 M4: Number of CC edges

Figure 7 presents the simulation results for metric M4. The x-axis presents the number of network nodes while the y-axis shows the number of CC edges in the resulting topology. It should be noted that:

- CoopSink has more CC edges when the values for constant t are smaller. This happens because, with smaller values for t, greater is the restriction to remove edges in the CoopSink algorithm. These results are consistent with the results from metric M2, once the energy consumption for CoopSink-1.0 was substantially greater that CoopSink-1.4 and CoopSink-1.8;
- With less than 60 nodes, the CoopBridges has less CC edges than CoopSink-1.4 and CoopSink-1.8, and this number increases and surpass CoopSink later. This

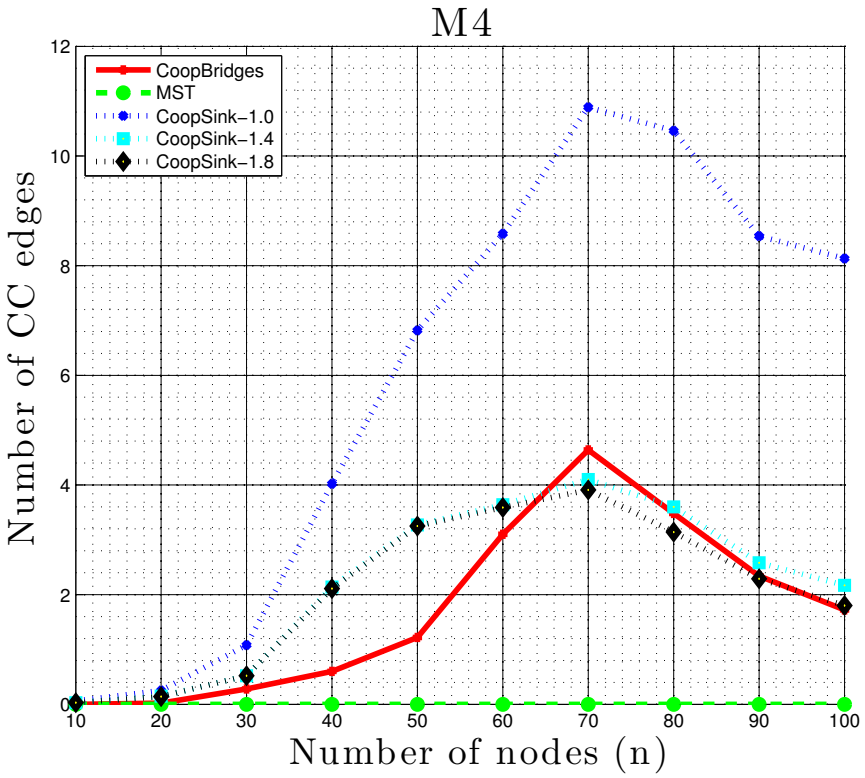


Fig. 7. Simulation results for M4.

happens because the connectivity to the sink in CoopBridges improves with $n \geq 50$ nodes, explaining this increase in the number of CC edges;

- From 60 nodes, the CoopBridges remains with basically the same number of CC edges than CoopSink-1.4 and CoopSink-1.8. However, the CoopBridges still has less connectivity to the sink. This shows that CoopSink is able explore CC links more efficiently than CoopBridges when the connectivity to the sink is considered.

6 Conclusion

In this work, it was presented a new technique, named CoopSink, that can be used to perform topology control in cooperative ad hoc networks. More specifically, CoopSink is a technique developed for ad hoc networks where there is the necessity to create efficient routes to a sink node. With this purpose, CC is used to increase connectivity and reduce energy consumption while maintaining efficient routes to the sink. When compared to other similar proposals, CoopSink was able to improve network connectivity to the sink up to 6.8 times while routes are up to 2.3 times better when the ESF to the sink is considered. Future works aim to analyse the impact of the required node synchronisation that is necessary for CC communication.

References

[1] Agarwal, M., L. Gao, J. H. Cho and J. Wu, *Energy efficient broadcast in wireless ad hoc networks with hitch-hiking*, Mobile Networks and Applications **10** (2005), pp. 897–910.

[2] Basagni, S., M. Conti, S. Giordano and I. Stojmenovic, "Mobile ad hoc networking," Wiley.com, 2004.

[3] Blough, D. M., M. Leoncini, G. Resta and P. Santi, *On the symmetric range assignment problem in wireless ad hoc networks*, in: *Proceedings of the IFIP 17th World Computer Congress-TC1 Stream/2nd IFIP International Conference on Theoretical Computer Science: Foundations of Information Technology in the Era of Networking and Mobile Computing*, 2002, pp. 71–82.

[4] Boukerche, A., H. Oliveira, E. F. Nakamura and A. A. F. Loureiro, *A novel location-free greedy forward algorithm for wireless sensor networks*, in: *Communications, 2008. ICC'08. IEEE International Conference on*, IEEE, 2008, pp. 2096–2101.

[5] Cardei, M., J. Wu and S. Yang, *Topology control in ad hoc wireless networks using cooperative communication*, Mobile Computing, IEEE Transactions on **5** (2006), pp. 711–724.

[6] Chen, W.-T. and N.-F. Huang, *The strongly connecting problem on multihop packet radio networks*, Communications, IEEE Transactions on **37** (1989), pp. 293–295.

[7] Clementi, A. E., P. Penna and R. Silvestri, *On the power assignment problem in radio networks*, Mobile Networks and Applications **9** (2004), pp. 125–140.

[8] de Morais Cordeiro, C., H. Gossain and D. P. Agrawal, *Multicast over wireless mobile ad hoc networks: present and future directions*, Network, IEEE **17** (2003), pp. 52–59.

[9] Ekbatanifard, G. and R. Monsefi, *Mamac: A multi-channel asynchronous mac protocol for wireless sensor networks*, in: *Broadband and Wireless Computing, Communication and Applications (BWCCA), 2011 International Conference on*, IEEE, 2011, pp. 91–98.

[10] Grant, M., S. Boyd and Y. Ye, *Cvx: Matlab software for disciplined convex programming* (2008).

[11] Hou, T.-C. and V. Li, *Transmission range control in multihop packet radio networks*, Communications, IEEE Transactions on **34** (1986), pp. 38–44.

[12] Ibrahim, A. S., A. K. Sadek, W. Su and K. R. Liu, *Cooperative communications with relay-selection: when to cooperate and whom to cooperate with?*, Wireless Communications, IEEE Transactions on **7** (2008), pp. 2814–2827.

[13] Jakllari, G., S. V. Krishnamurthy, M. Faloutsos, P. V. Krishnamurthy and O. Ercetin, *A framework for distributed spatio-temporal communications in mobile ad hoc networks*, in: *Proc. IEEE Infocom*, 2006, pp. 1–13.

[14] Javali, N., "Topology Control for Wireless Ad-hoc Networks," ProQuest, 2008.

[15] Jia, X., D. Li and D. Du, *QoS topology control in ad hoc wireless networks*, in: *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, 1 **2**, IEEE, 2004, pp. 1264–1272.

[16] Kruskal, J. B., *On the shortest spanning subtree of a graph and the traveling salesman problem*, Proceedings of the American Mathematical society **7** (1956), pp. 48–50.

[17] Kurth, M., A. Zubow and J.-P. Redlich, *Cooperative opportunistic routing using transmit diversity in wireless mesh networks*, in: *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, IEEE, 2008, pp. 1310–1318.

[18] Laneman, J. N., *Cooperative communications in mobile ad hoc networks*, IEEE Signal Processing Magazine **1053** (2006).

[19] Laneman, J. N., D. N. Tse and G. W. Wornell, *Cooperative diversity in wireless networks: Efficient protocols and outage behavior*, Information Theory, IEEE Transactions on **50** (2004), pp. 3062–3080.

[20] Li, L., J. Y. Halpern, P. Bahl, Y.-M. Wang and R. Wattenhofer, *Analysis of a cone-based distributed topology control algorithm for wireless multi-hop networks*, in: *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, ACM, 2001, pp. 264–273.

[21] Li, N., J. C. Hou and L. Sha, *Design and analysis of an mst-based topology control algorithm*, in: *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, 1 **3**, IEEE, 2003, pp. 1702–1712.

[22] Li, X.-Y., *Topology control in wireless ad hoc networks*, Mobile Ad Hoc Networking (2003), pp. 175–204.

[23] Li, X.-Y., P.-J. Wan and Y. Wang, *Power efficient and sparse spanner for wireless ad hoc networks*, in: *Computer Communications and Networks, 2001. Proceedings. Tenth International Conference on*, IEEE, 2001, pp. 564–567.

[24] Lima, M. M., H. A. de Oliveira, E. F. Nakamura, A. A. Loureiro and B. H.-M. Gerais-Brasil, *Roteamento e agregaç ao de dados baseado no rssi em redes de sensores sem fio*, in: *Simposio Brasileiro de Redes de Computadores, 2013. SBRC*, SBC, 2013.

[25] Mohapatra, P. and S. V. Krishnamurthy, "Ad Hoc Networks - Technologies and Protocols," Springer Sciente + Business Media, Inc. chapters 1, 3, 6, 2005.

[26] Nosratinia, A., T. E. Hunter and A. Hedayat, *Cooperative communication in wireless networks*, Communications Magazine, IEEE **42** (2004), pp. 74–80.

[27] Ramanathan, R. and R. Rosales-Hain, *Topology control of multihop wireless networks using transmit power adjustment*, in: *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, 1 **2**, IEEE, 2000, pp. 404–413.

[28] Sadek, A. K., W. Su and K. R. Liu, *Multinode cooperative communications in wireless networks*, Signal Processing, IEEE Transactions on **55** (2007), pp. 341–355.

[29] Shi, Y., S. Sharma, Y. T. Hou and S. Kompella, *Optimal relay assignment for cooperative communications*, in: *Proceedings of the 9th ACM international symposium on Mobile ad hoc networking and computing*, ACM, 2008, pp. 3–12.

[30] Tang, L., Y. Sun, O. Gurewitz and D. B. Johnson, *Pw-mac: An energy-efficient predictive-wakeup mac protocol for wireless sensor networks*, in: *INFOCOM, 2011 Proceedings IEEE*, IEEE, 2011, pp. 1305–1313.

[31] Wang, L., B. Liu, D. Goeckel, D. Towsley and C. Westphal, *Connectivity in cooperative wireless ad hoc networks*, in: *Proceedings of the 9th ACM international symposium on Mobile ad hoc networking and computing*, ACM, 2008, pp. 121–130.

[32] Wu, J., M. Cardei, F. Dai and S. Yang, *Extended dominating set and its applications in ad hoc networks using cooperative communication*, Parallel and Distributed Systems, IEEE Transactions on **17** (2006), pp. 851–864.

[33] Yu, J., H. Roh, W. Lee, S. Pack and D.-Z. Du, *Cooperative bridges: topology control in cooperative wireless ad hoc networks*, in: *INFOCOM, 2010 Proceedings IEEE*, IEEE, 2010, pp. 1–9.

[34] Zhang, J., G. Zhou, C. Huang, S. Son and J. Stankovic, *Tmmac: An energy efficient multi-channel mac protocol for ad hoc networks*, in: *Communications, 2007. ICC'07. IEEE International Conference on*, IEEE, 2007, pp. 3554–3561.

[35] Zhu, Y., M. Huang, S. Chen and Y. Wang, *Energy-efficient topology control in cooperative ad hoc networks*, Parallel and Distributed Systems, IEEE Transactions on **23** (2012), pp. 1480–1491.

On Vertex Cover with Fractional Fan-Out Bound

Satoshi Fujita
Department of Information Engineering, Hiroshima University
Higashi-Hiroshima, 739-8527, Japan

Abstract*—In this paper, we consider a new variant of the minimum weight vertex cover problem (MWVC) in which** *each vertex can cover a fractional amount of edges incident on it. For example, if the degree of a vertex is five and the designated fraction is $2/3$, then it can cover at most $\lceil (2/3) \times 5 \rceil = 4$ edges among five incident edges. This problem is motivated by a sustainable monitoring of the environment by a set of agents placed at the vertices of graph G so that the failure of agents can be easily recovered by its nearby agents within a short time. This paper investigates the computational complexity of this optimization problem. More specifically, we show that the number of vertices of odd degree, denoted as n_o, plays a key role in determining the hardness of the problem, so that when the given fraction is $1/2$, the complexity of the problem increases as n_o increases, i.e., it can be solved in polynomial time when $n_o = \mathcal{O}(1)$, although it cannot be approximated within an arbitrary constant factor when $n_o = n$, where n is the total number of vertices in the given graph.**

***Index Terms*—Minimum weight vertex cover problem, computational complexity, APX-hardness.**

I. Introduction

Let $G = (V, E)$ be an undirected graph with vertex set V and edge set E, and let w be a weight function from V to $\mathbb{R}^+$. In the following, we call $w(u)$ the weight of vertex u. A **vertex cover** of G is a subset of V such that any edge in E has at least one end-vertex in the subset. Minimum weight vertex cover problem (MWVC, for short) is the problem of finding a vertex cover with minimum weight. In the following, we refer to MWVC with $w(u) = 1$ for all u's as MVC, for brevity.

A. Related Work

The computational complexity of MWVC and MVC has been extensively investigated during past decades. MVC is NP-hard even for planar graphs [8], while it is polynomially solvable for bipartite graphs, chordal graphs, graphs with bounded treewidth, and others [3]. It has a simple 2-approximation algorithm based on the maximal matching, while it is hard to approximate within an arbitrary constant factor unless $\mathcal{P} = \mathcal{NP}$, i.e., APX-hard[1] [11].

Several variants of MVC have also been investigated in the literature. Minimum connected vertex cover problem is the problem of finding a minimum vertex cover which induces a single connected component of G. This problem is known to be NP-hard even for planar graphs of maximum degree 4 [7], which was later refined to be planar *bipartite* graphs with maximum degree 4 [5], while it is polynomially solvable when the degree of the input graph is bounded by 3 [12]. As for the approximability of the problem, it is known that the minimum connected vertex cover problem is 2-approximable [2]. Capacitated vertex cover problem (CVC, for short) is a variant of MWVC in which each vertex is given a capacity and the number of incident edges covered by a vertex is bounded by the capacity of the vertex [9]. It is known that CVC is 2-approximable, and several fixed-parameter algorithms have been proposed for CVC and its variants [10]. Maximum partial vertex cover problem is the problem of, given two integers $k \geq 0$ and $t \geq 0$, determining whether there exists a vertex subset $U \subseteq V$ of size at most k such that U covers at least t edges in G [4]. It admits a 2-approximation similar to other variants of MVC [4], [6] and the connection to fixed-parameter algorithms is deeply investigated in [10].

B. Our Contribution

In this paper, we consider a new variant of MWVC in which *each vertex can cover a fractional amount of edges incident on it*. The problem we will consider in this paper is formally stated as follows. Let ρ be a real in $(0, 1]$. A **vertex cover with fan-out bound** ρ is a vertex subset $U(\subseteq V)$ and a function f: $E \to U$ such that:

This paper was supported by the Grant-in-Aid for Scientific Research No.24500082 of the JSPS.

[1]A problem is said to be APX-hard if there is a PTAS reduction from every problem in APX to that problem, where APX is a subclass of $\mathcal{NP}$ problems which admit constant-factor approximation algorithms. It is known that any APX-hard problem does not admit an approximation scheme with arbitrarily small approximation factor, unless $\mathcal{P} = \mathcal{NP}$.

- $f(e) = u$ implies $u \in e$, i.e., every edge must be covered by a vertex in U incident on it, and
- for all $u \in U$, the number of edges assigned to u must not exceed $\lceil \rho \times d(u) \rceil$, where $d(u)$ denotes the degree of u in G.

MWVC with fan-out bound ρ, abbreviated as ρ-MWVC hereafter, is the problem of finding a minimum weight vertex cover under fan-out bound ρ. Note that this is a generalization of MWVC since "$\rho = 1$" corresponds to the ordinary MWVC and this is a special case of CVC so that the capacity of the vertices is controlled by a single parameter ρ.

A motivation of introducing fan-out bound to MWVC is to realize a *sustainable monitoring of the environment* by several agents. In an ordinary setting, given a network modeled by graph G, the agent placed at a vertex is expected to cover all edges incident on it (e.g., hallways in a musium). In addition, to reduce the cost of the monitoring as much as possible, the number of agents to be deployed must be minimized, which is attained by solving MWVC for graph G (although it is NP-hard). In other words, it is commonly requested that agents are placed at the vertices so that *the number of edges covered by two agents is as small as possible* (if every edge is covered by exactly one end-vertex, then it naturally derives a *minimal* vertex cover of G). However, such a rigid assignment is not robust against failures. In fact, if an agent crushes, we need to reconfigure a large portion of the assignment of agents, and in many cases, we need to deploy a new agent as a substitute of the crushed one which generally takes (at least) few hours before completing the recovery. On the other hand, if the fan-out of each vertex u is bounded by $\lceil \rho \times d(u) \rceil$ for an appropriate $\rho < 1$, the role of a crushed agent can be efficiently taken over by its nearby agents, which significantly reduces the recovery time. In addition, by bounding the number of edges actually monitored by each agent by a certain value, we can reduce the load of each agent, while it increases the number of agents necessary to cover all edges. As such, parameter ρ used in the definition of the problem effectively controls the residual availability of each agent, which strongly motivates us to investigate the property of the new problem, including the computational complexity and the existence of polynomial time algorithms for special cases (note that since ρ-MWVC is at least as hard as MWVC, we need to make some restrictions on the problem to derive positive results).

Main results derived in this paper are summarized as follows: 1) $(1/2)$-MWVC is polynomially solvable if the number of vertices of odd degree is bounded by $\mathcal{O}(1)$; and 2) $\left(\frac{r}{2r-1}\right)$-MWVC for $(2r-1)$-regular graphs[2] is APX-hard for any fixed $r \geq 3$. Since

$$\left\lceil \frac{r}{2r-1} \times (2r-1) \right\rceil = \left\lceil \frac{1}{2} \times (2r-1) \right\rceil = r$$

holds, $\left(\frac{r}{2r-1}\right)$-MWVC is equivalent to $(1/2)$-MWVC for $(2r-1)$-regular graphs. Thus the above results indicate that when $\rho = 1/2$, the complexity of ρ-MWVC strongly depends on the number of vertices of odd degree, namely, although the problem is polynomially solvable if the number of such vertices is small, it is hard to approximate within an arbitrary factor if the number of such vertices is n even if the underlying graph is restricted to be regular.

The remainder of this paper is organized as follows. Section II describes elementary results. Section III describes positive results including a polynomial time algorithm for a subclass of instances. Section IV gives a proof of the APX-hardness. Finally, Section V concludes the paper with future work.

II. Elementary Results

Since any subset $U \subseteq V$ can cover at most $\sum_{u \in V} \lceil \rho \times d(u) \rceil$ edges under fan-out bound ρ, parameter ρ should satisfy the following inequality:

$$\sum_{u \in V} \lceil \rho \times d(u) \rceil \ \leq \ |E|. \tag{1}$$

In the following, we assume $\rho \geq 1/2$, without loss of generality. The reader should note that although "to be $\rho \geq 1/2$" is a sufficient condition to have a feasible solution (see Lemma 1 below), this is not necessary in general, since if the given graph is 3-regular, any $\rho > 1/3$ admits a feasible solution.

Lemma 1: For any G, $\rho \geq 1/2$ is a sufficient condition for the existence of a feasible solution to ρ-MWVC.

Proof: It is enough to show that there is a feasible solution when $\rho = 1/2$ and $U = V$. If every vertex in V has even degree, we can calculate a feasible solution in the following manner: 1) identify a cycle C in G and fix the orientation of edges in the cycle so that it forms a directed cycle; 2) for each edge $e = \{u, v\}$ in C, if it is oriented from u to v, then assign e to u; 3) remove all edges in C from G to have a new graph G'. By construction, a feasible solution for G can be obtained from a feasible solution for G' and the assignment given to C (recall that we are assuming $\rho = 1/2$). Hence by repeating similar steps until the resulting graph becomes

[2]Graph G is said to be r-regular (or simply regular) if all vertices have the same number of adjacent vertices.

empty (note that such a recursion always terminates because we are assuming that every vertex in G has even degree), we have a feasible solution for G.

If G contains a vertex of odd degree, on the other hand, we can extend the above argument in the following manner. Let W be the set of vertices with odd degree. Note that $|W|$ must be even since $\sum_{u \in V} d(u) = 2|E|$. By connecting $|W|/2$ arbitrary pairs of the vertices in W by dummy edges, we have a super-graph G'' of G such that all vertices have even degree. Thus, after constructing a vertex cover of G'' with $\rho = 1/2$ using the above argument, we can obtain a solution for G by simply omitting dummy edges in G''. Hence the lemma follows. ■

Corollary 1: If every vertex in G has even degree, $(1/2)$-MWVC of G can be calculated in linear time.

With the above notions, we can derive a naive approximation scheme for solving ρ-MVC as follows. Let Δ and Δ^* denote the maximum and average degree of graph G, respectively. By assumption, each vertex in G can cover at most $\lceil \rho\Delta \rceil \leq \rho\Delta + 1$ incident edges. Thus, to cover all of $|E|$ edges, any feasible solution must contain at least

$$\frac{|E|}{\rho\Delta + 1}$$

vertices. As Lemma 1 claims, when $\rho \geq 1/2$, $U = V$ is a feasible solution to the problem. Since $|V|$ can be represented as $|V| = 2|E|/\Delta^*$, the approximation ratio of such a naive scheme is at most

$$\frac{|V|}{|E|/(\rho\Delta + 1)} = \frac{2(\rho\Delta + 1)}{\Delta^*}.$$

Thus we have the following proposition.

Proposition 1: For any $\rho \geq 1/2$, there is an approximation scheme for solving ρ-MVC with approximation ratio

$$\frac{2(\rho\Delta + 1)}{\Delta^*}$$

where Δ and Δ^* are the maximum and average degree of G, respectively.

Corollary 2: If the given G is r-regular, the approximation ratio of the above naive scheme is $2\rho + 2/r$.

Thus for example, when $r = 5$ and $\rho = 2/3$, the approximation ratio of the scheme is calculated as $2 \times 2/3 + 2/5 \leq 1.734$.

III. Positive Results

As Lemma 1 claims, if $\rho = 1/2$ and G contains no vertex of odd degree, then $U = V$ is the unique solution to the problem (although there might exist exponential number of candidates for function f). However, if G contains vertices of odd degree, we could "reduce" the size of U from V by carefully assigning edges to the vertices. This section first describes a heuristic scheme to calculate such a solution. The reader should note that although we could bound the approximation ratio of the resulting scheme by a certain value as in Proposition 1, as will be proved in the next section, the approximation ratio could not be arbitrarily small, unless $\mathcal{P} = \mathcal{NP}$.

Let ϕ denote a $(1/2)$-MWVC with $U = V$ obtained by applying the procedure described in the proof of Lemma 1. In the following, to make the exposition clear, we identify function ϕ with the orientation of edges satisfying the constraint (see the proof of Lemma 1 as for the meaning of the orientation of edges). Given such an initial configuration, we can reduce the weight of U in the following steps. Recall that by construction, under the assignment ϕ, each vertex u has at least $\lfloor d(u)/2 \rfloor$ "incoming" edges and at most $\lceil d(u)/2 \rceil$ "outgoing" edges. Let $a(u)$ be a variable representing the number of incident edges of u which can be changed from an incoming edge to an outgoing edge without violating the constraint on the fan-out bound ρ for u. Since u can have at most $\lceil \rho \times d(u) \rceil$ outgoing edges, under ϕ, $a(u)$ is initialized to $\lceil \rho \times d(u) \rceil - \lceil d(u)/2 \rceil$ or $\lceil \rho \times d(u) \rceil - \lfloor d(u)/2 \rfloor$ for each u (note that the above value is not negative since $\rho \geq 1/2$).

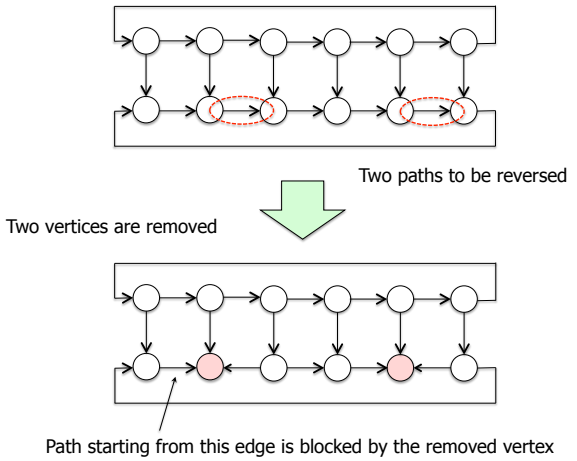
Let u^* be a vertex in U. Vertex u^* can be "removed" from U by changing all of its outgoing edges to incoming edges. More specifically, an outgoing edge of u^*, say e^*, can be changed to be an incoming edge by conducting the following steps:

1) identify a directed path starting from e^* which ends up with a vertex v with $a(v) > 0$,
2) change the direction of the path to be "from v to u^*," and
3) decrement $a(v)$ by one and increment $a(u^*)$ by one.

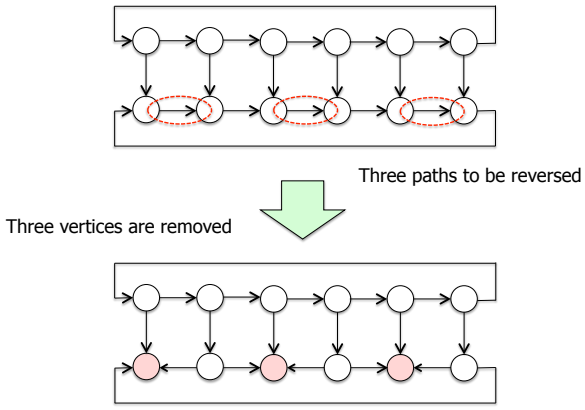
Note that such a change of the direction of a path does not change the value of $a(\cdot)$ of non-terminal vertices on the path; i.e., it works as an alternating path used in max-flow algorithms.

A possible heuristic to find a small U is to repeat the above "path reversal" operation until no additional removal can be applied. Unfortunately, such a reversible path does not always exist in G even if there remains a vertex $v \in U$ with $a(v) > 0$; i.e., the goodness of the solution depends on the order of removals of the vertices from U. In fact, there is an instance such that we can remove three vertices from U by removing vertices in an appropriate order, but by using a wrong order of removals, we can not remove more than two vertices (see Figure 1 for illustration).

As will be shown in the next section, the problem



(a)



(b)

Fig. 1. Counter-example for the naive path reversal scheme.

of finding a solution with minimum weight is in fact NP-hard and difficult to approximate within an arbitrary approximation ratio. However, by setting an appropriate constraint on the set of instances, we can calculate an optimum solution in polynomial time as the following theorem claims.

Theorem 1: If the number of odd vertices in G is $\mathcal{O}(1)$, then $(1/2)$-MWVC can be solved in polynomial time.

Proof: Let $W(\subseteq V)$ denote the subset of vertices of odd degree. Since any vertex of even degree does not have more outgoing edges than incoming edges under any configuration for $(1/2)$-MWVC, the total number of paths contributing to the removal of vertices from $U = V$ is at most $|W|$. In other words, if we assume that the set of vertices contributing to the removal of vertices is $Y(\subseteq W)$ and G has a sufficiently large connectivity to

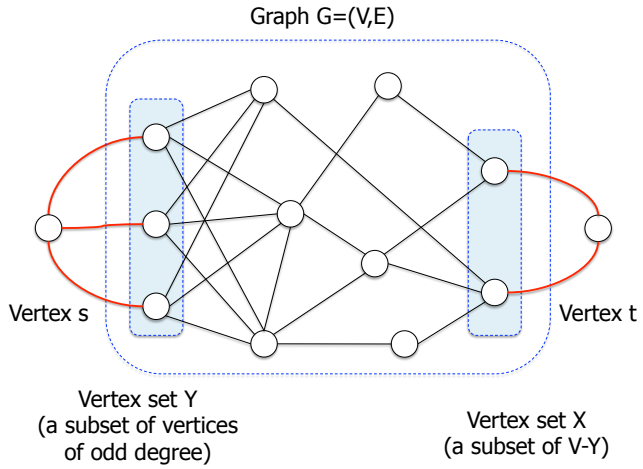


Fig. 2. Graph G' in the proof of theorem 1 (the capacity of red edges is infinity and the capacity of the other edges is one).

allow edge-disjoint paths connecting Y and the set of removed vertices, say X, in G so that all edges incident on X are "fully" used, then we can maximize the weight of removed vertices by solving a Knapsack problem in which:

1) each item corresponding to a vertex in $V - Y$ is assigned cost (i.e., $d(\cdot)$ in our terminology) and value ($w(\cdot)$ in our terminology), and
2) the total value of selected items is maximized subject to the total cost of selected items is bounded by $|Y|$.

Thus by examining the feasibility of each subset of $V-Y$ as a set of removed vertices and by taking the maximum of the resulting value over all feasible subsets, we have a maximum, feasible subset of vertices which can be removed from G with the aid of subset Y, where the feasibility check for subset $X(\subseteq V - Y)$ proceeds as follows:

- Attach vertices s and t to G so that s connects to all vertices in Y by links of infinite capacity and t connects to all vertices in X by links of infinite capacity (see Figure 2 for illustration);
- Calculate the maximum flow from s to t by assuming that each edge in G has unit capacity; and
- If the size of the maximum flow is smaller than $\sum_{u \in X} d(u)$, then X is not feasible, otherwise, it is feasible, i.e., there exists a set of edge-disjoint paths from Y to X so that all edges incident on X are incoming edges.

The number of subsets of $V - Y$ to be examined in the above procedure is polynomial since the size of each subset is assumed to be constant (i.e., there are at most $n^{\mathcal{O}(1)}$ such subsets), and for each subset, we can

calculate the feasibility of the subset in polynomial time by using max-flow algorithm. Thus an optimum solution with respect to $Y(\subseteq W)$ is calculated in polynomial time. In addition, since the number of subsets of W is polynomial, the total running time of the overall scheme is polynomial. Hence the theorem follows. ■

IV. APX-Hardness

This section proves the APX-hardness of ρ-MWVC. More precisely, we prove the following theorem.

Theorem 2: For any $\rho = \frac{r}{2r-1}$ with fixed integer $r \geq 3$, ρ-MWVC for $(2r-1)$-regular graphs is APX-hard.

The proof of the theorem is based on an L-reduction from MVC for cubic graphs which is known to be APX-complete [11]. Given two NP optimization problems $\mathcal{F}$ and $\mathcal{G}$ and a polynomial time transformation ξ from instances of $\mathcal{F}$ to instances of $\mathcal{G}$, we say that ξ is an **L-reduction** from $\mathcal{F}$ to $\mathcal{G}$ if there are positive constants α and β such that the following two conditions hold for every instance x of $\mathcal{F}$ [11].

1) Optimum solution of $\xi(x)$ with respect to problem $\mathcal{G}$, denoted by $opt_{\mathcal{G}}(\xi(x))$, is at most α times of the optimum solution of x with respect to problem $\mathcal{F}$, denoted by $opt_{\mathcal{F}}(x)$.
2) For every feasible solution y of $\xi(x)$ with objective value c_2, we can in polynomial time find a solution y' of x with objective value c_1 such that $|opt_{\mathcal{F}}(x) - c_1| \leq \beta |opt_{\mathcal{G}}(\xi(x)) - c_2|$.

It is known that if $\mathcal{F}$ is APX-hard and there is an L-reduction from $\mathcal{F}$ to $\mathcal{G}$, then $\mathcal{G}$ is also APX-hard [11]. Our proof consists of two steps. The first step is a reduction from MVC for cubic graphs to MVC for r-regular graphs (Section IV-A), and the second step is a reduction from MVS for r-regular graphs to $(\frac{r}{2r-1})$-MWVC for $(2r-1)$-regular graphs (Section IV-B).

A. First Step

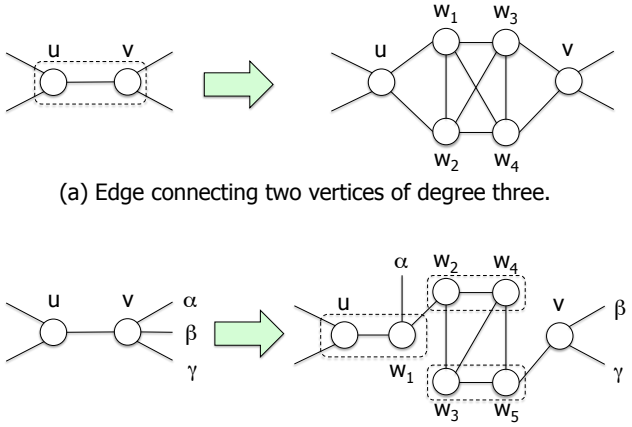
This subsection proves the following theorem.

Theorem 3: MVC for r-regular graphs is APX-hard for any fixed $r \geq 3$.

This theorem is an immediate consequence of the following two lemmas.

Lemma 2: MVC for r-regular graphs is APX-hard, for $r = 3$ and 4.

Proof: Since APX-hardness for $r = 3$ is proved in [1], it is enough to prove the claim for $r = 4$. Let $G = (V, E)$ be a cubic graph, where we assume G is connected, without loss of generality. Before proceeding to the detailed description of the proof, we introduce two notions which play a key role in the proposed reduction. A **perfect matching** of an n-set S is a set of $\lfloor n/2 \rfloor$ disjoint 2-sets drawn from S. Given two edge-disjoint paths p_1 and p_2 sharing a vertex in cubic graph G, we say that p_1 **dominates** p_2 if: 1) p_1 passes through a terminal vertex of p_2 and 2) p_2 does not pass through a terminal vertex of p_1. Note that any two edge-disjoint paths p_1 and p_2 which share two vertices in G but are not sharing their end vertices can be transformed into two edge-disjoint paths p_1' and p_2' such that p_1' dominates p_2', while keeping the set of edges used in these paths.



(a) Edge connecting two vertices of degree three.

(b) Edge connecting vertices of degrees three and four.

Fig. 3. Two transformations used in Lemma 2.

At first, we calculate a perfect matching M of V satisfying the following conditions, which will be referred to as Condition DOM hereafter:

1) pairs in M are connected by a set P of edge-disjoint paths in G; namely, so that every edge in G is used at most once in P;
2) if two paths p_1 and p_2 in P share a vertex, then either p_1 dominates p_2 or p_2 dominates p_1; and
3) the domination relation between paths in P is a partial order on P.

Since $|V|$ is even, V has a perfect matching of cardinality $|V|/2$. In addition, we can find a perfect matching satisfying Condition DOM in polynomial time by repeating local modification starting from arbitrary perfect matching of V.

For each $\{u, v\} \in M$, let $p(u, v) \in P$ denote the path connecting u and v. If $p(u, v)$ consists of one edge, we can increase the degree of u and v by one, by applying the transformation shown in Figure 3 (a). For the other pairs of vertices, we conduct the following operation in an order such that *the processing for a path $p(u, v)$ can start only after all paths dominated by p have been processed*. The reader should note that under such an ordering, we have a situation such that for each path currently being processed, all vertices on the path, except

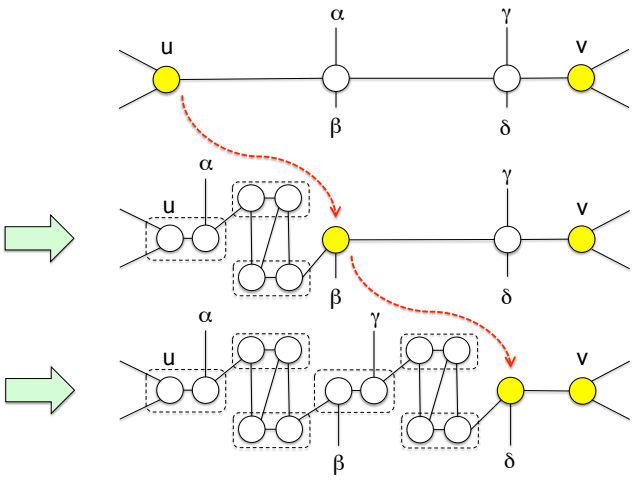


Fig. 4. The movement of a vertex of degree 3 from the position of vertex u to the position of the adjacent vertex of v.

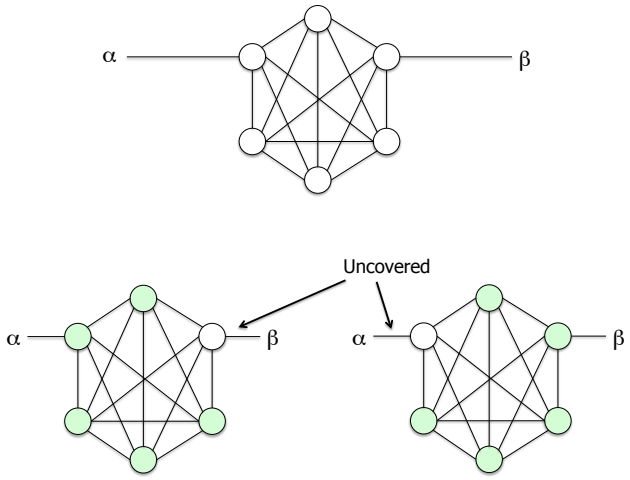


Fig. 5. Component used in the proof of Lemma 3 for $r = 5$ (two figures at the bottom represents vertex covers of the component of cardinality five).

for the terminal vertices, have degree 4. The idea of the transformation is to sequentially "move" the position of a *vertex of degree 3* from the position of u to the adjacent vertex of v on the path, by repeating the transformation Tb represented in Figure 3 (b), where pairs of vertices of degree 3 enclosed by a dashed rectangle in the figure are replaced by a component consisting of vertices of degree 4, by applying the transformation Ta shown in Figure 3 (a). After completing the movement of a vertex of degree 3 to the adjacent vertex of v, we increase the degree of those vertices by applying Ta to them. Figure 4 illustrates the movement in two hops. As such, we can always have a 4-regular graph G' from cubic graph G by applying two operations shown in Figure 3.

The fact that the above transformation from G to G' is in fact an L-reduction is verified as follows. Every application of transformation Ta increases the size of the solution by 3. Every application of transformation Tb increases the size of the solution by 3 and for each pair of the vertices enclosed by a dashed rectangle, an application of Ta increases the size of the solution by 3. Thus, the total amount of increase due to the move of a degree-3 vertex to its neighbor is $3 + 3 \times 3 = 12$. Since the set of paths P is established in an edge-disjoint manner, such a move of a degree-3 vertex occurs at most $|E| \leq 2n$ times before completing the overall transformation. On the other hand, since the maximum degree of G is 3, the size of the optimum solution for G is at least $|E|/3 \geq n/3$. Thus the optimum solution for G' is at most constant times of the optimum solution for G, i.e., the first condition of the L-reducibility holds, and simultaneously, for any vertex cover of G', we can construct a vertex cover of G satisfying the second condition. Thus the claim follows. ■

Lemma 3: MVC for r-regular graphs is APX-hard for any fixed $r \geq 5$.

Proof: Let G' be a 4-regular graph which is obtained from cubic graph G by applying the transformation described in the proof of Lemma 2. We prove the claim by providing a transformation from G' to an r-regular graph for $r \geq 5$, which is a part of the transformation of a given cubic graph G to an r-regular graph G'' (the reader should note that G' is *not* an arbitrary 4-regular graph).

At first, we show that graph G' has a perfect matching in the graph theoretical sense. Recall that the role of Tb is to transform a vertex of degree 4 into six vertices of degree 3 each, and the actual increment of the degree is attained by applying Ta to a part of the component enclosed by a dashed rectangle or two neighboring vertices of degree 3 each. In addition, for each edge in G, the transformation of the edge is conducted at most once. Hence by (imaginarily) suspending the application of Ta to the graph obtained by applying Tb to G (graph shown in Figure 3 (b) represents such a graph), we have a cubic graph to have a perfect matching consisting of edges to which Ta will be applied. Thus, by selecting edges in each copy of the component used in Ta so that $\{\{u, w_1\}, \{w_2, w_3\}, \{w_4, v\}\}$ (see Figure 3 (a) for the notation), we have a perfect matching of the resulting graph G'.

Let $\hat{E}$ be a perfect matching of 4-regular graph G'. Since the number of vertices in G' is even, $\hat{E}$ is an edge cover of G', i.e., each vertex in G' is incident on exactly one edge in $\hat{E}$. Consider the following construction of

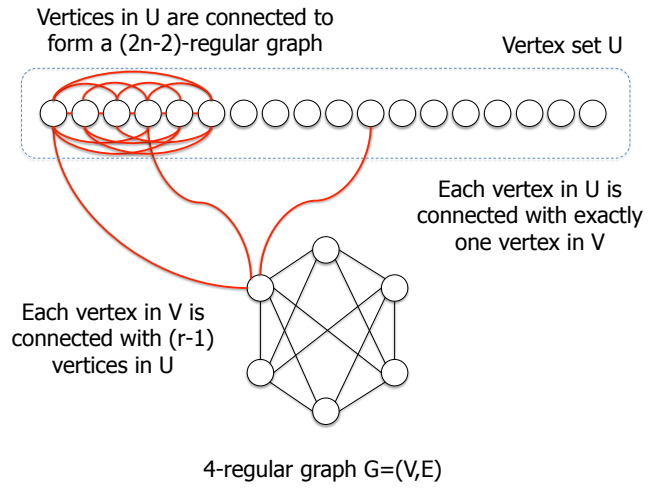


Fig. 6. Transformation used in Lemma 4 ($r = 4$).

an r-regular graph ($r \geq 5$) G'' from G':

1) Prepare $(r-3)|\hat{E}|$ copies of the component depicted in Figure 5, which is obtained by "cutting" an edge in K_{r+1} at the middle point. The reader should note that to cover all edges in the component except for one "open" edge, we must use at least r vertices and there is a subset of r vertices which cover all edges in the component except for one open edge (see two graphs in the bottom of the figure, where green vertices form a vertex cover).
2) Replace each edge $\{u, v\} \in \hat{E}$ by $r-3$ copies of the component each of which connects u and v.

By construction, the resulting graph is r-regular. In addition, if G' has a vertex cover of size c, then G'' has a vertex cover of size $c + r(r-3)|\hat{E}|$, and vise versa. Since $c \geq n/2$ and $|\hat{E}| = n/2$, by Lemma 2, it gives an L-reduction from MVC for cubic graphs to MVC for r-regular graphs. Hence the lemma follows. ■

B. Step Two

Next, we prove the following lemma, which completes the proof of Theorem 2.

Lemma 4: For any $r \geq 3$, there is an L-reduction from MVC for r-regular graphs to $(\frac{r}{2r-1})$-MWVC for $(2r-1)$-regular graphs.

Proof: Let $G = (V, E)$ be an r-regular graph consisting of n vertices. Let U be a set of $(r-1)n$ vertices such that $U \cap V = \emptyset$. We construct a $(2r-1)$-regular graph G' with vertex set $U \cup V$ by connecting those vertices in the following manner (see Figure 6 for illustration):

- Vertices in V are connected as in G, and each vertex in V is connected with $r-1$ vertices in U, so that the degree of those vertices becomes $2r-1$.
- Each vertex in U is connected with exactly one vertex in V and $2r-2$ other vertices in U so that the degree of those vertices becomes $2r-1$, in such a way that the subgraph of G' induced by U is a $(2r-2)$-regular graph. Note that such a connection is always possible for any $n \geq 3$ since it is known that there exists a k-regular graph of order ℓ iff $\ell \geq k+1$ and $k\ell$ is even.

In addition, we assign the "weight" to each vertex in G' in the following manner:

- $w(u) := 1$ for each $u \in V$; and
- $w(v) := \epsilon$ for each $v \in U$, where ϵ is a constant satisfying $\epsilon < \frac{1}{r(n-1)}$

By construction, the total weight of the vertices in U is smaller than the weight of a single vertex in V. Thus for any feasible solution f for G' (with respect to $(1/2)$-MWVC), we can have a solution f' such that: all vertices in U are selected and the difference to the optimum solution f^* increases by at most one. Thus without loss of generality, we may assume that all vertices in U are selected in every feasible solution for G'.

Given a minimum vertex cover of G of size c, we can construct a solution of $(\frac{r}{2r-1})$-MWVC for G' of size $c+1$ by including all vertices in U into the subset and by determining the assignment of edges incident on U such that: 1) each vertex has $r-1$ incoming edges and $r-1$ outgoing edge connecting to the vertices in U and 2) it has an outgoing edge connecting to a vertex in V. Note that it realizes a weighted vertex cover of G' under fractional bound $\rho = \frac{r}{2r-1}$, since each vertex in V has $r-1$ incoming edges connecting to vertices in U and at most r outgoing edges connecting to vertices in V. Thus the first condition of the L-reducibility holds. Since such a correspondence holds even for optimum solution for the problems, the second condition also holds. Thus the lemma follows. ■

V. Concluding Remarks

This paper proposes a new variant of the minimum weighted vertex cover problem with a fractional fan-out bound. It is clarified that the computational complexity of the problem strongly depends on the number of vertices of odd degree when the designated fraction is $1/2$.

Future works are listed as follows:

- To extend Theorem 1 so that polynomially solvable cases will be expanded, e.g., when the number of vertices of odd degree is $\mathcal{O}(\log n)$ or $\mathcal{O}(\log \log n)$. To this end, we need to apply the dynamic programming to the algorithm, while it seems to be complicated since we should examine the feasibility

of subset X with respect to the existence of edge-disjoint paths from Y unlike ordinary calculations which merely consider the cost bound.

- To expend Theorem 2 so that a similar claim holds for ρ-MVC. To this end, we need to develop a new technique for the reduction which does not rely on the weight of the vertices.
- To evaluate the sustainability of monitoring systems constructed based on the notion of ρ-MWVC with respect to several metrics such as the fault-tolerance and the convergence speed.

REFERENCES

[1] P. Alimonti and V. Kann. "Some APX-completeness results for cubic graphs." *Theoretical Computer Science*, 237:123–134, 2000.

[2] E. M. Arkin, M. M. Halldórsson, R. Hassin. "Approximating the tree and tour covers of a graph." *Information Processing Letters*, 47(6):275–282, 1993.

[3] A. Brandstadt, V. B. Le, and J. Spinrad. *Graph Classes: A Survey.* Society for Industrial and Applied Mathematic, Philadelphia, 1999.

[4] N. H. Bshouty and L. Burroughs. "Massaging a linear programming solution to give a 2-approximation for a generalization of the Vertex Cover problem." In *Proc. 15th Symposium on Theoretical Aspects of Computer Science (STACS)*, Lecture Notes in Computer Science, volume 1373, 1998, pages 298–308.

[5] H. Fernau and D. Manlove. "Vertex and Edge Covers with Clustering Properties: Complexity and Algorithms." *Journal of Discrete Algorithms*, 7(2):149–167, 2009.

[6] R. Gandhi, S. Khuller, and A. Srinivasan. "Approximation algorithms for partial covering problems." In *Proc. 28th International Colloquium on Automata, Languages, and Programming (ICALP)*, Lecture Notes in Computer Science, volume 2076, 2001, pages 225–236.

[7] M. R. Garey and D. S. Johnson. "The Rectilinear Steiner Tree Problem is NP-Complete." *SIAM Journal of Applied Mathematics*, 32(4):826–834, 1977.

[8] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, San Francisco, 1979.

[9] S. Guha, R. Hassin, S. Khuller and E. Or. "Capacitated vertex covering." *Journal of Algorithms,* 48:257–270, 2003.

[10] J. Guo, R. Niedermeier, S. Wernicke. "Parameterized Complexity of Generalized Vertex Cover Problems." In *Algorithms and Data Structures*, Lecture Notes in Computer Science, volume 3608, 2005, pages 36-48.

[11] C. H. Papadimitriou and M. Yannakakis. "Optimization, approximation, and complexity classes." *J. Comput. Systems Sci.*, 43:425–440, 1991.

[12] S. Ueno, Y. Kajitani, and S. Gotoh. "On the non separating independent set problem and feedback set problem for graphs with no vertex degree exceeding three." *Discrete Mathematics*, 72(1-3):355–360, 1988.

Solvability for The Maximum Legal Firing Sequence Problem of Conflict-Free Petri Nets with Inhibitor Arcs

Satoshi Taoka

Graduate School of Engineering, Hiroshima University
1-4-1, Kagamiyama, Higashi-Hiroshima, 739-8527 Japan

***Abstract*— The subject of this paper is the maximum legal firing sequence problem (MAX-INLFS) for inhibitor-arc Petri nets *IN*. It is well-known that modeling capability of inhibitor-arc Petri nets is equivalent to that of Turing machines, and MAX-INLFS has wide applications to fundamental problems of Petri net such as the marking reachability problem, the scheduling problem, and so on. It is known that, when *IN* has weighted forward conflict-free structure and has only one place (called a rivet) to which at least one inhibitor-arc is incident, MAX-INLFS can be solved in pseudo-polynomial time if weights of all edges entering the rivet are equivalent; otherwise it is NP-hard. In this paper, when *IN* has more than one rivet *rv*, we show that MAX-INLFS can be solved in in $O(2^{|RV|}|P||X|)$ time, where *RV* is a set of rivets in *IN*.**

I. Introduction

An *inhibitor arc* (or simply an *inhibitor*) is a special directed edge (p, t) of unit weight, from a place p to a transition t such that, whenever p has a token, t cannot be fired. Such a place p is called a *rivet*. An inhibitor-arc Petri net $IN = (P, T, I, E, \alpha, \beta)$ consists of a Petri net (called the *underlying* Petri net) with any set of inhibitor arcs added. In figures of this paper, any inhibitor arc is represented as a dashed line terminating with a small circle attached to a transition. It is shown in [1] (see also [2]) that modeling capability of inhibitor-arc Petri nets is equivalent to that of Turing machines since inhibitor-arc Petri nets can test "zero" (that is, whether a place has at least one token or not).

The Legal Firing Sequence problem **INLFS** of inhibitor-arc Petri nets is defined by "Given an inhibitor-arc Petri net *IN*, an initial marking M_0 and a firing count vector X, find a firing sequence, or a sequence of transitions, which is legal on M_0 with respect to X." A component $X(t)$ of X denotes the prescribed total firing number of a given transition t. Without loss of generality we assume $X(t) > 0$ for any $t \in T$. We say that a firing sequence δ is *legal* on an initial marking M_0 if and only if the first transition of the sequence is can be fired at M_0 and the rest can be fired one after another subsequently. If such δ satisfies that each transition t appears exactly $X(t)$ times in δ then we say that δ is legal on M_0 with respect to X.

Let us introduce the Maximum Legal Firing Sequence problem **MAX-INLFS** defined as follows (see Fig. 1): "Given an inhibitor-arc Petri net *IN*, an initial marking M_0 and a firing count vector X, find a firing sequence δ such that δ is legal on M_0 within X: (i) δ is legal on M_0 and $\overline{\delta} \leq X$ (meaning that $\overline{\delta}(t) \leq X(t)$ for any $t \in T$); (ii) the length $|\overline{\delta}|$ of δ is maximum among those sequences satisfying (i), where $\overline{\delta}(t)$ is the total number of occurrences of t in δ for any $t \in T$." Let **LFS** or **MAX-LFS**, respectively, denote **INLFS** or **MAX-INLFS** for the underlying Petri net N of *IN* (that is, all inhibitor arcs of *IN* are removed). **MAX-INLFS** has wide applications to fundamental problems of Petri net such as the marking reachability problem, the scheduling problem, and so on.

There are many related results for **LFS**, **MAX-LFS**, **INLFS** and **MAX-INLFS**. It is shown in [3] that **INLFS** can be solved in $O(|X|)$ time for any inhibitor-arc Petri net with unweighted state machine structure (that is, the underlying Petri net is an unweighted state machine) if *IN* has only one rivet and is *non-adjacent type* (see [3] for the definition). On the other hand, **RINLFS** (a decision problem of **INLFS**) is NP-hard even if the following condition (1) or (2) holds: (1) *IN* has unweighted state machine structure and has at least three rivets, or (2) *IN* has unweighted forward conflict-free structure and $X(t) = 1$ for any $t \in T$. Note that NP-hardness under the above condition (1) or (2) is proved when the number of rivets in *IN* is not constant. It is shown in [4] that **MAX-LFS** for a weighted conflict-free Petri net can be solved in $O(|E||X|)$. Furthermore **MAX-INLFS** can be solved in $O(|P||X|)$ time when *IN* has weighted marked graph structure (that is, the underlying Petri net is a weighted marked graph) and has only one rivet. It is shown in [5] that, when *IN* has weighted forward conflict-free structure (that is, the underlying Petri net is a weighted forward conflict-free) and has only one rivet *rv*, (1) **MAX-INLFS** can be solved in $O(|P||X|)$ time if weights of all edges $(t, rv) \in E$ are equivalent; (2) otherwise **RINLFS** is NP-hard.

In this paper, when *IN* has weighted forward conflict-free structure (that is, the underlying Petri net is a weighted forward conflict-free) and has more than one rivet *rv*, **MAX-INLFS** can be solved in $O(2^{|RV|}|P||X|)$ time, where *RV* is a set of rivets in *IN*.

II. Preliminaries

A *Petri net* is a bipartite digraph $N = (P, T, E, \alpha, \beta)$, where P is the set of *places*, T is that of *transitions* such that $P \cap T = \emptyset$, and $E = E_{pt} \cup E_{tp}$ is an edge set such that E_{pt} consists of edges from P to T with weight function $\alpha : E_{pt} \to Z^+$ (non-negative integers) and E_{tp} consists of edges from T to P with weight function $\beta : E_{tp} \to Z^+$, In all figures in this paper, edge weight one is not shown for simplicity.

We denote an inhibitor arc from $u \in P$ to $v \in T$ as $(u, v)_i$. Petri nets with inhibitor arcs are referred to as *inhibitor-arc Petri nets*, denoted as $IN = (P, T, I, E, \alpha, \beta)$, We used the notation N for an ordinary Petri net (without inhibitor arcs) and *IN* for an inhibitor-arc Petri net unless otherwise stated. Let ${}^\bullet v = \{u \in P \cup T \mid (u, v) \in E\}$ and $v^\bullet = \{u' \in P \cup T \mid (v, u') \in E\}$. Note that inhibitor arcs are ignored in these definitions. Let ${}^\circ v = \{u \in P \mid (u, v)_i \in I\}$ and $v^\circ = \{u' \in T \mid (v, u')_i \in I\}$.

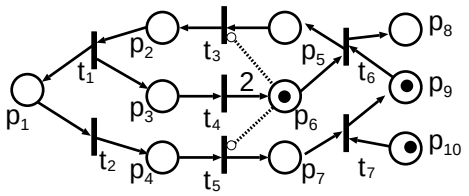


Fig. 1. An example: an inhibitor-arc Petri net IN for which an optimum solution of **MAX-INLFS** is $\delta = t_6t_3t_1t_2t_5t_7t_4t_6$ with $\overline{\delta} = [1,1,1,1,1,2,1] \leq X$.

We denote $RV = \{p \in P \mid p^\circ \neq \emptyset\}$. Let $T_s = {}^\bullet RV$ and $P' = \{rv \in RV \mid M(rv) > 0\} \neq \emptyset$.

A *marking* M for N is a function $M : P \to Z^+$, and $|M|$ denotes the total sum of $M(p)$ over all $p \in P$. A transition t of Petri net N is *enabled* at a marking M of N (denoted as $M[t\rangle$) if $M(p) \geq \alpha(p,t)$ for any $p \in {}^\bullet t$. *Firing* such t on M is to define a marking M' such that, for any $p \in P$, we have $M'(p) = M(p) + \beta(t,p)$ if $p \in t^\bullet - {}^\bullet t$, $M'(p) = M(p) - \alpha(p,t)$ if $p \in {}^\bullet t - t^\bullet$, $M'(p) = M(p) - \alpha(p,t) + \beta(t,p)$ if $p \in {}^\bullet t \cap t^\bullet$ and $M'(p) = M(p)$ otherwise. We denote as $M' = M[t\rangle$. (Hence $M[t\rangle$ denotes a marking after firing t at M and shows that t is enabled at M.) For IN, t is enabled at M if $M(p) \geq \alpha(p,t)$ for any $p \in {}^\bullet t$ and $M(q) = 0$ for any rivet q connected to t by an inhibitor arc. Let $\delta = t_{i_1} \cdots t_{i_s}$ be a sequence of transitions, and $\overline{\delta}(t)$ be the total number of occurrences of t in δ, where $T = \{t_1, \ldots, t_n\}$ and $i_j \in \{1, \ldots, n\}$. $\overline{\delta} = [\overline{\delta}(t_1) \cdots \overline{\delta}(t_n)]^{tr}$ $(n = |T|)$ is called the *firing count vector* of δ. Let $|\overline{\delta}|$ denote the sum of $\overline{\delta}(t)$ over all $t \in T$. For a marking M and an n-dimensional vector $X = [X(t_1) \cdots X(t_n)]^{tr}$, δ is called a *firing sequence* that is *legal* on M (denoted as $M[\delta\rangle$) if and only if t_{i_j} is enabled at M_{j-1} for $j = 1, \cdots, s$, where $M_0 = M$ and $M_j = M_{j-1}[t_{i_j}\rangle$. The resulting marking M_s also denotes $M[\delta\rangle$ for simplicity. Furthermore, for the markings M and M_s, and the firing sequence δ, $\langle\delta]M_s$ represents M. If $\overline{\delta} \leq X$ for such δ then we say that δ is legal on M within X. A transition t is saturated (or unsaturated) in δ if $\overline{\delta}(t) = X(t)$ (or $\overline{\delta}(t) < X(t)$). Let $\delta\delta'$ denote concatenating δ' at the rear of δ for two firing sequences δ and δ'.

A directed cycle consisting of a pair of edges (p,t) and (t,p) is called a self-loop. In this paper, we assume that no self-loop exists in N (and in IN). N is called a conflict-free Petri net if and only if (i) or (ii) holds for any $p \in P$: (i) $|p^\bullet| \leq 1$; (ii) any $t \in p^\bullet$ and p forms a self-loop. Since we assume that N has no self-loop, we consider only (i) for conflict-free Petri nets (which such a net is called a forward conflict-free Petri net). N is a *marked graph* if and only if any $p \in P$ has $|{}^\bullet p| \leq 1$ and $|p^\bullet| \leq 1$. Any marked graph is conflict-free.

III. An algorithm for MAX-INLFS

We show an algorithm *solve_INLFS_for_fcf* to solve **MAX-INLFS** when IN has weighted forward conflict-free structure (WFCF for short) structure.

An outline of the algorithm is as follows. Since $rv \in P'$ has some tokens, firing of any transition $t \in rv^\circ$ is prohibited and we consider **MAX-LFS** for N and X_v, where $X_v(t) \leftarrow 0$ for any $t \in P'^\circ \cup T_s$ and $X_v(t') \leftarrow X(t') - \overline{\delta}(t')$ for any $t' \in T - (P'^\circ \cup T_s)$. Then some rivets $rv \in P'$ may have no tokens. If such rivets exist then P' is updated and then we consier **MAX-LFS** as mentioned above again. This above operation is repeated as many as possible. Then one transition t_s, which is enabled, in T_s is selected and it fires. The above two operations are repeated as many as possible.

Now the description of the algorithm is given.

Algorithm *solve_INLFS_for_fcf*;
Input: An inhibitor-arc Petri net IN, an initial marking M_0, and a firing count vector X;
Output: A maximum firing sequence δ_m that is legal on M_0 within X;

1. $\delta_m \leftarrow$ (an empty sequence); $\delta \leftarrow$ (an empty sequence); $M \leftarrow M_0$;
2. *extend_sequence*(δ);

Procedure *extend_sequence*(δ);

1. $\delta_1 \leftarrow$ (an empty sequence); $\delta_2 \leftarrow$ (an empty sequence);
2. **while** $P' = \{rv \in RV \mid M(rv) > 0\} \neq \emptyset$ **do**
   - 2.1. Find a firing sequence δ_2 obtained by repeating firing of unsaturated enabled transitions $t \in T - (P'^\circ \cup T_s)$ beginning with a marking M as many times as possible, where $\overline{\delta\delta_1\delta_2}(t) \leq X(t)$ for any $t \in T - (P'^\circ \cup T_s)$;
   - 2.2. $\delta_1 \leftarrow \delta_1\delta_2$; $M \leftarrow M[\delta_2\rangle$; /* Since each $t_t \in P'^\bullet$ fires as many times as possible, the number of tokens in $rv \in P'$ becomes as small as possible. */
   - 2.3. If there exist rivets $rv \in P$ having no token for the current marking M is P' then break this loop; otherwise, update P'; /* this loop is repeated */
3. $T'_s = \{t \in T_s \mid \overline{\delta\delta_1}(t) < X(t), t$ is enabled$\}$;
4. **while** $T'_s \neq \emptyset$ **do**
   - 4.1. Select t_s from T'_s; $T'_s \leftarrow T'_s \setminus \{t_s\}$;
   - 4.2. $M \leftarrow M[t_s\rangle$; /* fire t_s once */
   - 4.3. *extend_sequence*(δt_s);
   - 4.4. $M \leftarrow \langle t_s]M$; /* the resulting marking is $M_0[\delta\delta_1\rangle$ */
5. If every $t \in T$ satisfies $(\overline{\delta\delta_1}(t) = X(t))$ or $(\overline{\delta\delta_1}(t) < X(t)$ and t is not enabled at $M)$ and $|\overline{\delta_m}| < |\overline{\delta\delta_1}|$ then $\delta_m \leftarrow \delta\delta_1$; /* If Step 4 executes then Step 5 does not execute */
6. $M \leftarrow \langle\delta_1]M$; /* the resulting marking is $M_0[\delta\rangle$ */ □

We will prove the next theorem.

Theorem 3.1: **MAX-INLFS** can be solved in $O(2^{|RV|}|P||X|)$ time if IN has WFCF structure. □

References

[1] J. L. Peterson: "Petri Net Theory and the Modeling of Systems", Prentice-Hall, Englewood Cliffs, N.J. (1981).

[2] T. Murata: "Petri nets: Properties, analysis and applications", Proc. IEEE, **77**, 4, pp. 541–580 (1989).

[3] S. Taoka and T. Watanabe: "Time complexity analysis of the legal firing sequence problem of petri nets with inhibitor arc", IEICE Trans. Fundamentals, **E89-A**, 11, pp. 3216–3226 (2006).

[4] S. Ochiiwa, S. Taoka and T. Watanabe: "Pseudo-polynomial time solvability for the maximum legal firing sequence problem of inhibitor-arc Petri nets –weighted marked graphs with inhibitor arcs–", Proc. ITC-CSCC 2012 (2012).

[5] S. Taoka, S. Ochiiwa and T. Watanabe: "Solvability for the maximum legal firing sequence problem of conflict-free petri nets with inhibitor arcs", Proc. ITC-CSCC 2014, pp. 861–864 (2014).

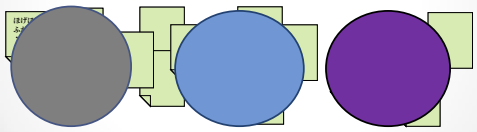
[6] M. R. Garey and D. S. Johnson: "Computers and Intractability: a Guide to the Theory of NP-Completeness", Freeman, San Francisco (1979).

ニュースストリームの動的クラスタリング

広島大学大学院
情報工学専攻
小島　寛樹

研究背景

- 一般的なクラスタリングでは文書集合は静的
  - 文書がすべて揃った状態で文書の重み付け・クラスタリングを行う



研究背景

- ストリームのクラスタリングでは文書集合は動的
  - ニュース記事は送られてきた時点でクラスタリングをしたい
  - 文書が揃っていない状態で文書の重み付け・クラスタリングを行う
- **静的なものと比べクラスタリング精度が低下**

文書が十分に揃っていない状態でも
精度を落とさないクラスタリング手法を提案

文書の重み付け

- 一般的な文書の重み付け手法：tf-idf
  - tf(term frequency): 単語の出現頻度
  - idf(Inverse document frequency): 逆文書頻度

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,i}}, idf_i = log\frac{|D|}{|\{d: d \ni t_i\}|}$$

$n_{i,j}$は文書d_jに単語t_iが出てきた回数
$|D|$は総文書数
$|\{d: d \ni t_i\}|$は単語t_iを含む文書数

- ストリームの場合ではidfは動的に変化

idfの計算方法

- 一般的なidf

$$idf_i = log\frac{|D|}{|\{d: d \ni t_i\}|}$$

$|D|$は総文書数
$|\{d: d \ni t_i\}|$は単語t_iを含む文書数

- 文書が出現した日xまでの情報で計算するidf

$$idf_{i,x} = log\frac{\sum_{a=1}^{x}|D_a|}{\sum_{a=1}^{x}|\{d_a: d_a \ni t_i\}|}$$

分子は初日からx日までの総文書数
分母は単語t_iを含む初日からx日までの文書数

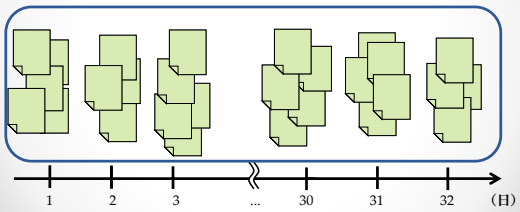
- 文書が出現した日xから過去1か月の情報で計算するidf

$$idf_{i,x} = log\frac{\sum_{a=x-30}^{x}|D_a|}{\sum_{a=x-30}^{x}|\{d_a: d_a \ni t_i\}|}$$

分子はx日から過去1ヶ月の総文書数
分母は単語t_iを含むx日から過去1ヶ月の文書数

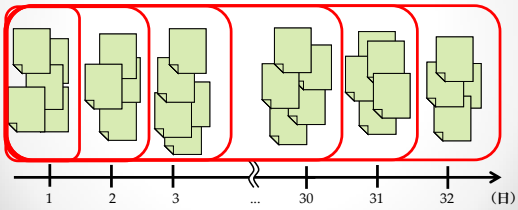
idfの計算範囲(1/3)

一般的なidfの計算範囲



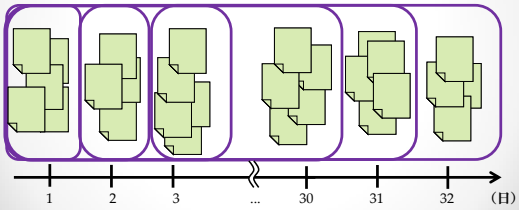
idfの計算範囲(2/3)

文書が出現した日xまでの情報で計算するidfの計算範囲



idfの計算範囲(3/3)

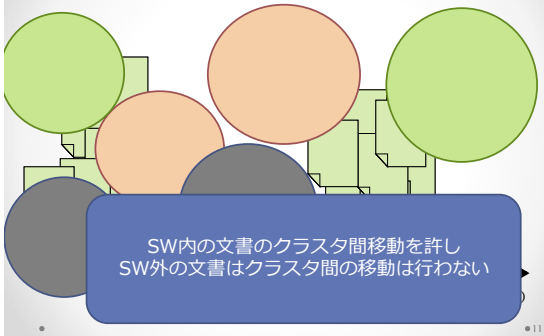
文書が出現した日xから過去1か月の情報で計算するidfの計算範囲



クラスタリング手法

- 一般的なクラスタリング手法:Kmeans法
  - Kmeans法ではクラスタ数は固定
- Kmeans法のアルゴリズム
  1. ランダムにクラスタを割り当てる
  2. 各クラスタの重心を計算
  3. 文書を最も近い重心を持つクラスタに割り当てる
  4. 割り当てに変化がなくなるまで2~3を繰り返す
- クラスタリングを行う範囲（スライディングウィンドウ:SW）を設定

クラスタリングの動き



実験

- Kmeans法でクラスタリングを行い、SW、idfの違いでの結果を比較
- 読売新聞のニュース記事
  - 2013年6月から2014年3月までの10ヶ月の記事
  - 記事数：35559
  - 特徴数：8340(MeCabを用いて形態素解析後、前処理を行った)
- SWのサイズ
  - 1週間(7日間)
  - 1ヶ月(30日間)
  - 6ヵ月(180日間)
- idfの計算方法
  - 全体で計算したもの(全体)
  - その日までの情報で計算したもの(その日まで)
  - その日から1ヶ月前までの情報で計算したもの(過去1ヶ月)

比較方法

- idfを全体で計算したもので重み付けをし、全体を見てクラスタリングをした結果とを以下の指標で違いを比較
  - purity(クラスタの純度)
  - cluster entropy(クラスタのエントロピー)
  - class entropy(クラスのエントロピー)
  - F-measure(F値)
    - エントロピーは値が小さいほど答えと近く、純度とF値は大きいほど答えと近い
- クラスタの中身を人の目で見て比較

結果(1/2)

- 全体を見てクラスタリングをした結果との比較

SW	idfの計算範囲	Purity	Cluster entropy	Class entropy	F-measure
	全体	21.4%	76.0%	78.4%	19.7%
1週間					
1ヶ月					
6ヶ月					

その日までの情報でのidfでは精度がより低下

過去1ヶ月の情報でのidfでは精度が低下しづらい

結果(2/2)

- クラスタの中身を人の目で確認

A 原子力規制委員会は、8日の新規制基準施行に伴い、電力会社が申請した原子力発電所の安全審査について…(13/07/09)

B 日本原燃は7日、青森県六ヶ所村にある使用済み核燃料再処理工場など、核燃料サイクル関連の4施設の安全審査を原子力規制委員会に… (14/01/07)

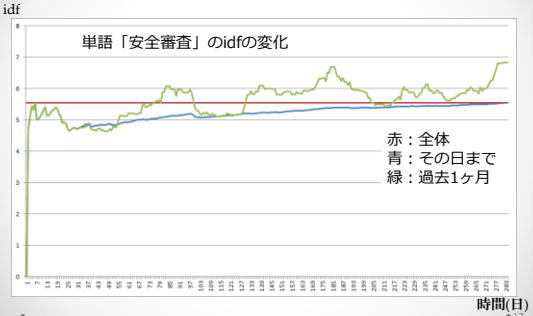
C 中部電力は6日、浜岡原子力発電所4号機（静岡県御前崎市）を再稼働させるための前提となる安全審査を…(14/02/06)

- これらの記事A,B,Cは1つのクラスタにまとめられるべき
  - SWが6ヵ月のものではまとめられている
  - SWが1ヶ月のものでは記事B,Cはまとめられている
  - SWが1週間のものではどれもまとめられていない
- 過去1ヶ月の情報でのidfを用いた場合、SWが1週間でも全てまとめられている

考察(重み付けについて)

- なぜ過去1ヶ月の情報での重み付けではクラスタリング精度が向上したのか
  - 過去1ヶ月に限定してidfを計算するということは全体をみると稀な単語でもある期間においては稀ではないこと、またその逆を反映できる
- idfの各計算方法でのある単語のidfの変化を調べ、クラスタリング結果への影響を分析

考察(重み付けについて)



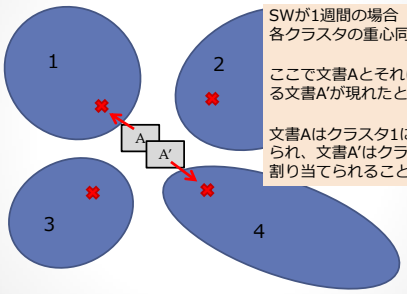
考察(クラスタリング手法について)

- SWが小さいとうまくまとめられていない
- コサイン類似度を用いて、各スライディングウィンドウでの各クラスタ間の類似度を計算
  - 類似度は0から1の範囲で表し、1が最も類似している

	1週間	1ヶ月	6ヶ月
類似度(平均)	0.355	0.117	0.074

- スライディングウィンドウが小さいほど各クラスタ同士は似ている

考察(クラスタリング手法について)

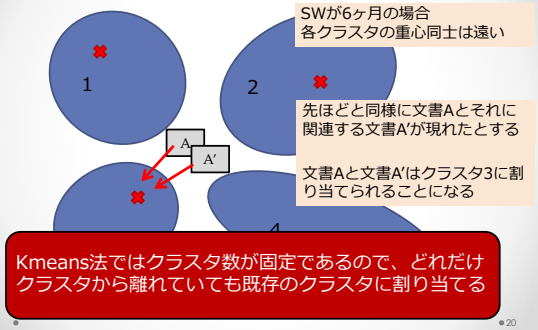


SWが1週間の場合
各クラスタの重心同士が近い

ここで文書Aとそれに関連する文書A'が現れたとする

文書Aはクラスタ1に割り当てられ、文書A'はクラスタ4に割り当てられることになる

考察(クラスタリング手法について)



SWが6ヶ月の場合
各クラスタの重心同士は遠い

先ほどと同様に文書Aとそれに関連する文書A'が現れたとする

文書Aと文書A'はクラスタ3に割り当てられることになる

Kmeans法ではクラスタ数が固定であるので、どれだけクラスタから離れていても既存のクラスタに割り当てる

まとめ

- ストリームのクラスタリングにおいて、SWが小さい場合、精度が落ちる
  - 各クラスタが近くにでき、うまく分類できない
  - トピック数が固定で無理矢理に文書を割り当ててしまう
- SWが小さい場合、文書の重み付けにおいて範囲を限定して計算を行うことで精度が落ちにくくなる
  - 単語の特徴をうまく拾い上げることができる

iBeaconモジュールを利用した2次元位置座標推定の考察

広島大学　神崎 僚太

背景

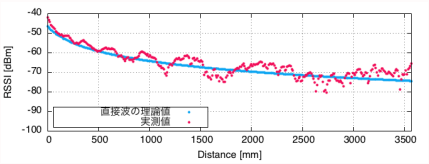
- スマートフォンなど高性能なモバイル端末の普及
- 屋外ではGPSによる位置情報を利用した様々なサービスが提供されている
  - ナビゲーション，周辺の店舗検索，位置ゲーム，...
- 屋内ではGPSが利用できないため別の手段が必要

iBeacon

- 端末とビーコンモジュールの近接を通知するフレームワーク
- Apple社が策定
- Bluetooth Low Energy (BLE) 規格を使用
  - iOS 7 以降で対応
  - Android 4.3 以降ならビーコンを受信可能
- ビーコンからはモジュールの識別情報と**受信信号強度**が取得できる

-> 複数の送信機からの受信信号強度を比較することで座標推定が行えるのではないか

受信信号強度による座標推定

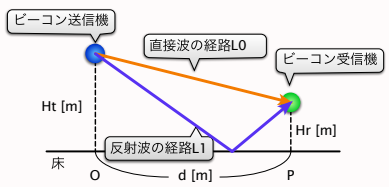


ビーコンモジュールから送信された電波は直接波だけでなく様々な経路を通って受信される

- 反射波の干渉により理論値とずれる
- 床からの反射を考慮したモデルにより補正を試みた

モデル (1/2)

- 直接波と床からの反射波のみを考慮



- $L_0 = \sqrt{(H_r - H_t)^2 + d^2}$
- $L_1 = \sqrt{(H_r + H_t)^2 + d^2}$

モデル (2/2)

- iBeacon (BLE) は2.4GHzを使用 -> 波長 0.125 m
- 反射時に位相が反転
- 位相差 $\alpha = 2\pi \frac{L_0 - L_1}{0.125} + \pi$

送信機から1mでの電波強度をTmWとすると

- 直接波の振幅 $T_0 = T/L_0$
- 反射波の振幅 $T_1 = T/L_0$

よって

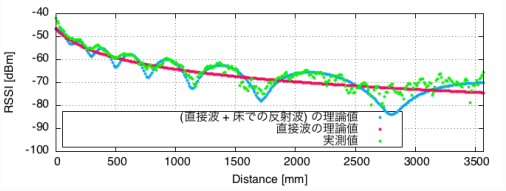
- 受信強度 $R = \sqrt{T_0^2 + T_1^2 + 2T_0T_1\cos\alpha}$

実験のセッティング

- 天井 2.6m
- 6m x 6m 程度の遮蔽物のない空間
- ビーコン送信機と受信機の高さを床から60cmで固定
- 送受信機の距離を1cmずつ変えて10秒ずつ測定,
  平均値を記録
- 送信機 Aplix MyBeacon Pro
- 受信機 Sony Xperia Z1 (Android 4.4.2)

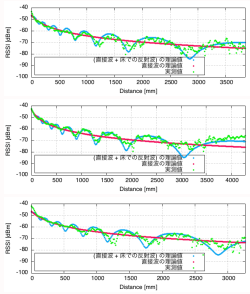
実験エリアの中心に送信機を設置し，4方向で測定

実験結果 (1/2)



- 床での反射を考慮したモデルによってより高精度で実測値を近似
  - 実測値のノイズは床からの反射による影響が支配的
- モデル上でデッドスポットとなる範囲は実測値が不安定
  - モデル外の複雑な反射波の影響が大きくなるため

実験結果 (2/2)



- 他の3方向でも同様の結果
- ビーコン送信機を中心として同心円状にデッドスポットが存在

-> 座標推定では値が不安定になるデッドスポットを避け， 複数のビーコンの情報を組み合わせることが有効と考えられる

まとめ

iBeacon モジュールを利用した座標位置推定

- 床での反射波の干渉を補正すれば距離の推定精度が向上
- モデル上のデッドスポットでは実測値が不安定になる

今後の課題

- 柱や壁などの障害物による反射の考慮
- 人体による吸収の考慮
- 受信強度値の不安定なデッドスポットを考慮した位置座標推定法の実現

P2P システムのためのスケーラブルな木構造ベースの整合性維持手法

広島大学大学院工学研究科情報工学専攻　分散システム学研究室　中島 大志

概要

P2P ファイル共有システムのための新しい整合性維持手法を提案する．提案手法の基本的な考え方は，共有ファイルごとに静的な木を構築することによって，更新情報を全レプリカノードに効率的に伝播させるというものである．木の根へのリンクは，共有ファイルから木の根へのマッピングを保存した Chord リングを参照することで得られる．提案手法の性能はシミュレーションによって評価される．シミュレーションの結果，メッセージ数・更新伝播遅延・離脱耐性において従来手法に対して優位性があることが示された．

1　はじめに

動画や音楽といったコンテンツを P2P ネットワーク上で共有する P2P ファイル共有システムは，世界中で活用されている．Gnutella, WinMX といった従来の P2P ファイル共有システムは，ユーザが共有ファイルを更新することができなかった．しかし，近年では，オンライン・ストレージや共同編集システムなど，ユーザが共有ファイルを更新できる P2P システムが高い注目を集めている．

一方，P2P ファイル共有システムでは，共有ファイルのレプリカ（複製）を分散配置することが一般的である．レプリカによって，負荷分散，離脱耐性，クエリ検索速度の向上といった性能面での利点が得られる．

もしも P2P システムがレプリカを配置しながら，そのユーザが共有ファイルを更新できる機能を提供する場合，共有ファイルの更新時にその全レプリカに更新情報を伝達してレプリカも正しく更新を行う必要がある．それを行う機構を**整合性維持機構**といい，これまでに多くの整合性維持手法が考えられてきた．中でも，Li らの手法 [4] は，レプリカノードのみで構成される Chord リングを用いて更新情報を伝えることで，効率的に更新情報の伝播を行うことに成功している．

本研究では，Li らの手法の問題点に着目して，新しい木構造ベースの整合性維持手法を提案する．提案手法では，静的にレプリカノードのみから木構造を構成して，共有ファイルが更新されたときにはその木の根から葉へ向けて更新情報を伝播する．更新時に木構造の根ノードを知るため，システム全体で 1 つの Chord リングを構成しておき，共有ファイルから木の根ノードへのマッピングを提供する．ノードが離脱した場合には木構造が崩れるため，先祖ノードも記憶しておくといった離脱耐性手法をとっている．

本研究の性能は P2P シミュレータである OverSim[1] によるシミュレーションで評価し，提案手法は Li らの手法よりメッセージ数・更新伝播遅延・離脱率においてまさっていることが示された．

2　関連研究

既存の整合性維持手法としては，push/pull[3]，IRM[5]，SCOPE[2] などがあるが，中でも効率がよいことが実験的に示されている手法が Li らの手法 [4] である．Li らの手法では，共有ファイルごとにレプリカノードのみからなる Chord リング [6] を構成する．そして更新時には，Chord リングから木構造を動的に抽出して，その根から葉へ向けて更新情報を伝播していく．ノード x が共有ファイル f の更新を行ったとして，f の Chord リングから木構造を抽出する方法は次の通りである．

- まず，ノード x は木構造の根ノードになる．
- x を除いた f の Chord リング空間を考え，この空間を d 個の部分空間に分割する．
- d 個の部分空間 S_i のそれぞれ最初のノード i_1 を，x の子ノードとする．また，i_1 の担当空間を S_i とする．
- 同様に，各子ノードは自分の担当空間を d 個の部分空間に分割して，それぞれの最初のノードを自分の子ノードとする．
- 以上を繰り返すことで，x を根として Chord リング上のノードだけからなる木構造が構成される．

Li らの手法では，共有ファイルごとに Chord リングを構成するため，共有ファイルの数が増加するにつれて Chord リングのメンテナンス・コストが増加していくという欠点がある．

3 提案手法

3.1 概要

提案手法では，共有ファイルごとに静的にレプリカノードからなる木構造を構成する．そして，共有ファイルの更新時には，(1) 木構造の根ノードを発見して，(2) 根ノードから順に子ノードに対して更新情報を伝達していくことで，更新情報の伝播を行う．Li らの手法と異なり，共有ファイルごとに Chord リングを構成する必要がないため，メンテナンス・コストは Li らの手法より少ないと考えられる．実際，Li らの手法では，共有ファイルごとに各ノードは $\Theta(\log X)$ 個の隣接ノードを管理する必要がある一方で，提案手法では，共有ファイルごとに各ノードは自身の子ノードと定数個の先祖ノードのみを管理すればよい．

木構造を設計する上で考慮した課題は次の通りである：

- 共有ファイル f の更新を行うためには，まず f に対応する木の根ノードを発見する必要がある．提案手法では，Chord リングを用いて共有ファイルから根ノードへのマッピングを用意することで，高速な根ノードの発見を可能にしている．
- 木のリンクに沿って更新情報を伝播する場合，木構造はできるだけ平衡化されていることが望ましい．また，親ノードの負荷を抑えるため，子ノード数を制限する必要がある．提案手法では，レプリカノードの追加時に，子ノード数を一定に制限する一方で，子孫ノード数を利用して木構造をできるだけ平衡化する方法を取り入れている．
- 葉以外のノードが木から離脱すると，木構造が分断されてしまい，更新情報が伝播されなくなってしまう．提案手法では，ノード離脱時に木構造を修復する方法を提供している．

3.2 根ノードの発見

共有ファイル f に対応する木を T_f，その木の根ノードを r_f と記すことにしよう．T_f は f のレプリカを持つ全ノードから構成される．f を更新した場合には，まず T_f の根ノードを発見して，更新情報を T_f の根ノード r_f へ送信する．更新情報を受け取った根ノードは自分の子ノードへ更新情報を伝達し，各子ノードも同様に更新情報を自分の子ノードへ伝えていく．結果として，f の全レプリカノードに更新情報が伝播される．ここで，根ノードの発見には Chord リングが用いられる．ファイル共有システムに参加している全ノードで 1 個の Chord リングを構成し，この Chord リングは共有ファイル f の集合から木の根ノード r_f の集合への写像を格納している．f の更新を行うノードは，Chord リングに対して木の根ノード r_f を問い合わせることで，更新情報を送信する木の根ノードを知ることができる．共有ファイルごとに Chord リングを構成する Li らの手法と異なり，提案手法はシステム全体で 1 個の Chord リングしか必要としないことに注意しよう．

根ノードの発見を高速に行うために，提案手法ではキャッシュ機構を用意する．あるノード x が 1 度 Chord リングの探索を行って，共有ファイル f から木の根ノード r_f へのキーバリューペアが得られたとしよう．x はこのキーバリューペアの情報をキャッシュしておき，再び f の更新を行った際には，Chord の探索を行わずにキャッシュされた r_f に直接更新情報を送る．もしも，キャッシュされた r_f が離脱しているか，すでに T_f の根ノードでなくなっていれば，x は Chord の探索を行って正しい根ノードを発見し直す．

3.3 レプリカノードの追加

T_f 中の各ノード u は，T_f における子ノード集合 $Child_f(u)$ と子孫ノード数 $D_f(u)$ を局所的に保持している．また，システムのパラメータとして最大子ノード数 d を用意する．ノード x が新たに f のレプリカノードになるとしよう．ノード x は T_f 中のノード y から f のレプリカを取得した後，自身の局所変数を $Child_f(z) := \emptyset, D_f(x) := 1$ という形で初期化して，ノード y に向けて参加要求 $\mathsf{Add}(x)$ を送る．$\mathsf{Add}(x)$ を受け取った y は，局所変数 $D_f(y)$ の値を 1 増やした後，

- もし $|Child_f(y)| < d$ ならば，$Child_f(y)$ に x を加えて x を自分の子とした上で，その旨を x に通知し，
- そうでなければ，$Child_f(y)$ の中でもっとも変数 D_f の値が小さな子ノードに向けて参加要求 $\mathsf{Add}(x)$ を転送する．

同様の手続きは，参加要求を受け取ったノード上でも，(x の親ノードが決まるまで) 繰り返し実行される．根から葉へのどんな経路でも，$|Child_f(y')| < d$ となるようノード y' は最低 1 個存在するため，この繰り返しは必ず終了することに注意しよう（たとえば葉ノードは必ずこの条件を満たす）．

3.4 離脱耐性

まずは，根ノード以外のノードが離脱した場合の対処法を示す．各ノードは定数世代分の祖先ノードを記録してお

くための変数 Anc_f を用意しておく．そして，隣接ノードの離脱が起きたかどうかは，隣接ノード間で定期的にメッセージ通信を行うことで検知できる．親ノードの離脱を検知したノード x は，Anc_f の中でシステムに参加しているもっとも根に近いノード y に向けて再参加要求 Add(x) を送る．Add(x) を受け取ったノード y の動きは，前節と同様である．

次に，根ノードが離脱した場合の対処法を示す．提案手法では，根ノードの子からランダムに $c(\geq 1)$ 個のノードを選択して根のコピーノードとする．そして，Chord リングには，キーバリューペアの値として，根ノードだけではなく，根ノードおよびそのコピーノードを登録する．根ノード r_f とそのコピーノードは，ファイル f のレプリカと変数 $Child_f(r), D_f(r)$ という 3 つの情報を共有する．ファイル f を更新するときや根ノードに対して Add 要求を送るときに根ノードが離脱していた場合，コピーノードをランダムに 1 つ選択して新しい根ノード r'_f として，残りのコピーノードは r'_f のコピーノードとなる．また，根ノード r_f は自らのコピーノードが c 個未満になった場合，まだコピーノードになっていない任意のノードを $Child_f(r_f)$ から選択して，コピーノードとする．

4 評価

4.1 設定

OverSim[1] を用いたシミュレーションにより，提案手法の性能を実験的に評価した．シミュレーションの設定は次の通りである．ノード数を 500，共有ファイル数を 500 に固定した P2P ファイル共有システムを考える．各ノードは $s = 1, N = 500$ の Zipf 分布に従った人気度によってファイルを共有している．各ファイルはレプリカノードによって秒単位で $\lambda = 0.05$ のポアソン分布に従って更新される．各ノードは秒単位で $\lambda = 10000, k = 1.0$ のワイブル分布にしたがって参加・離脱する．パラメータ d の値は $d = 16$ と定義している．

総メッセージ数・更新伝播遅延・離脱耐性という 3 つの項目に関して提案手法の性能を評価し，2 で述べた Li らの手法と比較した．

図 1 は総メッセージ数の結果で，図 2 は更新伝播遅延の結果である．提案手法は Li らの手法よりもよい結果を示し，またキャッシュの効果によりさらに総メッセージ数と更新伝播遅延の削減が行われていることが分かる．メッセージの内訳を分析したところ，Li らの手法は Chord リングのメンテナンスに大きな負荷がかかっていた．

全参加ノード中の D[%] が “同時に” 離脱した状況を考える．この離脱率 D に応じた更新伝播遅延を評価した．図 3 に結果を示す．提案手法は離脱率が上昇するとともに更新伝播遅延も増加しているが，それでも Li らの手法より低い離脱率に抑えられていることが分かる．

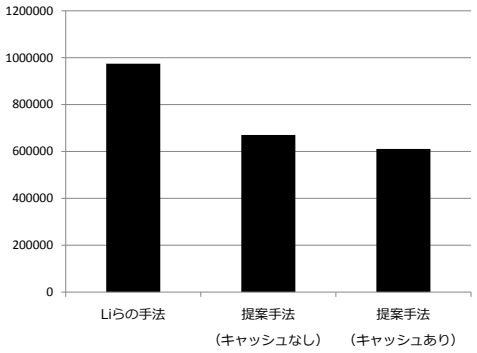


図 1　総メッセージ数.

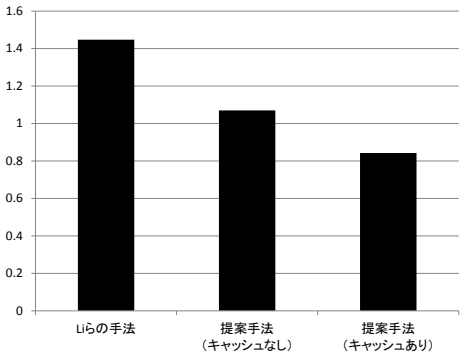


図 2　更新伝播遅延.

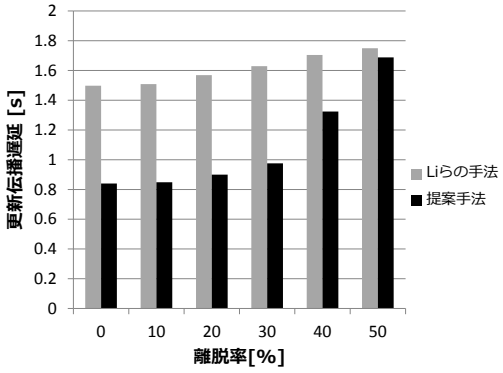


図 3　離脱率に応じた更新伝播遅延.

5 おわりに

本研究では，共有ファイルごとに静的な木を構築することによる，P2P ファイル共有システム向けの整合性維持手法を提案した．Li らの手法と異なり，提案手法は共有ファイルごとの Chord リングを必要とせず，根ノードの予備を用意して先祖ノードを記憶しておくことで離脱耐性を実現している．シミュレーションの結果，提案手法は Li らの手法よりメッセージ数・更新伝播遅延・離脱率においてまさっていることを示した．

今後の課題として，実際のアプリケーションを開発した上での実験評価などがある．また提案手法の向上方法として，木の子ノード数を適応的に決定していくことが考えられる．

参考文献

[1] Ingmar Baumgart, Bernhard Heep, and Stephan Krause. OverSim: A Flexible Overlay Network Simulation Framework. In *Proceedings of 10th IEEE Global Internet Symposium Symposium*, pages 79–84, 2007.

[2] Xin Chen, Shansi Ren, Haining Wang, and Xiaodong Zhang. SCOPE: Scalable Consistency Maintenance in Structured P2P Systems. In *IEEE INFOCOM*, volume 3, pages 1502–1513, 2005.

[3] Jiang Lan, Xiaotao Liu, Prashant Shenoy, and Krithi Ramamritham. Consistency Maintenance in Peer-to-Peer File Sharing Networks. In *Proceedings of the The Third IEEE Workshop on Internet Applications*, WIAPP '03, pages 90–94, 2003.

[4] Zhenyu Li, Gaogang Xie, and Zhongcheng Li. Efficient and Scalable Consistency Maintenance for Heterogeneous Peer-to-Peer Systems. *IEEE Transactions on Parallel and Distributed Systems*, 19(12):1695–1708, 2008.

[5] Haiying Shen. IRM: Integrated File Replication and Consistency Maintenance in P2P Systems. *IEEE Transactions on Parallel and Distributed Systems*, 21(1):100–113, 2010.

[6] Ion Stoica, Robert Morris, David Liben-Nowell, David R Karger, M Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, 2003.

レビューを対象とした信頼性判断支援システムの提案

伊木　惇（中国電力）
亀井清華（広島大学）
藤田　聡（広島大学）

背景（1）

- Amazonや楽天市場などのレビュー
  - Web上の商品購入者の60%
    - 評価値やレビューを含んだサイトから購入
  - 購入者の70%
    - レビューを読んでから購入
- レビューは読者の購買意欲を左右する

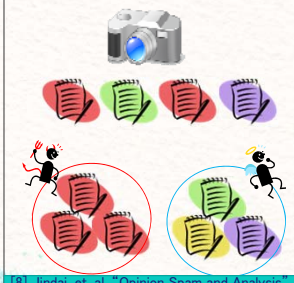
背景（2）

- ステルスマーケティング（サクラ・スパムの存在）
  - サクラ
    - 商品の評判を意図的に上げる（下げる）ために不当な偽のレビューの投稿を行う悪意を持った投稿者
  - スパム
    - サクラによって投稿されたレビュー
    - レビューを読んだユーザを騙し，誤解させる

- 各レビューがスパムであるか否かを判断する必要
  - スパムらしい＝信頼性が低い

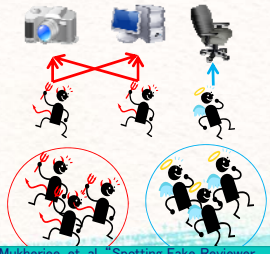
関連研究～スパム検知（1）

- **Jindaiらの研究**
  - 複製されたレビューはスパムらしい



[8] Jindai, et. al, "Opinion Spam and Analysis", 2008

- **Mukherjeeらの研究**
  - 特定の商品に，協力してレビューを投稿しているユーザはサクラらしい



[7] Mukherjee, et. al, "Spotting Fake Reviewer Groups in Consumer Reviews", 2012

関連研究～スパム検知（2）

- スパムであるか否かを二値分類する
  - スパムの一面的な特徴を捉えたもの
    - 全てのスパムの検知には至っていない
  - スパムとスパム検知はいたちごっこ
    - 検知だけでは不十分
  - 精度は人手で判断したものとの比較
    - 人手での判断は難しい
      - 信頼性を判断するために役立つ情報が乏しい
      - 必要な情報を個人が十分に収集するのは困難

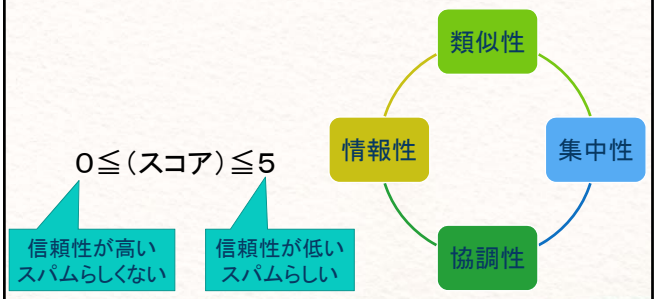
研究目的

レビューを対象とした信頼性判断支援

- 「スパムらしさ」を表す指標をユーザに提示
  - ユーザ自身が信頼性を意識することを促す
  - ユーザ自身が信頼性を判断できるように支援する

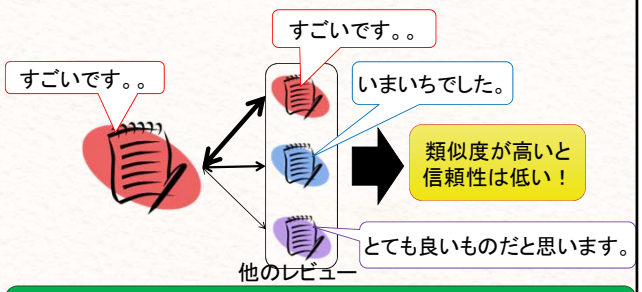
信頼性指標

●各レビューに各指標に関するスコアを提示



類似性

●どの程度，他のレビューの文章と類似しているか



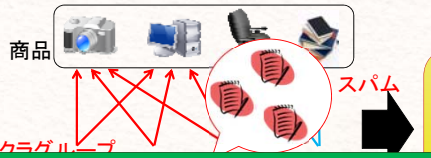
各レビューをベクトル化して，同じジャンルの他のレビューとの間の類似度を計算し，最大の類似度を正規化

[8] Jindai, et. al, "Opinion Spam and Analysis", 2008

協調性

●どの程度，サクラグループに投稿された可能性があるか

■サクラグループ：同じグループのメンバーが同じ商品に対して投稿を行うことで，協力して評判を変える



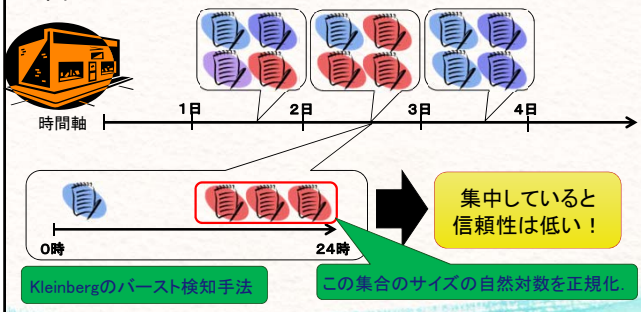
1. 頻出アイテムセット抽出アルゴリズムを用いて，4つ以上同じ商品に投稿した3人以上の投稿者グループを抽出．
2. 抽出した各グループについて，
   協調度＝(グループの大きさ)×(共通した商品の数)
3. スコアは投稿者が属しているグループの内の最大協調度を正規化

[7] Mukherjee, et. al, "Spotting Fake Reviewer Groups in Consumer Reviews", 2012

集中性

●どの程度，時間的に集中して投稿されたか

■普段より短時間に多く投稿がされているときのものを検出



Kleinbergのバースト検知手法

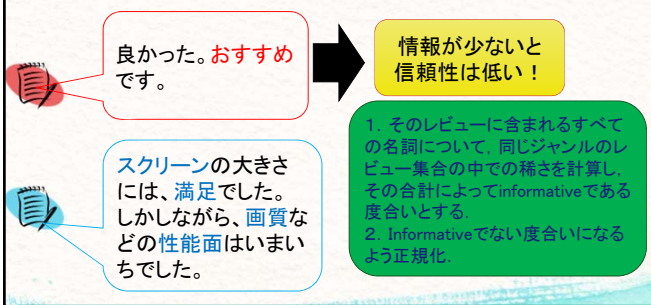
この集合のサイズの自然対数を正規化．

[9] Xie, et. al, "Review Spam Detection via Temporal Pattern Discovery", 2012

情報性

●どの程度，informativeでない文章か

■Informativeなレビュー： 他のレビューであまり使われていない名詞を多く含んだレビュー



1. そのレビューに含まれるすべての名詞について，同じジャンルのレビュー集合の中での稀さを計算し，その合計によってinformativeである度合いとする．
2. Informativeでない度合いになるよう正規化．

[10] Ott, et. al, "Finding Deceptive Opinion Spam By Any Stretch of the Imagination", 2011

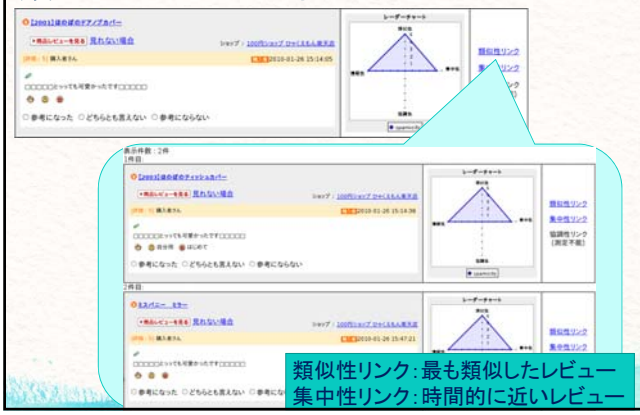
提案システム(1)

提案システム（2）

①指標の視覚化
- 信頼性意識の向上
- 直観的な判断

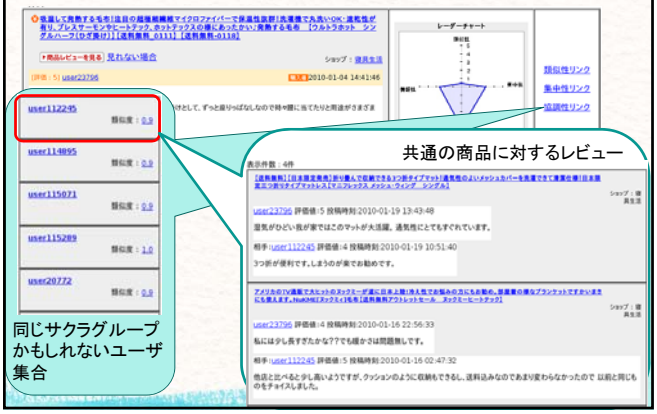
②情報にアクセスするためのリンク
- 判断材料となる情報を提示

類似性リンク及び集中性リンク



類似性リンク：最も類似したレビュー
集中性リンク：時間的に近いレビュー

協調性リンク



データセット

●楽天市場の「みんなのレビュー・口コミ」データ

●期間：2010年1月1日～1月31日の1カ月間

●ジャンルの内訳
- ■本（44,668件）
- ■家電（57,469件）
- ■家具（82,064件）
- ■合計（184,201件）

予備実験

●各指標に基づいてスパムレビューを抽出する

指標	スコア5の件数	スコア4以上の件数
類似性	9265	926
協調性	67	381
集中性	59	154
情報性	813	1117
類似性＋協調性	3	270
類似性＋集中性	7	116
類似性＋情報性	131	145
協調性＋集中性	0	0
協調性＋情報性	0	3
集中性＋情報性	0	5
類似性＋集中性＋情報性	0	3

本：5438
家電：1497
家具：2330

集中性は普段から高い（低い）評価ばかり投稿されていると高くなりにくい

事前アンケート

	普段からレビューがスパムか意識してますか？
はい	9（43%）
いいえ	12（57%）

●被験者21人にレビューを読む際に見る情報を選択してもらった
- ■普段からスパムを意識する人であっても，以下の情報はほとんど見ないことが確認された
  - ◆レビューの投稿時間
  - ◆投稿者履歴
  - ◆他の商品のレビュー

実験（アンケート）

- 提案システムを利用しない場合と利用する場合との比較
  - 被験者21人にレビューを1件ずつ，計30件を提示
    - 投稿者履歴，店情報，商品情報，他のレビュー等へのアクセス可
  - 「信頼できる」・「どちらとも言えない」・「信頼できない」の判断を集計
  - 判断の参考にした情報を集計
  - システムを利用しない場合の実験から1週間経過後，システムを利用する場合の実験
    - 同じレビュー30件を順番を入れ替えて提示

判断結果

- 被験者の信頼できるかどうかの内訳
  - 21人×30件=630回答

	信頼できる	どちらとも言えない	信頼できない
利用しない	49%	22%	29%
利用する	39%	14%	47%

- 被験者間の判断の一致度(21人)

	一致度(Fleissのκ係数)
利用しない	0．22

- 値が1

信頼性の判断が容易になっている

参考にした情報

- 参考にした情報（1つのレビューあたりの平均参考人数）

	レビュー本文	投稿者履歴	他の投稿者のレビュー	ショップ履歴
利用しない	18．1人	8．8人	7．6人	1．2人
利用する	16．1人	4．3人	2．9人	0．2人

- 提案システムを利用する場合のみ

類似性		集中性		協調性		情報性
スコア	リンク	スコア	リンク	スコア	リンク	スコア
11.2人	8.0人	5.8人	3.5人	7.1人	3.8人	2.5人

信頼性に対する意識

- 実験前

	普段からレビューがスパムか意識してますか？
はい	9(43%)
いいえ	12(57%)

- 実験後

	信頼性を判断する必要を感じましたか？
はい	20(95%)
いいえ	0(0%)
どち	

信頼性に関する意識の向上↑

まとめと今後の課題

- まとめ
  - レビューを対象とした信頼性指標とシステム提案
    - →信頼性の意識の向上を促進
    - →有効な判断支援
- 今後の課題
  - 大規模な実験・データセットの利用
    - 被験者数やデータセットの期間を増やす
  - 別の指標の検討
    - レビュー本文の意味や評価値との整合性など

An Efficient Silent Self-Stabilizing Algorithm for 1-Maximal Matching in Anonymous Network

Yuma Asada and Michiko Inoue

Nara Institute of Science and Technology, Ikoma, Nara 630-0192 Japan
{asada.yuma.ar4, kounoe}@is.naist.jp

Abstract. We propose a new self-stabilizing 1-maximal matching algorithm which is *silent* and works for any *anonymous* networks without a cycle of a length of a multiple of 3 under a central *unfair* daemon. Let n and e be the numbers of nodes and edges in a graph, respectively. The time complexity of the proposed algorithm is $O(e)$ moves. Therefore, the complexity is $O(n)$ moves for trees or rings whose length is not a multiple of 3. That is a significant improvement from the best existing results of $O(n^4)$ moves for the same problem setting.

Keywords: distributed algorithm, self-stabilization, graph theory, matching problem

1 Introduction

Self-Stabilization [5] can tolerate several inconsistencies of computer networks caused by transient faults, erroneous initialization, or dynamic topology change. It can recover and stabilize to consistent system configuration without restarting program execution.

Maximum or *maximal matching* is a well-studied fundamental problem for distributed networks. A matching is a set of pairs of adjacent nodes in a network such that any node belongs to at most one pair. It can be used in distributed applications where pairs of nodes, such as a server and a client, are required. This paper proposes an efficient anonymous self-stabilizing algorithm for *1-maximal matching*. A 1-maximal matching is a $\frac{2}{3}$-approximation to the maximum matching, and expected to find more matching pairs than a *maximal matching* which is a $\frac{1}{2}$-approximation to the maximum matching.

Self-stabilizing algorithms for the maximum and maximal matching problems have been well studied[7]. Table 1 summarizes the results, where n and e denote the numbers of nodes and edges, respectively.

Blair and Manne[1] showed that a *maximum* matching can be solved with $O(n^2)$ moves for non-anonymous tree networks under a read/write daemon. They proposed an algorithm to construct a rooted tree, and showed bottom-up algorithms including a maximum matching[2] can be combined with the proposed algorithm so that the combined algorithm simultaneously solves the two problems without increasing the time complexity. For *anonymous* networks, Karaata et

Table 1. Self-stabilizing matching algorithms.

Reference	Matching	Topology	Anonymous	Daemon	Complexity
[1]	maximum	tree	no	read/write	$O(n^2)$ moves
[10]	maximum	tree	yes	central	$O(n^4)$ moves
[3]	maximum	bipartite	yes	central	$O(n^2)$ rounds
[9]	maximal	arbitrary	yes	central	$O(e)$ moves
[6]	1-maximal	tree, ring*	yes	central	$O(n^4)$ moves
[12]	1-maximal	arbitrary	no	distributed	$O(n^2)$ rounds
this paper	1-maximal	arbitrary*	yes	central	$O(e)$ moves

* without a cycle of length of a multiple of 3.

al.[10] proposed a maximum matching algorithm with $O(n^4)$ moves for trees under a central daemon, and Chattopadhyay et al.[3] proposed a maximum matching algorithm with $O(n^2)$ rounds for bipartite networks under a central daemon.

Recently, Datta and Larmore[4] proposed a *silent* weak leader election algorithm for anonymous trees. The algorithm elects one or two co-leaders with $O(n \cdot Diam)$ moves in a bottom-up fashion under an unfair distributed daemon, where $Diam$ is a network diameter. Though there is no description, it seems that it can be combined with the maximum matching algorithm[2] without increasing the time complexity.

Hsu and Huang[9] proposed a *maximal* matching algorithm for anonymous networks with arbitrary topology under a central daemon. They showed the time complexity of $O(n^3)$ moves, and, it has been revealed that the time complexity of their algorithm is $O(n^2)$ moves by Tel[13] and Kimoto et al.[11] and $O(e)$ moves by Hedetniemi et al. [8].

Goddard et al.[6] proposed a *1-maximal* matching with $O(n^4)$ moves for anonymous trees and rings whose length is *not* a multiple of 3 under a central daemon. They also showed that there is no self-stabilizing 1-maximal matching algorithm for anonymous rings with length of a multiple of 3. Manne et al. [12] also proposed a 1-maximal matching algorithm for non-anonymous networks with any topology under a distributed unfair daemon. Their algorithm stabilizes in $O(n^2)$ rounds and $O(2^n \cdot \Delta \cdot n)$ moves, where Δ is the maximum degree of nodes.

Our contribution. In this paper, we propose a new self-stabilizing 1-maximal matching algorithm. The proposed algorithm is *silent* and works for any *anonymous* networks without a cycle of a length of a multiple of 3 under a central *unfair* daemon. We will show that the time complexity of the proposed algorithm is $O(e)$ moves. Therefore, the complexity is $O(n)$ moves for trees or rings whose length is not a multiple of 3. That is a significant improvement of the best existing result of $O(n^4)$ for the same problem setting[6].

The remaining of the paper is organized as follows. In Section 2, we define distributed systems and the 1-maximal matching problem. A 1-maximal matching algorithm is proposed in Section 3, and proves for its correctness and performance are given in Section 4. Finally Section 5 concludes this paper.

2 Preliminaries

A distributed system consists of multiple asynchronous processes. Its topology is represented by an undirected connected graph $G = (V, E)$ where a node in V represents a process and an edge in E represents the interconnection between the processes. A node is a state machine which changes its states by actions. Each node has a set of actions, and a collection of actions of nodes is called a *distributed algorithm*. Let n and e denote the numbers of nodes and edges in a distributed system.

In this paper, we consider *state-reading model* as a communication model where each node can directly read the internal state of its neighbors. An action of a node is expressed $\langle label \rangle :: \langle guard \rangle \mapsto \langle statement \rangle$. A guard is a Boolean function of all the states of the node and its neighbors, and a statement updates its local state. We say a node is privileged if it has an action with a true guard. Only privileged node can *move* by selecting one action with a true guard and executing its statement.

Moves of nodes are scheduled by a *daemon*. Among several daemons considered for distributed systems, we consider an *unfair central daemon* in this paper. A central daemon chooses one privileged node at one time, and the selected node atomically moves. A daemon is unfair in a sense that it can choose any node among privileged nodes.

A problem $\mathcal{P}$ is specified by its legitimate configurations where configuration is a collection of states of all the nodes. We say a distributed algorithm $\mathcal{A}$ is *self-stabilizing* if $\mathcal{A}$ satisfies the following properties. 1) **convergence**: The system eventually reaches to a legitimate configuration from any initial state, and 2) **closure**: The system once reaches to a legitimate configuration, all the succeeding moves keep the system configuration legitimate. A self-stabilizing algorithm is *silent* if, from any arbitrary initial configuration, the system reaches a terminal configuration where no node can move. A self-stabilizing algorithm is *anonymous* if it does not use global IDs of nodes. We only assume that nodes have pointers and a node can determine whether its neighbor points to itself, some other nodes, or no node.

A *matching* in an undirected graph $G = (V, E)$ is a subset M of E such that each node in V is incident to at most one edge in M. We say a matching is *maximal* if no proper superset of M is a matching as well. A maximal matching M is *1-maximal* if, for any $e \in M$, any matching cannot be produced by removing e from M and adding two edges to $M - \{e\}$. A maximal matching is a $\frac{1}{2}-$approximation to the maximum matching. On the other hand, a 1-maximal matching is a $\frac{2}{3}-$approximation. In this paper, we propose a silent and anonymous self-stabilizing algorithm for the 1-maximal matching problem for graphs without a cycle of length of a multiple of 3.

3 Algorithm MM1

First, we will show an overview of a proposed self-stabilizing 1-maximal matching algorithm MM1. Each node i uses stages to construct 1-maximal matching. There

are seven stages; *S1a*, *S1b*, *S2a*, *S2b*, *S3*, *S4*, and *S5*. Stages *S1a* and *S1b* mean that the node is not matching with any node. A stage *S2a* means the node is matching with a neighbor node, and, *S2b*, *S3*, *S4*, *S5* mean the node is trying to increase matches. A node i has three variables; $\texttt{level}_i$, $\texttt{m-ptr}_i$, $\texttt{i-ptr}_i$. We describe how to use the variables in our algorithm.

S1a, S1b, S2a We say a node is *free* if the node is in *S1a* or *S1b*. A node in *S1a* does not invite any nodes, while a node in *S1b* invites its neighbor node. Fig.1 shows how free nodes make a match. When a free node i finds a free neighbor node j, i invites j by $\texttt{i-ptr}_i$ (i is in *S1b*). Then invited node j updates its level to 2 and points to i by $\texttt{m-ptr}_j$ to accept the invitation (j is in *S2a*). Finally i points to j by $\texttt{m-ptr}_i$ to make a match (i is in *S2a*). A node in *S2a* is at level 2 and does not invite any nodes. If two adjacent nodes i and j point to each other by `m-ptr`, we consider they are matching, that is $(i, j) \in M$.

S2b, S3, S4, S5 Matching nodes try to increase the number of matches if they have free neighbor nodes. Fig.2 shows how to increase matches, where matches are increased by breaking a match between i and j, and creating new matches between i and k, and j and l. In Fig.2(a), nodes i and j invite their free neighbors k and l if they do not invite i and j, respectively (i and j are in *S2b*). When both nodes notice that i and j invite free neighbor nodes, they change their level to 3 (i and j are in *S3*). That indicates that they are ready to be approved as in Fig.2(b). Then k and l point to the inviting nodes by `i-ptr` to approve their invitations (k and l are in *S1b*). Node i and j change their level to 4 if the neighbors approve the invitations (i and j are in *S4*) as in Fig.2(c), and change their level to 5 when they notice that both invitations are approved (i and j are in *S5*). This indicates that they are ready to break a match as in Fig.2(d). Then they create new matches with the free nodes, where k and l first move to *S2a* (Fig.2(e)) and then i and j move to *S2a* (Fig.2(f)), respectively. A node in *S1a* or *S1b* can make a match with the other node while an inviting node is in *S3*. However, once the inviting node moves to *S4*, it cannot change its `i-ptr` while the inviting node is in *S4*.

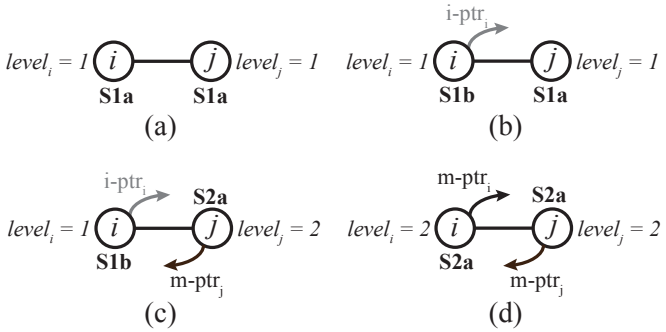


Fig. 1. Making a match between free nodes

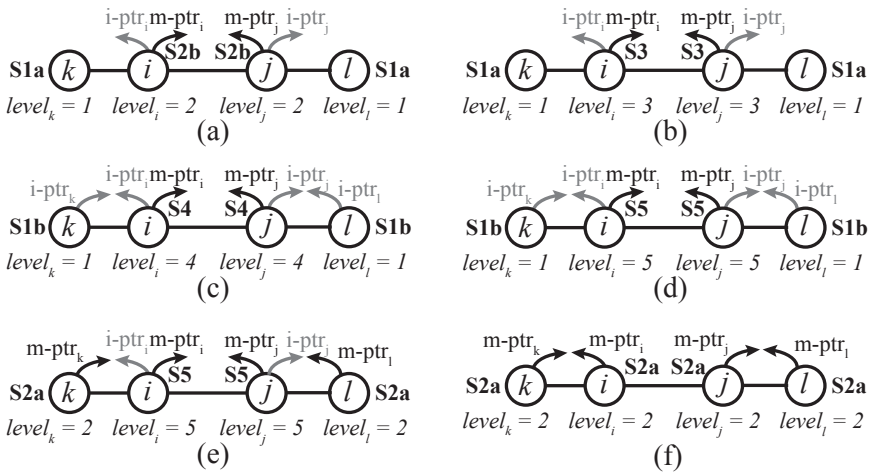


Fig. 2. Increasing matches

Reset Each node always checks its validity, and resets to *S1a* if it finds its invalidity. We consider two kinds of validities, *one node validity* and *two nodes validity*. The one node validity means that a state represents some stage. For example, if a level is 1 and `m-ptr` points to some neighbor, the state is one node invalid. The two nodes validity means that a relation between states of two adjacent nodes is consistent. For example, if a node i is in *S2a*, a node pointed to by `m-ptr` should point to i by `m-ptr` at level 2 or by `i-ptr` at level 1 or 5. The full definition of the validity function is shown in Fig.3. A node does not move while some neighbor is one node invalid.

Cancel A node cancels an invitation or progress to increase matches, if it detects that the invitation cannot be accepted or it cannot increase matches. When canceling, a node goes back to *S1a* if it is at level 1, and to *S2a* if it is at level 2 or higher.

The algorithm MM1 uses some statement macros and a guard function. The variables, validity functions, statement macros and a guard function are shown in Fig.3, and a code of MM1is shown in Fig.4. In the algorithm, each node i uses $N(i)$ to represent a set of its neighbors. That is a set of local IDs for each node and the algorithm does not use any global IDs. We only assume that each node can determine whether its neighbor point to itself, some other node, or no node by pointers `i-ptr` and `m-ptr`.

4 Correctness

Lemma 1. *There are no nodes at level 5 in any terminal configuration of MM1.*

Proof. By contradiction. Assume that a node i is in *S5* in a terminal configuration. In this case, $\texttt{i-ptr}_i = k$ holds for some k, and $\texttt{level}_k = 1 \wedge \texttt{i-ptr}_k = i$ or

Variables
$\texttt{level}_i \in \{1,2,3,4,5\}$
$\texttt{m-ptr}_i \in N(i) \cup \{\perp\}$
$\texttt{i-ptr}_i \in N(i) \cup \{\perp\}$

Valid Predicates
S1b_valid(i,k): $\texttt{level}_i = 1 \wedge \texttt{m-ptr}_i = \perp \wedge \texttt{i-ptr}_i = k$
S2a_valid(i,j): $\texttt{level}_i = 2 \wedge \texttt{m-ptr}_i = j \wedge \texttt{m-ptr}_i = \perp$
S2b_valid(i,j,k): $\texttt{level}_i = 2 \wedge \texttt{m-ptr}_i = j \wedge \texttt{m-ptr}_i = k \wedge j \neq k$
S3_valid(i,j,k): $\texttt{level}_i = 3 \wedge \texttt{m-ptr}_i = j \wedge \texttt{m-ptr}_i = k \wedge j \neq k$
S4_valid(i,j,k): $\texttt{level}_i = 4 \wedge \texttt{m-ptr}_i = j \wedge \texttt{m-ptr}_i = k \wedge j \neq k$
S5_valid(i,j,k): $\texttt{level}_i = 4 \wedge \texttt{m-ptr}_i = j \wedge \texttt{m-ptr}_i = k \wedge j \neq k$

One Node Validity
S1a_valid1(i): $\texttt{level}_i = 1 \wedge \texttt{m-ptr}_i = \perp \wedge \texttt{i-ptr}_i = \perp$
S1b_valid1(i): $\exists k \in N(i)$ `S1b_valid(i,k)`
S2a_valid1(i): $\exists j,k \in N(i)$ `S2a_valid(i,j)`
S2b_valid1(i): $\exists j,k \in N(i)$ `S2b_valid(i,j,k)`
S3_valid1(i): $\exists j,k \in N(i)$ `S3_valid(i,j,k)`
S4_valid1(i): $\exists j,k \in N(i)$ `S4_valid(i,j,k)`
S5_valid1(i): $\exists j,k \in N(i)$ `S4_valid(i,j,k)`
valid1(i): *S1a_valid(i)* $\wedge$ *S1b_valid(i)* $\wedge$ *S2a_valid(i)* $\wedge$ *S2b_valid(i)* $\wedge$ *S3_valid(i)* $\wedge$ *S4_valid(i)* $\wedge$ *S5_valid(i)*
invalid1(i): $\neg$ *valid1(i)*

Valid Functions (One Node Validity and Two Node Validity)
S1a(i): *S1a_valid1(i)*
S1b(i): *S1b_valid1(i)*
S2a(i): $\exists j \in N(i)(S2a\_valid(i,j) \wedge (\texttt{level}_j = 2 \wedge \texttt{m-ptr}_j = i) \vee (\texttt{level}_j = 1 \wedge \texttt{i-ptr}_j = i) \vee (\texttt{level}_j = 5 \wedge \texttt{i-ptr}_j = i))$
S2b(i): $\exists j,k \in N(i)(S2b\_valid(i,j,k) \wedge (\texttt{level}_j = 2 \vee \texttt{level}_j = 3 \vee \texttt{level}_j = 4) \wedge \texttt{m-ptr}_j = i)$
S3(i): $\exists j,k \in N(i)(S3\_valid(i,j,k) \wedge (\texttt{level}_j = 2 \vee \texttt{level}_j = 3 \vee \texttt{level}_j = 4) \wedge \texttt{m-ptr}_j = i)$
S4(i): $\exists j,k \in N(i)(S4\_valid(i,j,k) \wedge (\texttt{level}_j = 2 \vee \texttt{level}_j = 3 \vee \texttt{level}_j = 4 \vee \texttt{level}_j = 5) \wedge \texttt{m-ptr}_j = i \wedge \texttt{i-ptr}_j \neq \perp \wedge \texttt{level}_k = 1 \wedge \texttt{i-ptr}_k = i)$
S5(i): $\exists j,k \in N(i)(S5\_valid(i,j,k) \wedge (\texttt{level}_k = 1 \wedge \texttt{i-ptr}_k = i) \vee (\texttt{level}_k = 2 \wedge \texttt{m-ptr}_k = i))$
valid(i): *S1a(i)* $\wedge$ *S1b(i)* $\wedge$ *S2a(i)* $\wedge$ *S2b(i)* $\wedge$ *S3(i)* $\wedge$ *S4(i)* $\wedge$ *S5(i)*
invalid(i): $\neg$ *valid(i)*

Statement Macros
`make_match`: $\texttt{i-ptr}_i = \perp, \texttt{m-ptr}_i = j, \texttt{level}_i = 2$
`reset_state`: $\texttt{i-ptr}_i = \perp, \texttt{m-ptr}_i = \perp, \texttt{level}_i = 1$
`abort_exchange`: $\texttt{i-ptr}_i = \perp, \texttt{level}_i = 2$

Guard Function
no_invalid1_neighbor(i): $\forall x \in N(i)$ *valid1(x)*

Fig. 3. Variables, validity functions, statement macros and guard function

Reset
reset1 :: *invalid1(i)* $\mapsto$ reset_state
reset2 :: *invalid1(i)* $\wedge$ *no_invalid1_neighbor(i)* $\mapsto$ reset_state

S1a
match1 :: *S1a(i)* $\wedge$ *no_invalid1_neighbor(i)* $\wedge \exists x \in N(i)(\texttt{i-ptr}_x = i \wedge \texttt{level}_x = 1) \mapsto \texttt{i-ptr}_i = \perp, \texttt{m-ptr}_i = x, \texttt{level}_i = 2$
approve1 :: *S1a(i)* $\wedge$ *no_invalid1_neighbor(i)* $\wedge \exists x \in N(i)(\texttt{i-ptr}_x = i \wedge \texttt{level}_x = 3) \mapsto \texttt{i-ptr}_i = x$
invite1 :: *S1a(i)* $\wedge$ *no_invalid1_neighbor(i)* $\wedge \exists x \in N(i) \texttt{level}_x = 1 \mapsto \texttt{i-ptr}_i = x$

S1b
match2 :: *S1b(i)* $\wedge$ *no_invalid1_neighbor(i)* $\wedge \exists x \in N(i)(\texttt{i-ptr}_x = i \wedge \texttt{level}_x = 1) \wedge \exists k \in N(i)($*S1b_valid(i,k)* $\wedge \texttt{level}_k < 4) \mapsto \texttt{i-ptr}_i = \perp, \texttt{m-ptr}_i = x, \texttt{level}_i = 2$
match3 :: *S1b(i)* $\wedge$ *no_invalid1_neighbor(i)* $\wedge \exists k \in N(i)($*S1b_valid(i,k)* $\wedge \texttt{m-ptr}_k = i \wedge \texttt{level}_k = 2) \mapsto$ make_match
migrate1 :: *S1b(i)* $\wedge$ *no_invalid1_neighbor(i)* $\wedge \exists k \in N(i)($*S1b_valid(i,k)* $\wedge \texttt{i-ptr}_k = i \wedge \texttt{level}_k = 5) \mapsto$ make_match
cancel1 :: *S1b(i)* $\wedge$ *no_invalid1_neighbor(i)* $\wedge \exists k \in N(i)($*S1b_valid(i,k)* $\wedge (\texttt{level}_k = 2 \vee (\texttt{level}_k = 3 \wedge \texttt{i-ptr}_k \neq i) \vee (\texttt{level}_k = 4 \wedge \texttt{i-ptr}_k \neq i) \vee (\texttt{level}_k = 5 \wedge \texttt{i-ptr}_k \neq i))) \mapsto \texttt{i-ptr}_i = \perp$

S2a
invite2 :: *S2a(i)* $\wedge$ *no_invalid1_neighbor(i)* $\wedge \exists x \in N(i)(\texttt{level}_x = 1 \wedge \texttt{i-ptr}_x \neq i) \wedge \exists j \in N(i)($*S2a_valid(i,j)* $\wedge \texttt{m-ptr}_j = i) \mapsto \texttt{i-ptr}_i = x$

S2b
cancel2 :: *S2b(i)* $\wedge$ *no_invalid1_neighbor(i)* $\wedge \exists j, k \in N(i)($*S2b_valid(i,j,k)* $\wedge \texttt{level}_k \geq 2) \mapsto$ abort_exchange
proceed1 :: *S2b(i)* $\wedge$ *no_invalid1_neighbor(i)* $\wedge \exists j, k \in N(i)($*S2b_valid(i,j,k)* $\wedge \texttt{i-ptr}_j \neq \perp) \mapsto \texttt{level}_i = 3$

S3
cancel3 :: *S3(i)* $\wedge$ *no_invalid1_neighbor(i)* $\wedge \exists j, k \in N(i)($*S3_valid(i,j,k)* $\wedge ((\texttt{level}_j = 2 \wedge \texttt{i-ptr}_j = \perp) \vee \texttt{level}_k \geq 2)) \mapsto$ abort_exchange
proceed2 :: *S3(i)* $\wedge$ *no_invalid1_neighbor(i)* $\wedge \exists j, k \in N(i)($*S3_valid(i,j,k)* $\wedge \texttt{i-ptr}_k = i \wedge \texttt{level}_k = 1) \mapsto \texttt{level}_i = 4$

S4
cancel4 :: *S4(i)* $\wedge$ *no_invalid1_neighbor(i)* $\wedge \exists j, k \in N(i)($*S4_valid(i,j,k)* $\wedge \texttt{level}_j = 2 \wedge \texttt{i-ptr}_j = \perp) \mapsto$ abort_exchange
proceed3 :: *S4(i)* $\wedge$ *no_invalid1_neighbor(i)* $\wedge \exists j, k \in N(i)($*S4_valid(i,j,k)* $\wedge (\texttt{level}_j = 4 \vee \texttt{level}_j = 5)) \mapsto \texttt{level}_i = 5$

S5
migrate2 :: *S5(i)* $\wedge$ *no_invalid1_neighbor(i)* $\wedge \exists j, k \in N(i)($*S5_valid(i,j,k)* $\wedge \texttt{level}_k = 2 \wedge \texttt{m-ptr}_k = i \wedge \texttt{i-ptr}_k = \perp \wedge \texttt{level}_j = 5) \mapsto \texttt{i-ptr}_i = \perp, \texttt{m-ptr}_i = k, \texttt{level}_i = 2$

Fig. 4. Algorithm MM1

$\texttt{level}_k = 2 \wedge \texttt{m-ptr}_k = i$ holds since i is in *S5*. If it is $\texttt{level}_k = 1$, k can execute `migrate1`. If it is $\texttt{level}_k = 2$, i can execute `migrate2`. A contradiction. □

Lemma 2. *A node that points to its neighbor node by* `m-ptr` *also pointed by the neighbor's* `m-ptr` *in any terminal configuration of MM1.*

Proof. By contradiction. There is no node at level 5 in any terminal configuration and all nodes are valid. Assume that there are adjacent nodes i and j such that $\texttt{m-ptr}_i = j \wedge \texttt{m-ptr}_j \neq i$. A node i is in *S2a* since validity *S2b*(i), *S3*(i) or *S4*(i) do not hold. A node j is at level 1 and $\texttt{i-ptr}_j = i$ from *S2a*(i). Since i is in *S2a* and j is $\texttt{level}_j = 1 \wedge \texttt{i-ptr}_j = i$, j can execute `match3`. A contradiction. □

Lemma 3. *There are no two nodes i and j such that* $\texttt{level}_i = 1$, $\texttt{level}_j = 3$ *or* 4, $\texttt{i-ptr}_i = j$ *and* $\texttt{i-ptr}_j = i$ *in any termination configuration of MM1 for any graphs without a cycle of length of a multiple of 3.*

Proof. By contradiction. There is no node at level 5 in any terminal configuration and all nodes are valid. Assume that there are adjacent nodes i and j such that $\texttt{level}_i = 1$, $\texttt{level}_j = 3$ or 4, $\texttt{i-ptr}_i = j$, and $\texttt{i-ptr}_j = i$. If $\texttt{level}_j = 3$, j can execute `proceed2` since j is in *S3*.

Consider the case of $\texttt{level}_j = 4$. There is a node $k \in N(j)$ such that $\texttt{level}_k = 2$ or 3 or 4, $\texttt{m-ptr}_j = k, \texttt{i-ptr}_k \neq \perp$. Node k can execute `proceed1` if $\texttt{level}_k = 2$ and j can execute `proceed3` if $\texttt{level}_k = 4$. Hence $\texttt{level}_k$ is limited to 3. Therefore, there is a node $l \in N(k)$ such that $\texttt{i-ptr}_k = l$ and $\texttt{level}_l = 1$. Node l satisfies $\texttt{i-ptr}_l \neq k$ because it is in a terminal configuration. Therefore, there is a node $m \in N(l)$ such that $\texttt{i-ptr}_l = m$ and $\texttt{level}_m = 4$. Repeating the above observation, we can show there is an infinite sequence of nodes at levels $1, 4, 3, 1, 4, 3, \cdots$. However, there is no such a sequence since there is no cycle of length of a multiple of 3. A contradiction. □

Theorem 1. *A maximal matching is constructed in any terminal configuration of MM1 for any graphs without a cycle of length of a multiple of 3.*

Proof. By contradiction. There is no node at level 5 in any terminal configuration and all nodes are valid. Assume that a matching is not maximal in some terminal configuration. There are adjacent nodes i and j at level 1 by the assumption and Lemma 2.

If a node i or j is in *S1a*, it can execute `invite1`. Therefore, both nodes are in *S1b* (Observation 1). Let k be a node pointed by $\texttt{i-ptr}_i$. The level of k is not 5 by Lemma 1.

In case of $\texttt{level}_k = 1$, k is in *S1b* by Observation 1. Let x be a node pointed by $\texttt{i-ptr}_k$. A node k can execute `match2` to make a match with i if $\texttt{level}_x \neq 4$. Therefore, $\texttt{level}_x = 4$ and this implies $\texttt{i-ptr}_x \neq k$ by Lemma 3, and k can execute `cancel1`. In case of $\texttt{level}_k = 2$, k can execute `invite2` if k is in *S2a*. Node i can execute `cancel1` if k is in *S2b* since $\texttt{m-ptr}_k \neq i$ by Lemma 2. If $\texttt{level}_k = 3$ or 4, i can execute `cancel1` since $\texttt{i-ptr}_k \neq i$ by Lemma 3. A contradiction. □

Theorem 2. *A 1-maximal matching is constructed in any terminal configuration of MM1 for any graphs without a cycle of length of a multiple of 3.*

Proof. By contradiction. Assume that a matching is not 1-maximal in some terminal configuration. Since it is terminal, a maximal matching is constructed by Theorem 1. Therefore, there are matching nodes i and j and both have neighbors at level 1 from Lemma 2.

Both i and j are at level 2 or higher since they are matching. They are not in *S2a* since they have level 1 neighbors and can execute `invite1` if they are in *S2a*, or not at level 5 by Lemma 1. Since i and j are in *S2b*, *S3* or *S4*, both nodes point to some neighbor by `i-ptr`, and the neighbors are at level 1. That is because, i or j can execute `cancel2` in *S2b*, `cancel3` in *S3* and `reset2` in *S4* if it points to a node at level 2 or higher.

Nodes i and j are not in *S2b* since $\texttt{i-ptr}_i \neq \perp$ and $\texttt{i-ptr}_j \neq \perp$, and therefore, they can execute `proceed1` if they are in *S2b*.

Consider the case where i or j is in *S3*. Assume i is in *S3* w.l.o.g., and let k be a level 1 node that i points to by `i-ptr`. A node k can execute `approve` if $\texttt{i-ptr}_k \neq \perp$, and node i can execute `proceed2` if $\texttt{i-ptr}_k = i$. Therefore, $\texttt{i-ptr}_k = x$ for some $x \neq i$. Since there is no adjacent level 1 nodes by Theorem 1, there is no level 5 node by Lemma 1, and `m-ptr`s point to each other between two matching nodes by Lemma 2, x is at level 2, 3, or 4, and $\texttt{m-ptr}_x \neq k$. A node x is not at level 2 since k can execute `cancel1` if x is at level 2. In case where x is at level 3 or 4, $\texttt{i-ptr}_x \neq k$ by Lemma 3, and therefore, k can also execute `cancel1`. Therefore, none of i and j is not in *S3*.

That is, both i and j are in *S4*, however, both can execute `proceed3` in this case. A contradiction. □

Lemma 4. *If a node i at level 1 is valid, that is S1a(i) or S1b(i) holds, i is valid while it is at level 1 in MM1.*

Proof. Validity functions *S1a*(i) and *S1b*(i) check only the variables of a node i. That is the validity of a node at level 1 is independent of its neighbors' states. Any move for *S1a* or *S1b* keeps the state of node valid, a valid node at level 1 is valid while it is at level 1. □

Lemma 5. *Once a node executes one of `match1`, `match2`, `match3`, `migrate1` and `migrate2`, the node never executes `reset1` or `reset2` in MM1.*

Proof. By contradiction. Assume some nodes execute resets (`reset1` or `reset2`) after executing `match1`, `match2`, `match3`, `migrate1` or `migrate2`. Let i be a node that executes such a move r of a reset first. Let m be the last move of among `match1`, `match2`, `match3`, `migrate1` and `migrate2` before the reset. Since no move except `reset1` and `reset2` brings invalid states and i already executed m, when i executes r, i is two nodes invalid. Therefore, i detects some invalidity between i and some neighbor.

Let k be a node such that $\texttt{i-ptr}_i = k$ when i executes r. If k causes the reset r, i is at level 4 or 5 at that time. When i moves to *S4* by `proceed2`, i confirms

that k's validity, $\texttt{level}_k = 1$ and $\texttt{i-ptr}_k = i$. Node k never resets while it is at level 1 by Lemma 4 and the validity between i and k is preserved. Node k may move to *S2a* by `migrate1` but never resets before r by the assumption, and therefore, the validity i and k is also preserved.

Therefore, i executes r by detecting invalidity between i and j such that $\texttt{m-ptr}_i = j$. Since m is the last chance to set `m-ptr` for i, i sets $\texttt{m-ptr}_i = j$ by m. When i executes m, j is in *S1b*, *S2a*, or *S5*.

In case of *S1b*, when i executes m, i confirms j's validity and $\texttt{i-ptr}_j = i$. Node j is valid while it is at level 1 by Lemma 4. Node i moves to *S2b* after j sets $\texttt{m-ptr}_j = i$ and moves to *S2a* by `match2` or `match3`. Therefore, while j is at level 1, $\texttt{i-ptr}_j = i$ always holds and therefore i cannot reset. After j moves to level 2 by `match2` or `match3`, j does not reset before r from the assumption. Therefore, the validity between i and j is preserved until r.

In case of *S5*, that is i migrates to j, when i executes m, i confirms $\texttt{i-ptr}_j = i$. Since the validity of a node in *S5* only depends on its state and a state of a node pointing to by `i-ptr`, j is valid if the validity between i and j is preserved. Since i does not reset between m and r, the validity is preserved while j is in *S5*. After j moves to level 2 by `migrate2`, j does not reset before r from the assumption. Therefore, the validity between i and j is preserved until r.

In case of *S2a*, i confirms the validity between i and j and $\texttt{m-ptr}_j = i$ when i executes m. Since j is in *S2a*, $\texttt{i-ptr}_j$ does not point to any node. Therefore, even if j points to some node by `i-ptr` after m, the validity between j and the pointed node is preserved like between i and k. Therefore j is valid if the validity between i and j is preserved while $\texttt{m-ptr}_j = i$ and $\texttt{level}_j \leq 4$ (When j moves to *S5*, it does not take care of i). Since i does not reset between m and r, the validity is preserved. □

We say a move is a *progress move* if it is by `match1`, `match2`, `match3`, or `migrate1`. A level of node changes from 1 to 2 by a progress move.

Lemma 6. *Each node resets at most once in MM1.*

Proof. Once a node executes `reset1` or `reset2`, it moves to *S1a*. The node never resets while it is at level 1 from Lemma 4. The node executes a progress move to move to level 2, and never resets after that by Lemma 5. □

Lemma 7. *Each node execute a progress move at most once in MM1.*

Proof. A progress move changes levels of a node from 1 to 2, and a node never resets if it executes a progress move by Lemma 5. That is the node never goes back to level 1. Therefore, once a node executes a progress move it never executes a progress move again. □

Lemma 8. *In MM1, `cancel1`, `cancel2`, `cancel3` and `cancel4` are executed $O(e)$ times.*

Proof. In MM1, a node i executes a cancel (`cancel1`, `cancel2`, `cancel3` or `cancel4`) when it is initially possible, some neighbor node executed a cancel, or some neighbor node executed a progress move.

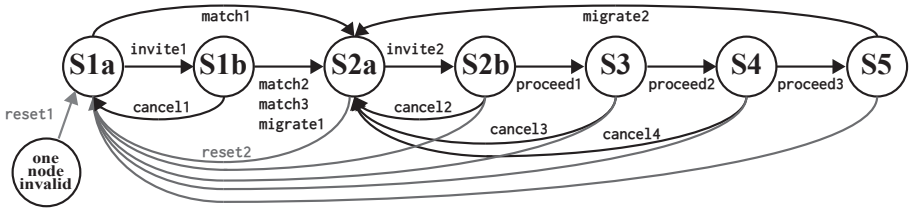


Fig. 5. Transitions of stages

Consider that some node j executes a progress move that changes a stage of j to *S2a*. Nodes that point to j by `i-ptr` will execute a cancel as follows. If such a node k is in *S1b*, k will execute `cancel1`, and if such a node k is in *S2b* or *S3*, k will execute `cancel2` or `cancel3`.

If some node executes `cancel2` or `cancel3`, it causes more cancels. If there is an adjacent node x and trying to increase matches, it will also cancels by `cancel3` or `cancel4`. That cancel may further causes one more cancel. If x already invited some node y to migrate to x, y will execute `cancel1`.

Now we classify cancels with *direct cancels* and *indirect cancels*. The direct cancel is a cancel caused by some progress move or its initial state. The indirect cancel is a cancel caused by a cancel of its neighbor.

From the above observation, any cancel causes at most two indirect cancels. Let deg_j be the degree of j. There are at most deg_j nodes that execute a cancel due to the progress move of j. From Lemma 7, j executes a progress move at most once, and therefore there are at most $\Sigma_{i \in V} deg_i = e$ direct cancels caused by progress moves. Moreover, there are at most n direct cancels caused by initial states. Therefore, the total number of moves by cancels are $O(e)$. □

Lemma 9. *In MM1,* `migrate2` *is executed* $O(n)$ *times.*

Proof. Let m_1 and m_2 be two consecutive moves by `migrate2` of a node i. The node i moves to *S2a* by m_1 and then invites some neighbor node j at level 1 to migrate to i. Then, node j executes `migrate1` that points to i by `m-ptr`. That is, there is a move by `migrate1` that points to i between two consecutive moves by `migrate2` of node i. Therefore, the total number of moves by `migrate2` $\leq$ the total number of moves by `migrate1` $+n$. From Lemma 7, it is bounded by $O(n)$. □

Theorem 3. *MM1 is silent and takes* $O(e)$ *moves to construct 1-maximal matching for any graphs without a cycle of length of a multiple of 3.*

Proof. Fig. 5 shows stage transition in MM1. In MM1, each node moves to a higher stage from the current stage in the order of *S1a*, *S1b*, *S2a*, *S2b*, *S3*, *S4* and *S5* except `reset1`, `reset2`, `cancel1`, `cancel2`, `cancel3`, `cancel4` and `migrate2`. Therefore, if a node does not execute these actions, the number of moves is at most 6.

Let R_i, C_i and M_i be the numbers of moves of a node i by reset (`reset1` or `reset2`), cancel (`cancel1`, `cancel2`, `cancel3` or `cancel4`), and `migrate2`. Let MOV_i denote the total number of moves of a node i. From the observation, it is bounded as follows.

$$MOV_i \leq 7(R_i + C_i + M_i + 1)$$

From Lemmas 6, 8 and 9, we have

$$\Sigma_{i \in V} R_i = O(n), \Sigma_{i \in V} C_i = O(e), \text{and } \Sigma_{i \in V} M_i = O(n).$$

Therefore, the total number of moves in MM1 can be derived as follows.

$$\Sigma_{i \in V} MOV_i \leq 7(\Sigma_{i \in V} R_i + \Sigma_{i \in V} C_i + \Sigma_{i \in V} M_i + \Sigma_{i \in V} 1) = O(e)$$

Since each node always takes a finite number of moves, MM1 always reaches a terminal configuration where 1-maximal matching is constructed by Theorem 2. This also implies MM1 is silent. □

5 Conclusion

We proposed a 1-maximal matching algorithm MM1 that is silent and works for any anonymous networks without a cycle of a length of a multiple of 3 under a central unfair daemon. The time complexity of MM1 is $O(e)$ moves. Therefore, it is $O(n)$ moves for trees or rings whose length is not a multiple of 3. We had a significant improvement from Goddard et al.[6] that is also an anonymous 1-maximal matching algorithm but works for only trees or rings which length is not a multiple of 3 and the time complexity is $O(n^4)$.

References

1. Blair, J.R., Manne, F.: Efficient self-stabilizing algorithms for tree networks. In: Proceedings. 23rd International Conference on Distributed Computing Systems. pp. 20–26. IEEE (2003)
2. Blair, J., Hedetniemi, S., Hedetniemi, S., Jacobs, D.: Self-stabilizing maximum matchings. Congressus Numerantium pp. 151–160 (2001)
3. Chattopadhyay, S., Higham, L., Seyffarth, K.: Dynamic and self-stabilizing distributed matching. In: Proceedings of the twenty-first annual symposium on Principles of distributed computing. pp. 290–297. ACM (2002)
4. Datta, A.K., Larmore, L.L.: Leader election and centers and medians in tree networks. In: Stabilization, Safety, and Security of Distributed Systems, pp. 113–132. Springer (2013)
5. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Commun. ACM 17(11), 643–644 (Nov 1974), http://doi.acm.org/10.1145/361179.361202
6. Goddard, W., Hedetniemi, S.T., Shi, Z., et al.: An anonymous self-stabilizing algorithm for 1-maximal matching in trees. In: Proc. International Conference on Parallel and Distributed Processing Techniques and Applications. pp. 797–803 (2006)

7. Guellati, N., Kheddouci, H.: A survey on self-stabilizing algorithms for independence, domination, coloring, and matching in graphs. Journal of Parallel and Distributed Computing 70(4), 406–415 (2010)
8. Hedetniemi, S.T., Jacobs, D.P., Srimani, P.K.: Maximal matching stabilizes in time $O(m)$. Information Processing Letters 80(5), 221–223 (2001)
9. Hsu, S.C., Huang, S.T.: A self-stabilizing algorithm for maximal matching. Information Processing Letters 43(2), 77–81 (1992)
10. Karaata, M.H., Saleh, K.A.: Distributed self-stabilizing algorithm for finding maximum matching. Comput Syst Sci Eng 15(3), 175–180 (2000)
11. Kimoto, M., Tsuchiya, T., Kikuno, T.: The time complexity of Hsu and Huang's self-stabilizing maximal matching algorithm. IEEE Trans. Infrmation and Systems E93-D(10), 2850–2853 (2010)
12. Manne, F., Mjelde, M., Pilard, L., Tixeuil, S.: A self-stabilizing 2/3-approximation algorithm for the maximum matching problem. Theoretical Computer Science 412(40), 5515–5526 (2011)
13. Tel, G.: Introduction to distributed algorithms. Cambridge university press (2000)

均一で密なユニットディスクグラフにおける局所情報に基づく経路の自己構成手法

東優樹　大下福仁　角川裕次　増澤利光
大阪大学 大学院情報科学研究科

概要　本稿では，均一で密なユニットディスクグラフにおいて，任意のホップ数冗長経路から最短経路の高々定数倍の経路を構築する手法について考察する．均一で密とは，ノードの通信可能範囲を 8 等分の領域に分割したとき各領域に少なくとも 1 つのノードが存在することを意味する．送信ノードから宛先ノードへの経路において宛先ノードから遠ざかる移動がなければ冗長な経路が少ないことから，8 等分された領域のうち連続した 2 つの領域のみで経路を構築することを考える．各ノードが局所情報のみを利用し，自律的に経路を変更することによって，目的の経路を構築する手法を紹介する．また，提案アルゴリズムによって得られた経路の最短経路に対する近似率について考察する．

1　はじめに

近年，無線デバイス（ノード）の普及により，分散システムにおいて無線ネットワークは一般的かつ重要となっており，例として MANET（モバイルアドホックネットワーク）や WSN（ワイヤレスセンサネットワーク）がある．無線ネットワークでは，ノードは二次元平面に配置されており，通信範囲内のノードとのみ直接通信可能である．宛先ノードが通信範囲外のときは，他のノードが宛先ノードへのメッセージを中継する．通信経路が長いほどメッセージ送信に伴う通信遅延や消費エネルギーが増加する．効率的なマルチホップルーティングを実現するために，数々のルーティングプロトコルが提案されている [1]．

送信ノードから宛先ノードへの最短経路を構築する手法として，仮想グリッドネットワーク上でルーティングを行う手法が提案されている [2]．しかし，仮想グリッドネットワークの構築やノードの移動時の処理にはコストがかかる．本稿では，ユニットディスクグラフ上で任意のホップ数冗長経路から最短経路の高々定数倍の経路を構築するアルゴリズムについて考察する．均一で密とは，ノードの通信可能範囲を 8 等分の領域に分割したとき各領域に少なくとも 1 つのノードが存在することを意味する．

経路の冗長な部分を減らすには，送信ノードから宛先ノードへの経路において宛先ノードから遠ざかる方向への経路が存在しなければ良い．そこで，送信ノードから宛先ノードへ 8 分割された領域のうち連続した 2 つの領域のみで到達する経路を構築する手法を考える．本稿では，各ノードが局所情報に基づいて，経路切断が発生しないように経路を局所的に変更するという動作を繰り返すことにより，与えられた通信経路を最短経路の高々定数倍の経路に変換するアルゴリズムの設計への取り組みを紹介する．

2　諸定義

本稿では，ユニットディスクグラフ $G = (V, E)$ での送信ノード v_s から宛先ノード v_t へのメッセージ転送を考える．ここで，V はノードの集合，E は隣接ノード間を結ぶ通信リンクの集合を表す．ネットワークのノード数を $n = |V|$ とし，各ノードの識別子を $v_i (0 \leq i \leq n-1)$ とする．また，ノード v_i, v_j 間の通信リンク $(v_i, v_j) \in E$ は $v_i, v_j \in V$ に対し，$|v_i - v_j| < 1$ が成り立つとき，かつそのときのみ存在する．ただし，$|v_i - v_j|$ は v_i と v_j のユークリッド距離を表す．また，ユニットディスクグラフ G に対し，以下の 2 つの制限を加える．

1. 各ノードは 4 ホップ先までのノードの情報を知っている．
2. 各ノードは距離 $\frac{1}{\sqrt{2}}$ 以下に 8 方向の辺を持つ（図 1）．つまり，0..7 の各領域に対して少なくとも 1 つのノードが存在する．また，全ノードが同じ方向感覚を有するものとする．

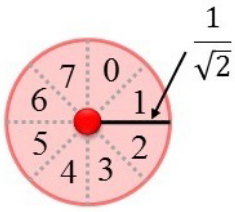


図 1: ノードの方向感覚

送信ノード v_s から宛先ノード v_t までの通信経路を $P = (v_s = p_0, p_1, ..., p_m = v_t)$ と表し，辺 (p_i, p_{i+1}) の方向 d_i を p_i から見た p_{i+1} の存在する領域とする．

定義 1 (通信経路準最適化問題)**.** v_s, v_t をユニットディスクグラフ G の異なるノードとしたとき，任意の $v_s - v_t$ 経路が与えられたら，最短 $v_s - v_t$ 経路の高々 α 倍以下の $v_S - v_t$ 経路を構築する．このとき，α は最短 $v_s - v_t$ 経路に対する近似率である．　□

$v_s - v_t$ 経路 P 上のノード v_i が所持する情報を以下に示す．v_i が P 上に k 回出現するとき，v_i は k 個の組 $(pre_1, next_1), (pre_2, next_2), ..., (pre_k, next_k)$ を所持する．P 上の p_i を v_i の j 番目の出現とすると，v_i が保持する j 番目の組 $(pre_j, next_j)$ は (p_{i-1}, p_{i+1}) である．

3　提案アルゴリズム

提案アルゴリズムは 3 つの操作 (操作 1，操作 2，操作 3) から構成され，これらの操作を繰り返すことで任意の経路を準最適経路へ変換する．

操作 1 局所的に冗長な部分を削除

操作 2 局所的に連続した 3 方向以上の経路を連続した 2 方向以下の経路に変換

操作 3 局所的に連続した 3 方向以上の経路を検出できるよう経路を変換

提案アルゴリズムは以下のように動作する．

1. 経路上の各ノード p_i は周囲の経路の情報を取得し，p_i において局所変更が可能かどうかを調べる．
2. p_i において局所変更が可能であれば適用し，1. に戻る．そうでなければそのまま 1. の処理に戻る．ただし，経路切断の発生を回避するため，局所変更適用時は近隣ノードと強調して，実際に適用するかどうか決定する．

Algorithm 1 提案アルゴリズム:for ノード p_i

```
loop
  近隣 1 ホップにあるノードの所持する経路情報を取得し，p_i において局所変更が可能か調べる．;
if p_i において局所変更が可能 then
  近隣ノードが局所変更を実行可能か調べる ;
  if 局所変更を実行しても経路切断が発生しない then
    実行可能な局所変更を実行 ;
  goto 2 ;
else
  goto 2 ;
```

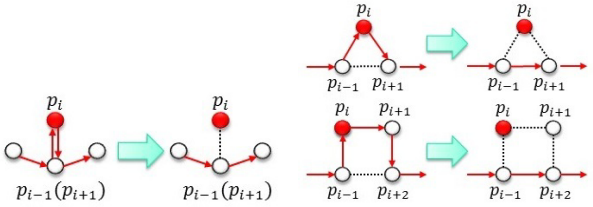
3.1　操作 1

操作 1 では，局所的にホップ数が冗長な経路に対して冗長部分を削除する変更を行う．操作 1 の局所変更を以下に示す．

定義 2 (冗長部分の除去)**.**

update1: 辺の往復の削除（図 2 (a)）
辺 $(p_{i-1},p_i),(p_i,p_{i+1})$ について $p_{i-1}=p_{i+1}$ ならば，$(p_{i-1},p_i),(p_i,p_{i+1})$ を削除する．ただし，$d_{i-2},d_{i-1},d_i,di+1$ が連続した 2 方向のみから成る経路の場合は適用しない．

update2: ホップ数の削減（図 2 (b)）
辺 $(p_{i-1},p_i),(p_i,p_{i+1})$ について $(p_{i-1},p_{i+1})\in E$ ならば，$(p_{i-1},p_i),(p_i,p_{i+1})$ を削除し，(p_{i-1},p_{i+1}) を追加．また，辺 $(p_{i-1},p_i),(p_i,p_{i+1}),(p_{i+1},p_{i+2})$ について $(p_{i-1},p_{i+2})\in E$ ならば，$(p_{i-1},p_i),(p_i,p_{i+1}),(p_{i+1},p_{i+2})$ を削除し，(p_{i-1},p_{i+2}) を追加する．



(a)update1　(b)update2

図 2: 冗長部分の削除

3.2　操作 2

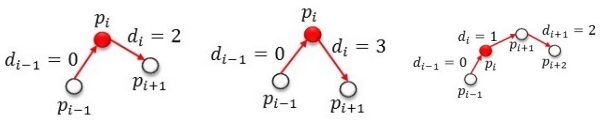
操作 2 では，局所的に連続した 3 方向以上の経路を連続した 2 方向以下の経路に変更する．各 update では，変更後の経路は連続した 2 方向以下の経路を選択するものとする．操作 2 の局所変更の適用条件を以下に示す．

定義 3 (方向の削減)**.**

update3: *1* つ飛ばし（図 3 (c)）
辺 $(p_{i-1},p_i),(p_i,p_{i+1})$ について，d_{i-1} と d_i が 1 つ飛ばし (連続する 3 方向の経路)．つまり，$|d_{i-1}-d_i|=2$ もしくは $|d_{i-1}-d_i|=6$.

update4: *2* つ飛ばし（図 3 (d)）
辺 $(p_{i-1},p_i),(p_i,p_{i+1})$ について，d_{i-1} と d_i が 2 つ飛ばし (連続する 4 方向の経路)．つまり，$|d_{i-1}-d_i|=3$ もしくは $|d_{i-1}-d_i|=5$.

update5: 折り返しが存在（図 3 (e)）
辺 $(p_{i-1},p_i),(p_i,p_{i+1}),(p_{i+1},p_{i+2})$ について，d_{i-1} ,d_i,d_{i+1} は連続しているが d_{i-1} と d_{i+1} が反対方向に進んでいる．つまり，$|d_{i-1}-d_i|\%6=1$ かつ $|d_i-d_{i+1}|\%6=1$ かつ $|di+1-d_{i+2}|\%4=2$.



(c)update3　(d)update4　(e)update5

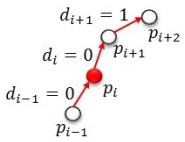
図 3: 3 方向以上の経路を削除

3.3　操作 3

操作 3 では，局所的に連続した 3 方向以上の経路を検出できるように変更する．操作 3 の局所変更を以下に示す．

定義 4 (3 方向以上の経路の検出).

update6: 同じ方向の後に隣接方向に移動（図 4 (f)）
辺 $(p_{i-1}, p_i), (p_i, p_{i+1}), (p_{i+1}, p_{i+2})$ について，d_{i-1} , d_i の方向が同じかつ d_i, d_{i+1} が隣接した方向である．つまり，$d_{i-1} = d_i$かつ $|d_i - d_{i+1}|\%6 = 1$.



(f)update6

図 4: 3 方向の検出

連続した 3 方向以上の経路の検出

update6 を繰り返し実行することで局所的に連続した 3 方向以上の経路を検出することができる（図 5）.

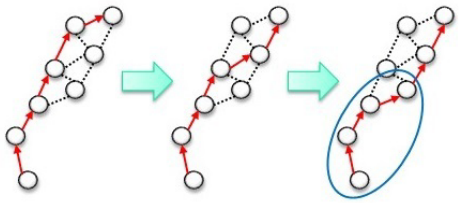


図 5: 局所的に 3 方向を検出する様子

3.4 局所変更の優先度

経路 P 上のノード p において複数の局所変更が実行可能な場合が起こりうる．その場合は以下の優先度に従い優先度の高い局所変更を実行する．

定義 5 (局所変更の優先度). $update1 > update2 > update3 > update4 > update6 > update5$ □

また，連続したノードで局所変更が実行されると経路が非連結になる可能性があるため経路の前後関係での局所変更の優先度を次のように定める，ノード p_i は前後 2 ホップのノード（$p_{i-2}, p_{i-1}, p_{i+1}, pi + 2$）で自身より優先度の高い変更が実行可能ならば局所変更を実行しない．また前 2 ホップのノード（p_{i-2}, p_{i-1}）で自身と同じ優先度の変更が実行可能ならば局所変更を実行しない．

4 近似率

提案アルゴリズムにより最終的に得られる連続した 2 方向のみの準最短経路 P の近似率について考察する．経路を連続した 2 方向のみの経路に変更することで，図 6 のように経路の存在する範囲を限定することができる．ここで，$|v_s - v_t| = r$，v_s と v_t のなす角を θ と

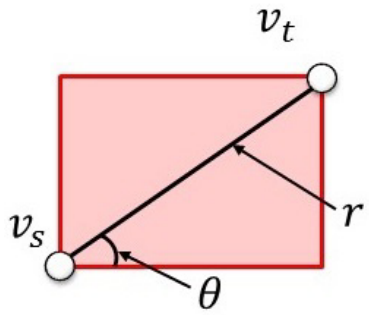


図 6: 経路の存在範囲

する．このとき，経路 P が宛先ノード v_t から遠ざかる方向に移動しないこと，少なくとも 2 ホップで距離 1 以上 v_t に近づくことを考慮するとホップ数と距離による近似率は表 1 のようになる.

表 1: 近似率

	最短	最長	近似率
ホップ数	r	$2\sqrt{2}r\sin(\theta + \frac{\pi}{4})$	$2\sqrt{2}$
距離	r	$\sqrt{2}r\sin(\theta + \frac{\pi}{4})$	$\sqrt{2}$

5 まとめと今後の課題

本稿では，均一で密なユニットディスクグラフにおいて，送信ノードから宛先ノードへの通信経路が与えられたとき，各ノードが局所的な情報のみを用いて局所的に通信経路を変更することにてよって，最短経路の高々定数倍となる経路を構築する手法を提案した．提案アルゴリズムによって得られる経路の最短経路に対する近似率はホップ数で $2\sqrt{2}$，距離で $\sqrt{2}$ となる.

今後の課題としては，収束時間の評価と 8 方向のうちに辺が存在しない方向が存在する場合に対するアルゴリズムの考案が挙げられる.

参考文献

[1] W. Dargie and C. Poellabauer. *Fundamentals of Wireless Sensor Networks: Theory and Practice.* Wireless Communications and Mobile Computing. Wiley, 2010.

[2] Shusuke Takatsu, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. *Zigzag: Local-Information-Based Self-Optimizing Routing in Virtual Grid Networks.* In ICDCS, pp. 357–368, 2013.

局所的トリガ数え上げ問題に対する分散アルゴリズム

安達駿，大下福仁，角川裕次，増澤利光
大阪大学情報科学科研究科

概要

ネットワークで接続された多数のプロセッサから構成される分散システムを利用した，外部環境モニタリングを考える．プロセッサに付随したセンサが降雨や渋滞などのイベントを検知すると，プロセッサでトリガが発生する．このとき，システム全体で発生したトリガの総数がある閾値に達したことを検出する問題はトリガ数え上げ問題とよばれ，これまでにいくつかの分散アルゴリズムが提案されている．本報告では，各プロセッサの近傍に注目し，近傍プロセッサで発生したトリガの総数がある閾値に達したことを検出する問題を局所的トリガ数え上げ問題として定義し，その分散アルゴリズムについて考察する．特に，近傍として距離 ρ 以内にあるプロセッサの集合（ρ-近傍とよぶ）を対象とする ρ-近傍トリガ数え上げ問題について考察する．本報告では (1) 局所分散制御に基づく解法，(2) 集中型制御に基づく解法，(3) クラスタ内制御に基づく解法（(1)，(2) の折衷手法）の 3 つの分散アルゴリズムを提案し，それぞれの通信計算量について考察する．

1　はじめに

1.1　研究背景

ネットワークで接続された n 個の**プロセッサ**から構成される分散システムを用いた分散モニタリングについて考える．分散モニタリングとは，プロセッサに付随したセンサを利用し，交通量や降水量などについての環境モニタリングを実現するシステムである．分散モニタリングにおいて，”交通量”や”降水量”，あるいは，データネットワークにおける”通信量”や”ログイン中のユーザ数”などが，ある一定数を超えたことを検出する問題は，トリガ数え上げ問題とよばれ，これまでにいくつかの研究成果が示されている [1, 2, 3, 4, 5]．トリガ数え上げ問題は，任意のタイミングで任意のプロセッサでトリガが発生しうる状況で，発生したトリガの総数が，ある与えられた値 w（**検出トリガ数**とよぶ）に達したことをいずれかのプロセッサが検出する問題として定式化されている．

トリガ数え上げ問題について，文献 [2] では，総メッセージ数が $O(n \log w)$，1 プロセッサの最大受信メッセージ数が $O(n \log w)$ となる集中型アルゴリズムと，総メッセージ数が $O(n \log n \log w)$，最大受信メッセージ数が $O(n \log n \log w)$ となる木構造に基づく分散アルゴリズムを提案している．

これらの分散アルゴリズムでは，あるプロ

セッサがシステム全体を連携させる代表としての役割を果たし，その他のプロセッサは代表プロセッサに従うという集中型制御に基づいている．一方，分散システムとして，センサネットワークを想定する場合，各プロセッサは限られた電力で動作すると想定することが多く，メッセージ数を減らし，各プロセッサのライフタイムを延ばすための工夫が必要とされている [3, 4, 5]．例えば，文献 [3] では，ネットワークにレイヤと呼ばれる構造を導入することで，総メッセージ数が $O(n \log n \log w)$，最大受信メッセージ数が高い確率で $O(\log n \log w)$ となる乱択アルゴリズムを実現している．

1.2 本報告の目的と結果

トリガ数え上げ問題では，分散システム全体で発生したトリガの総数が一定数を超えたことを検出することを目的とする．しかし，交通量や降水量を対象とする場合，各プロセッサは自身の近傍で発生したトリガを数え上げたいことがある．そこで本報告では，この問題を**局所的トリガ数え上げ問題**として定義し，特に，あるプロセッサから距離 ρ 以内にあるプロセッサの集合（ρ-近傍とよぶ）で発生したトリガの総数が検出トリガ数に達したことを検出する **ρ-近傍トリガ数え上げ問題**について検討する．

本報告では，まず 2 章で分散システムと問題の定義を行い，3 章で (1) 局所分散制御に基づく解法を示す．これは，各プロセッサが ρ-近傍で発生したトリガを集めることで検出トリガに達したことを検出するアルゴリズム SIMPLE_COUNTING による解法である．この解法の通信計算量は，システム全体で発生するトリガ数を W，最大 ρ-次数を $\Delta^{\rho}(G)$ とすると，総メッセージ数 $O(W \cdot \Delta(G)^{\rho})$，最大受信メッセージ数 $O(w)$ となる．これらの計算量はネットワークのトポロジに依存するが，ある ρ-近傍に全てのプロセッサが含まれる（最大 ρ-次数がプロセッサ数 n となる）とき，総メッセージ数は最悪となり，$O(n \cdot W)$ となる．

次に，4 章で (2) 集中型制御に基づく解法を示す．これは，分散システムの中心のプロセッサの 1 つを**サーバ**として指定し，このサーバがシステム全体のトリガ発生状況を管理することで，あるプロセッサの ρ-近傍で発生したトリガの数が検出トリガ数に達したことを検出するアルゴリズム $SERVER\_COUNTING$ による解法である．この解法の通信計算量は，分散システムの半径を $Rad(G)$ とすると，総メッセージ数 $O(W \cdot Rad(G))$，最大受信メッセージ数 $O(W)$ となる．また分散システムのトポロジが，半径 Rad(G) が最大となるようなトポロジのとき，総メッセージ数は最悪となり，$O(n \cdot W)$ となる．

これらの解法 (1),(2) の最悪時の総メッセージ数は，いずれも $O(n \cdot W)$ となるが，既に述べたように，そのトポロジは異なる．そこで 5 章では，解法 (1),(2) の折衷手法として (3) クラスタ内制御に基づく解法を考え，最悪時の通信計算量の改善を図る．この解法では，分散システムにクラスタ構造 (クラスタの集合 $\mathcal{T}$ が分散システムの被覆となっている) を導入し、そのクラスタごとにアルゴリズム $SERVER\_COUNTING$ を適応することで検出トリガに達したことを検出する．このアルゴリズムを $CLUSTER\_COUNTING$ と

よぶ．この解法の通信計算量は，クラスタの最大半径を $Rad(\mathcal{T})$，最大クラスタ次数を $\Delta(\mathcal{T})$ とすると，総メッセージ数 $O(\Delta(\mathcal{T})\cdot Rad(\mathcal{T})\cdot W)$，最大受信メッセージ数 $O(W)$ となる．

	最悪時総メッセージ数
解法 (1)	$O(n\cdot W)$
解法 (2)	$O(n\cdot W)$
解法 (3)	$O(\kappa\cdot n^{1/\kappa}\cdot\rho\cdot W)$

表 1: 提案アルゴリズムの通信計算量

解法 (3) は被覆に関するパラメタにより計算量が異なるため，最大クラスタ次数と平均クラスタ次数の低くなる 2 種類の被覆を与え評価する．最悪時の総メッセージ数は表 1 のように $O(\kappa\cdot n^{1/\kappa}\cdot\rho\cdot W)$ なり，自由に設定可能なパラメタ κ に適当な値を決めることで解法 (1),(2) と比べ改善が可能である．

そして最後に，6 章で本報告の内容をまとめる．

2 諸定義

本章では，ネットワークシステムのモデルについて述べる．次に，ρ-近傍トリガ数え上げ問題を定義し，この問題に対する分散アルゴリズムの評価尺度について説明する．

2.1 システムモデル

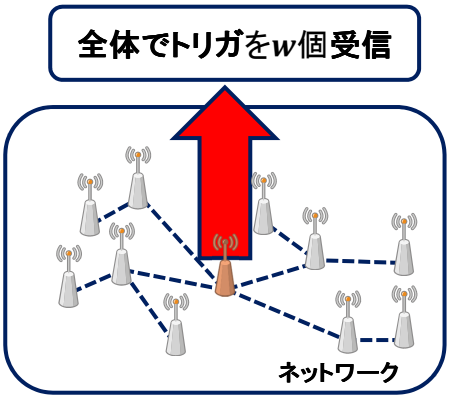
分散システム $G=(V,E)$ は n 個のプロセッサ $V=\{p_1,p_2,\cdots,p_n\}$ と，異なる 2 つのプロセッサ間の通信リンクの集合 E で構成される．各プロセッサ $p_i(1\le i\le n)$ は固有の識別子 $id(p_i)$ を有する．以降では，この $id(p_i)$ を p_i と区別せずに用いる．また，プロセッサの故障は考えないものとする．通信リンク (p_i,p_j) で接続されたプロセッサ p_i,p_j 間ではメッセージの送受信が可能であり，送信したメッセージは破損することなく有限時間内に受信プロセッサに届く．ただし，この遅延に上限がない非同期システムを想定し，リンクは長さに制限のない FIFO キューとしてモデル化する．

2.2 ρ-近傍トリガ数え上げ問題

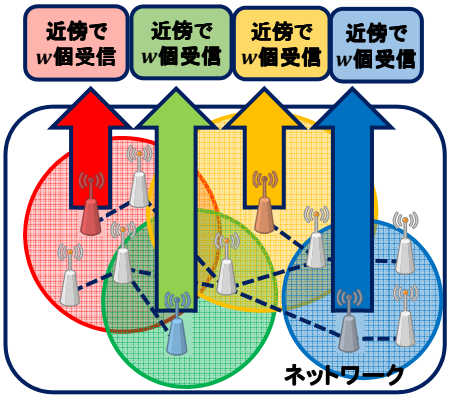
各プロセッサ $p_i(1\le i\le n)$ では，外界から任意のタイミングで**トリガ**が発生するものとする．ただし，トリガがどのプロセッサでどのようなタイミングで発生するかは，前もって決まっていない．また，同一プロセッサで複数のトリガが発生することもある．

このとき，分散システム全体で発生したトリガの総数がある閾値 w（以降，w を**検出トリガ数**とよぶ）に達したことを検出する問題が**大域的トリガ数え上げ問題** [2] である（図

1(a)). ただし w はあらかじめ指定された値とする．これに対し，本報告では分散システムの各プロセッサが自身の近傍でトリガが w 個発生したことを検出する問題を**局所的トリガ数え上げ問題**として新たに定義する（図 1(b)）．



(a) 大域的トリガ数え上げ問題



(b) 局所的トリガ数え上げ問題

図 1: 2 種類のトリガ数え上げ問題

本報告では，局所的トリガ数え上げ問題として，特に，各プロセッサ p_i に対し，距離 ρ 以内にあるプロセッサから構成される **ρ-近傍**（図 2）を考え，この ρ-近傍での発生トリガ数が検出トリガ数 w に達したことを p_i が検出する **ρ-近傍トリガ数え上げ問題**について考察し，そのアルゴリズムの提案を行う．

以下では，各プロセッサ $p_i(1 \leq i \leq n)$ について，その ρ-近傍を $S_{p_i}^{\rho} = \{p \in V | dist(p_i, p) \leq \rho\}$ と表し，$|S_{p_i}^{\rho}|$ を p_i の **ρ-次数**とよぶ．ここで，$dist(p_i, p)$ はプロセッサ p_i, p 間の距離（最短 p_i-p 経路の長さ（辺の数））を表し，$p_i \in S_{p_i}^{\rho}$ であることを注意しておく．また，$S_{p_i}^{\rho}$ は S_i^{ρ} と略記することもある．

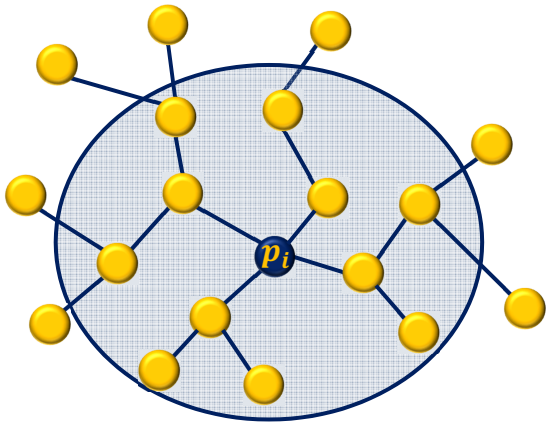


図 2: p_iの 2-近傍

2.3 アルゴリズムの評価尺度

メッセージの送受信には電力が消費される．しかし，センサネットワークなどでは各プロセッサの消費可能電力が限られていることが多いため，送受信するメッセージ数が多いと，ノードの寿命を縮めることになる．そこでメッセージ数に関する次の 2 つの評価尺度を導入し，アルゴリズムの評価を行う．

- **総メッセージ数**：
  分散システム全体で交換されるメッセージの総数．
- **最大受信メッセージ数**：
  システム中の各プロセッサが受信するメッセージ数の最大数．

本報告でのアルゴリズムは，各プロセッサが，トリガの受信時もしくはメッセージの受信時に限られるイベントによりメッセージを送信するイベント駆動型のアルゴリズムである．そのため，メッセージの送信数に関する評価尺度は導入していない．

3 ρ-近傍トリガ数え上げに対する素朴な局所分散型解法

本章では，ρ-近傍トリガ数え上げ問題を解く分散アルゴリズム SIMPLE_COUNTING を示し，その評価を行う．

3.1 アルゴリズム SIMPLE_COUNTING

本アルゴリズムでは，各プロセッサ $p_i(1 \leq i \leq n)$ はトリガが発生すると，このことを $p_i \in S_j^\rho$ を満たす全てのプロセッサ p_j に通知する．各プロセッサは，この通知により自身の ρ-近傍で受信されたトリガの数を検知し，それが w に達したときそのことを検出できる．

このとき，$p_i \in S_j^\rho$ を満たす p_j の集合は S_i^ρ に一致する（つまり，$S_i^\rho = \{p_j \in V | p_i \in S_j^\rho\}$）．よって，$p_i$ でトリガが発生したことの通知は，S_i^ρ の各プロセッサに対して行えばよい．そこで本アルゴリズムでは，各プロセッサ p_i に対し，その ρ-近傍 $S_i^\rho(1 \leq i \leq n)$ の p_i を根とする **幅優先全域木 T_i^ρ** があらかじめ構成されているものとする．このため，各プロセッサ p_i は $|S_i^\rho|$ 個の全域木に含まれることになるが，p_i はこれら $|S_i^\rho|$ 個の全域木の情報（子プロセッサの集合）を保持しているものとする．各プロセッサ p_i は，トリガが発生すると，この全域木 T_i^ρ を用いて S_i^ρ の各プロセッサにトリガの発生を通知する．以降では，各プロセッサの動作について説明を行う．

各プロセッサ p_i は，ρ-近傍 S_i^ρ で発生したトリガ数を記録する **カウンタ $C(p_i)$**（初期値は 0）を保持している．p_i でトリガが発生した場合，まず自身のカウンタ $C(p_i)$ をインクリメントする．そして p_i の ρ-近傍 S_i^ρ に属するすべ

てのプロセッサに，全域木 T_i^ρ を用いて，トリガの発生を通知する．これは T_i^ρ を用いて，S_i^ρ の全プロセッサにメッセージ $\boldsymbol{TRIGGER_{p_i}}$ を配信することで行う（図 3）．

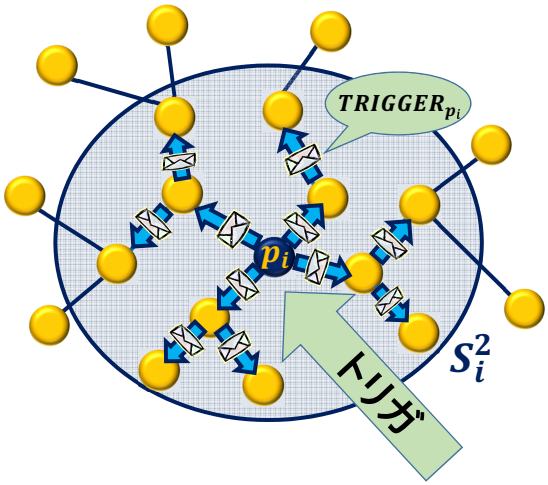


図 3: プロセッサ p_i のトリガ発生時の動作（$\rho = 2$）

このメッセージ $TRIGGER_{p_i}$ を受信したプロセッサ p_j は，自分のカウンタ $C(p_j)$ をインクリメントし，全域木 T_i^ρ に従ってメッセージを転送する．そしてカウンタの値が w に達したプロセッサは，このことを検出してアルゴリズムを終了する．

最後に，本手法の疑似コードを Algorithm1 に示す．

3.2 アルゴリズムの評価

まず総メッセージ数についての評価を行う．このアルゴリズムでは，トリガが発生したプロセッサ p_i は，自身の ρ-近傍 S_i^ρ に属する全てのプロセッサにメッセージ $TRIGGER_{p_i}$ を配信する．これは，S_i^ρ の全域木 T_i^ρ を用いて行うので，これに要するメッセージ数は $|S_i^\rho| - 1$ である．よって，$\Delta(G)^\rho = max\{\ |S_i^\rho|\ |p_i \in V\}$ を分散システム G の最大 ρ-次数，W を分散システム全体で発生したトリガ数とすれば，総メッセージ数は $O(W \cdot \Delta(G)^\rho)$ となる．

次に，最大受信メッセージ数について評価を行う．プロセッサ p_i が受信するメッセージは全て p_iのρ-近傍 S_i^ρ で発生したトリガである．従って，w 個のトリガが発生した時点でアルゴリズムは停止するので，最大受信メッセージ数は高々w である．これより，以下の定理が導かれる．

定理 1

最大 ρ-次数が $\Delta(G)^\rho$ の分散システム G において，分散アルゴリズム SIMPLE_COUNTING は総メッセージ数 $O(W \cdot \Delta(G)^\rho)$，最大受信メッセージ数 $O(w)$ で ρ-近傍トリガ数え上げ問題を解く．ただし，W はシステム全体で発生したトリガ数，w は検出トリガ数とする．

4 ρ-近傍トリガ数え上げに対する集中型解法

本章では，ρ-近傍トリガ数え上げ問題を解く分散アルゴリズム SERVER_COUNTING を示し，その評価を行う．

4.1 アルゴリズム SERVER_COUNTING

まず，分散システム G の**半径**と**中心**を定義する．

定義 1 分散システム $G = (V, E)$ におけるプロセッサ $v \in V$ の離心数 $Ecc(v, G)$ を以下のように定義する．

$$Ecc(v, G) = \max_{w \in V}\{dist_G(v, w)\}$$

この離心率 $Ecc(v, G)$ が最も小さくなるようなプロセッサ u を分散システム G の中心と呼び，$Ecc(u, G)$ を分散システム G の半径 $Rad(G)$ とよぶ．

$$Rad(G) = \min_{v \in V}\{Ecc(v, G)\}$$

この中心のプロセッサの 1 つを**サーバ s** として指定する．サーバ s は分散システム G のトポロジを知っており，発生したトリガの数を ρ-近傍 $S_i^\rho (1 \leq i \leq n)$ ごとに管理している．これらの情報から，サーバ s は ρ-近傍 S_i^ρ で発生したトリガの数が w に達したことを検知したとき，プロセッサ p_i に通知する．このようにして，ρ-近傍トリガ数え上げ問題を解く，というのがアルゴリズム SERVER_COUNTING の概要である．以降，アルゴリズムについて詳細に述べていく．

まず分散システム G に対し，サーバ s を根とする幅優先全域木 T_G があらかじめ構成されている．各プロセッサ p_i は，この全域木 T_G の情報（親プロセッサ，子プロセッサの集合）を保持している．サーバ s は，これに加え，分散システム G のトポロジ情報（プロセッサの識別子を含む）も保持する．また，3 章では各プロセッサがそれぞれカウンタを保持していたが，本章ではサーバ s が各 ρ-近傍クラスタ S_i に対し，カウンタ $C(p_i)$ を保持している．

次に，各プロセッサの動作について説明する．本アルゴリズムでは，サーバ s とそれ以外のプロセッサ p_i では動作が異なるため，それぞれ説明を行う．

- サーバ s 以外のプロセッサ p_i の動作（図 4）
  - トリガが発生した場合
    トリガが発生したというメッセージをサーバ s に全域木 T_G を用いて送信する．このメッセージには，サーバ s が送り主が分かるように p_i の識別子をメッセージに含める．以降，このメッセージを $\boldsymbol{TRIGGER(i)}$ とよぶ．
  - メッセージを受信した場合
    p_i が受信するメッセージは，トリガが発生した他のプロセッサ $p_j(1 \leq j \leq n)$ から送信された $TRIGGER(j)$ メッセージである．よってこのメッセージを受信した場合，全域木 T_G における親プロセッサに $TRIGGER(j)$ を転送することで，サーバ s へメッセージを転送する．

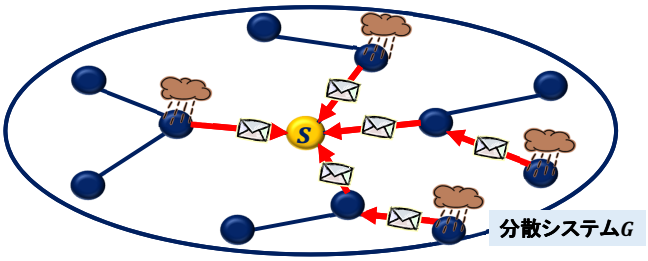


図 4: サーバ s 以外のプロセッサの動作

- サーバ s の動作
  - トリガが発生した場合
    $s \in S_j^\rho(1 \leq j \leq n)$ を満たす ρ-近傍 S_j^ρ のカウンタ $C(p_j)$（s が属する ρ-近傍のカウンタ）をインクリメントする．
  - メッセージ $TRIGGER(j)$ を受信した場合
    $p_j \in S_k^\rho$ を満たす ρ-近傍 S_k^ρ のカウンタ $C(p_k)$（p_j が属する ρ-近傍のカウンタ）をインクリメントする．

以上の処理を繰り返して，ある ρ-近傍 S_j^ρ で w 個のトリガを検出した（$C(p_j) = w$ になった）場合，サーバ s は幅優先全域木 T_G を用いて，メッセージ $\boldsymbol{ALARM(j)}$ を送り，そのことをプロセッサ p_j に知らせてアルゴリズムを終了する．

本アルゴリズムの疑似コードを示す．Algorithm2 がサーバ以外のプロセッサ p_i のアルゴリズム，Algorithm3 がサーバ s のアルゴリズムである．

したトリガ数，w は検出トリガ数とする．

4.2 アルゴリズムの評価

まず総メッセージ数についての評価を行う．このアルゴリズムでは，トリガが発生したプロセッサ p_i は，サーバ s へメッセージ $TRIGGER(i)$ を送信する．サーバは分散システム G の中心のプロセッサなので，これに要するメッセージ数は高々 $Rad(G)$ となる．よって，W を分散システム全体で発生したトリガ数とすれば，総メッセージ数は $O(W \cdot Rad(G))$ となる．この総メッセージ数は，半径 $Rad(G)$ が最大となるようなトポロジのとき（図 5）に最悪となり，総メッセージ数は $O(n \cdot W)$ となる．

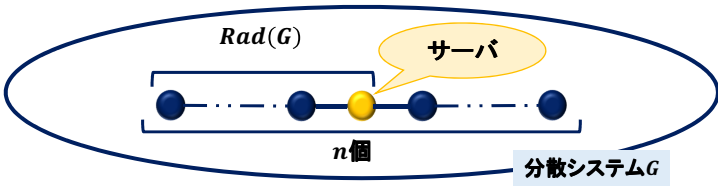


図 5: 総メッセージ数最悪時のトポロジ

次に，最大受信メッセージ数について評価を行う．プロセッサ p_i でトリガが発生したことを通知するメッセージは全てサーバ s に集められる．従って，最大受信メッセージ数はサーバが受信する W 個となる．これより，以下の定理が導かれる．

定理 2 半径が $Rad(G)$ の分散システム G において，分散アルゴリズム SERVER_COUNTING は総メッセージ数 $O(W \cdot Rad(G))$, 最大受信メッセージ数 W で ρ-近傍トリガ数え上げ問題を解く．ただし，W はシステム全体で発生

5 ρ-近傍トリガ数え上げに対するクラスタ型解法

本章ではまず，分散システム G の誘導部分システムであるクラスタとそのクラスタを粗大化した被覆を定義し，この被覆が与えられたときに，ρ-近傍トリガ数え上げ問題を解く分散アルゴリズム CLUSTER_COUNTING を示す．そして，最大次数の低い被覆，平均次数が低い被覆それぞれが与えられた場合の CLUSTER_COUNTING の通信計算量の評価を行う．

5.1 クラスタと被覆

まず**クラスタ**を定義する．分散システム $G = (V, E)$ のプロセッサの部分集合 $C(\subseteq V)$ に対し，両端のプロセッサが C に含まれる通信リンクの集合を $E' = \{(p_i, p_j) \in E | p_i, p_j \in C, i \neq j\}$ とする．このとき，C による誘導部分システム $G(C) = (C, E')$ をクラスタとよぶ．

特に混乱が生じない限り，クラスタ $G(C)$ を C と略記する．また，クラスタ $G(C)$ とプロセッサの集合を区別せずに扱う．次に，クラスタ C の**半径**と**中心**を定義する．

定義 2 クラスタ C におけるプロセッサ $v \in C$ の離心数 $Ecc(v, C)$ を以下のように定義する．

$$Ecc(v, C) = \max_{w \in C}\{dist_C(v, w)\}$$

この離心率 $Ecc(v, C)$ が最小となるプロセッサ u をクラスタ C の中心と呼び，$Ecc(u, C)$ をクラスタ C の半径 $Rad(C)$ とよぶ．

$$Rad(C) = \min_{v \in C}\{Ecc(v, C)\}$$

クラスタの集合 $\mathcal{C} = \{C_1, ..., C_m\}$ が，分散システム G のすべてのプロセッサを含む（$\cup\mathcal{C} = V$）とき，クラスタの集合 $\mathcal{C}$ を**被覆**とよぶ．また，各クラスタ $\{C_1, ..., C_m\}$ の半径のうち最大のものを被覆 $\mathcal{C}$ の **半径** $\boldsymbol{Rad(\mathcal{C})}$，各プロセッサ $p_i (1 \leq i \leq n)$ が属するクラスタの数を p_i の **クラスタ次数** $\boldsymbol{c\text{-}deg(p_i)} (= |\{C \in \mathcal{C} | p_i \in C\}|)$ とよぶ．また，以下のように，被覆 $\mathcal{C}$ の **最大クラスタ次数** $\boldsymbol{\Delta(\mathcal{C})}$ と **平均クラスタ次数** $\boldsymbol{\bar{\Delta}(\mathcal{C})}$ を定義する．

定義 3 最大クラスタ次数 $\Delta(\mathcal{C})$ を以下のように定義する．

$$\Delta(\mathcal{C}) = \max_{v \in V}\{c\text{-}deg(v)\}$$

定義 4 平均クラスタ次数 $\bar{\Delta}(\mathcal{C})$ を以下のように定義する．

$$\bar{\Delta}(\mathcal{C}) = \frac{1}{n} \cdot \sum_{v \in V} c\text{-}deg(v)$$

また本章では，プロセッサ p_i の ρ-近傍による誘導部分システムを p_i の ρ-近傍クラスタ S_i^ρ，すべてのプロセッサの ρ-近傍クラスタの集合 $\mathcal{S} = \{S_1, ..., S_n\}$ を ρ-近傍被覆 とよぶことにする．

5.2 アルゴリズム CLUSTER_COUNTING

本節では，被覆を利用して ρ-近傍トリガ数え上げ問題を解く分散アルゴリズム CLUSTER_COUNTING を示す．

本報告で利用する被覆 $\mathcal{T}$ は，いくつかの ρ-近傍クラスタを併合（**粗大化**，図6）して得られるクラスタで構成され，$\mathcal{T} = \{T_1, T_2, ..., T_m\} (m \leq n)$ と表す．つまり，$V = \bigcup_{j=1..m} T_j$ であり，

各 $S_i^\rho(1 \leq j \leq n)$ に対し $S_i \subseteq T_j$ なるクラスタ $T_i(1 \leq j \leq m)$ が存在する．

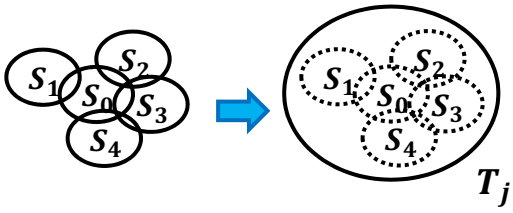


図 6: クラスタの粗大化

$\mathcal{T}$ の各クラスタ $T_j(1 \leq j \leq m)$ において，中心のプロセッサの 1 つがサーバとして指定されているものとし，このプロセッサを s_j と表す．サーバ s_j はクラスタ T_j のトポロジを知っており，T_j に属するプロセッサで発生したトリガの数を T_j に属するクラスタ S_i^ρ ごとに管理している．これらの情報から，サーバ s_j は $S_i^\rho \subseteq T_j$ なる ρ-近傍クラスタ S_i^ρ で発生したトリガの数が w に達したとき，プロセッサ p_i に通知する．つまり，各クラスタ T_j でアルゴリズム $SERVER\_COUNTING$ を実行することにより ρ-近傍トリガ数え上げ問題を解く，というのがアルゴリズム CLUSTER_COUNTING の概要である．以降，アルゴリズムについて詳細に述べていく．

まず各クラスタ $T_j(1 \leq j \leq m)$ に対し，サーバ s_j を根とする幅優先全域木 U_j があらかじめ構成されているものとする．各プロセッサ p_i は，高々最大クラスタ次数 $\Delta(\mathcal{T})$ 個の全域木に含まれることになるが，p_i はこれらの全域木の情報（親プロセッサ，子プロセッサの集合）を保持しているものとする．サーバ s_j は，各プロセッサが持つ情報に加え，クラスタ T_j のトポロジ情報（プロセッサの識別子を含む）を把握している．また，3 章では各プロセッサがそれぞれカウンタを保持していたが，本章ではサーバ s_j が T_j に含まれる各 ρ-近傍クラスタ S_i^ρ に対し，カウンタ $C(p_i)$ を保持している．

次に，各プロセッサ p_i の動作について説明する．本アルゴリズムでは，サーバとそれ以外のプロセッサでは動作が異なるため，それぞれ説明を行う．

- サーバ以外のプロセッサ p_i の動作
  - トリガが発生した場合
    トリガが発生したというメッセージを，$p_i \in T_j$ を満たす全てのクラスタ T_j のサーバ s_j に全域木 U_j を用いて送信する．このメッセージには，送り主と宛先が分かるように，送り主 p_i と宛先 s_j の識別子をメッセージに含める．以降，このメッセージを $\boldsymbol{TRIGGER_{p_i\text{-}s_j}}$ とよぶ．
  - メッセージを受信した場合
    p_i が受信するメッセージは，トリガが発生した他のプロセッサ p_l からサーバへ送信された $TRIGGER_{p_l\text{-}s_j}$ メッセージである．よってこのメッセージを受信した場合，全域木 U_j に従いサーバ s_j へ向けメッセージを転送する．
- p_i がサーバ s_k のときの動作
  - トリガが発生した場合
    $s_k \in S_l^\rho$ を満たす p_l のカウンタ $C(p_l)$（s_k が属する ρ-近傍クラスタのカウンタ）をインクリメント

する．ただしサーバ s_k が他のクラスタ $T_{s'}$ にも所属している場合は，$T_{s'}$ のサーバにトリガの発生を伝えるための，上記のサーバ以外のプロセッサの動作を実行する．

– メッセージを受信した場合
自分宛ではない $TRIGGER_{p_l\text{-}s_j}$ メッセージ（$s_j \neq s_k$）を受信した場合，上記のサーバ以外のプロセッサの動作に従いメッセージをサーバ S_j に向けて転送する．自分宛の $TRIGGER_{p_l\text{-}s_k}$ メッセージを受信した場合，$p_l \in S_t^\rho$ を満たす p_t のカウンタ $C(p_t)$（p_l が属する ρ-近傍クラスタのカウンタ）をインクリメントする．

以上の処理を繰り返して，ある ρ-近傍 S_l で w 個のトリガを検出した場合（あるカウンタ $C(p_l)$ の値が w になった場合），サーバ s_k は，T_k の幅優先全域木 U_k を用いて，このことをプロセッサ p_l に知らせる．このメッセージを $\boldsymbol{ALARM_{s_k}(L)}$ とよぶ．ただし，L は w 個のトリガを検出したプロセッサの識別子の集合である．

本アルゴリズムの疑似コードを示す．Algorithm4 がサーバ以外のプロセッサ p_i のアルゴリズム，Algorithm5 がプロセッサ p_i がサーバ s_k のときのアルゴリズムである．

5.3 アルゴリズムの評価

まず総メッセージ数についての評価を行う．このアルゴリズムでは，トリガが発生したプロセッサ p_i は，$p_i \in T_j$ を満たすサーバ s_j 全てに幅優先全域木 U_j を用いてメッセージ $TRIGGER_{p_i\text{-}s_j}$ を送信する．p_i から s_j まで $TRIGGER_{p_i\text{-}s_j}$ を届けるのに高々 $Rad(\mathcal{T})$ 個のメッセージが必要である．また p_i は，高々 $\Delta(\mathcal{T})$ のクラスタに属するので，トリガが1つ発生した際に必要なメッセージは高々 $\Delta(\mathcal{T}) \cdot Rad(\mathcal{T})$ となる．よって，システム全体で発生するトリガ数を W とすると，メッセージ数は $O(\Delta(\mathcal{T}) \cdot Rad(\mathcal{T}) \cdot W)$ となる．

定理 3 システム全体で発生するトリガ数を W とするとき，アルゴリズム $CLUSTER\_COUNTING$ の総メッセージ数は以下のようになる．

$$O(\Delta(\mathcal{T}) \cdot Rad(\mathcal{T}) \cdot W)$$

次に，最大受信メッセージ数について評価を行う．各プロセッサ p_i が受信するメッセージには2種類ある．1つは $ALARM_{s_k}(l)$ メッセージである．これは p_i が受信するのはアルゴリズムの終了時の一度のみである．もう1つは，トリガの発生を知らせる $TRIGGER_{p_k\text{-}s_j}$ である．$TRIGGER_{p_k\text{-}s_j}$ メッセージは，各プロセッサでトリガが1つ発生するたびに1つ生成される．システム全体で発生するトリガ数を W とすると，すべての $TRIGGER_{p_k\text{-}s_j}$ がプロセッサ p_i を経由する場合に最大受信メッセージ数が最大となる．よって，$ALARM_{s_k}(l)$ メッセージと合わせ，最大受信メッセージ数は高々 $W + \Delta(\mathcal{T})$ となる．

定理 4 システム全体で発生するトリガ数を W とするとき，アルゴリズム CLUSTER_COUNTING の最大受信メッセージ数は $O(W + \Delta(\mathcal{T}))$ となる．

5.3.1 最大クラスタ次数の低い被覆を与えた場合

本節では，5.2 節のアルゴリズム CLUSTER_COUNTING に最大クラスタ次数が低い被覆を与えた場合についての評価を行う．

最大クラスタ次数が低い被覆を構成するアルゴリズムには，MAX_COVER というアルゴリズムがある [6]．これは，ある被覆 $\mathcal{C}$ を入力として，C に属するクラスタを粗大化することで最大クラスタ次数の低い被覆 $\mathcal{T}$ を出力するアルゴリズムである．MAX_COVER が出力する被覆 $\mathcal{T}$ について，以下の補題 1 が成り立つことが知られている．

補題 1 [6] パラメータを κ，入力を被覆 $\mathcal{C}$ とした時，アルゴリズム MAX_COVER は以下の性質を満たす被覆 $\mathcal{T}$ を出力する．ただし，$|\mathcal{C}|$ は被覆 $\mathcal{C}$ のクラスタの数とする．

- $Rad(\mathcal{T}) \leq (2\kappa - 1)Rad(\mathcal{C})$
- $\Delta(\mathcal{T}) \leq 2\kappa \cdot |\mathcal{C}|^{\frac{1}{\kappa}}$

MAX_COVER に ρ-近傍被覆 $\mathcal{S}$ を入力すれば，$\mathcal{S}$ のクラスタの数は n 個であることより，次のような最大クラスタ次数の低い被覆 $\mathcal{T}$ が得られる．

- $Rad(\mathcal{T}) \leq (2\kappa - 1)\rho$
- $\Delta(\mathcal{T}) \leq 2\kappa \cdot n^{\frac{1}{\kappa}}$

この被覆 $\mathcal{T}$ を用いると，アルゴリズム CLUSTER_COUNTING の通信計算量は定理 3，4，補題 1 より次のようになる．

定理 5 最大クラスタ次数の低い被覆 $\mathcal{T}$ を用いると，アルゴリズム CLUSTER_COUNTING の総メッセージ数は，$O(\Delta(\mathcal{T}) \cdot Rad(\mathcal{T}) \cdot W) = O(\kappa^2 \cdot n^{\frac{1}{\kappa}} \cdot \rho \cdot W)$ となる．また，最大受信メッセージ数は $O(W + \kappa \cdot n^{1/\kappa})$ となる．

5.3.2 平均クラスタ次数の低い被覆を与えた場合

本節では，5.2 節のアルゴリズム CLUSTER_COUNTING に平均クラスタ次数が低い被覆を与えた場合についての評価を行う．

平均クラスタ次数が低い被覆を構成するアルゴリズムには，AV_COVER というアルゴリズムがある [6]．これは，ある被覆 $\mathcal{C}$ を入力として，C に属するクラスタを粗大化することで平均クラスタ次数の低い被覆 $\mathcal{T}$ を出力するアルゴリズムである．AV_COVER が出力する被覆 $\mathcal{T}$ について，以下の補題 2 が成り立つことが知られている．

補題 2 [6] パラメータを κ，入力を被覆 $\mathcal{C}$ とした時，アルゴリズム AV_COVER は以下の性質を満たす被覆 $\mathcal{T}$ を出力する．

- $Rad(\mathcal{T}) \leq (2\kappa + 1)Rad(\mathcal{C})$
- $\overline{\Delta}(\mathcal{T}) \leq n^{1/\kappa}$

AV_COVER に ρ-近傍被覆 $\mathcal{S}$ を入力すれば，次のような平均クラスタ次数の低い被覆 $\mathcal{T}$ が得られる．

- $Rad(\mathcal{T}) \leq (2\kappa+1)\rho$
- $\overline{\Delta}(\mathcal{T}) \leq n^{1/\kappa}$

この被覆 $\mathcal{T}$ を用いると，アルゴリズム CLUSTER_COUNTING の通信計算量は定理 3，4，補題 2 より次のようになる．

定理 6 平均クラスタ次数の低い被覆 $\mathcal{T}$ を用いると，アルゴリズム CLUSTER_COUNTING の総メッセージ数は，定理 3 より $O(\Delta(\mathcal{T})\cdot Rad(\mathcal{T})\cdot W) = O(\kappa\cdot n^{1/\kappa}\cdot\rho\cdot W)$ となる．また，最大受信メッセージ数は，定理 4 より $O(W+\kappa\cdot n^{1/\kappa})$ となる．

6 まとめ

本報告では，ρ-近傍トリガ数え上げ問題を提起し，この問題を解く分散アルゴリズムを 3 つ示した．最初に，各プロセッサがそれぞれ ρ-近傍での発生トリガ数を数えることで検出するアルゴリズム SIMPLE_COUNTING を示した．このアルゴリズムは，疎なトポロジにおいてメッセージ数の効率が良い．次に，ある 1 つのプロセッサが，すべての ρ-近傍での発生トリガ数を数えることで検出するアルゴリズム SERVER_COUNTING を示した．このアルゴリズムは，SIMPLE_COUNTING と対照的に，密なトポロジにおいてメッセージ数の効率が良い．最後に，これらのアルゴリズムの折衷手法として，クラスタごとに SERVER_COUNTING を適応する CLUSTER_COUNTING を示した．このアルゴリズムでは，最悪時の総メッセージ数に関して，先の 2 つより改善されることを示した．

本報告で提案した分散アルゴリズムはいずれも，トリガが発生するたびにトリガの発生を通知している．従って，トリガが非負の実数の重みを持ち，その和が閾値に達したことを検出する**局所的重み付きトリガ数え上げ問題**に対しても，重みを通知することにより，これらの分散アルゴリズムを適用できる．さらに，各プロセッサで数え上げるトリガの重みが，トリガの重みとトリガが発生したプロセッサからの距離に依存する場合にも，重みと距離を考慮することにより，これらの分散アルゴリズムを適用可能である．ただし，アルゴリズム SIMPLE_COUNTING では，トリガの重みに加えて距離の通知が必要となる．

局所的重み付きトリガ数え上げ問題で，トリガの負の重みを許すことも可能である．この場合にもこれらのアルゴリズムを適用できるが，非同期式システムでは，重み付きトリガ数え上げ問題を正確に解くことは不可能である．これは，重みが非負の場合は，発生したトリガの重みの和が閾値以上であるという性質は安定（一度でも満たされると，それ以降も満たされる）であるが，負の重みを許すと安定ではなくなるためである．このため，非同期式システムでは，誤検出（フォールスポジティブ）か検出漏れ（フォールスネガティブ）の発生を避けることはできない．

文献 [2] の木構造に基づくアルゴリズムでは，トリガが発生するたびにそのことの通知を避けることにより,（大域的）トリガ数え上げアルゴリズムのメッセージ数を削減している．この手法は，ネットワーク全体でのトリガ数の数え上げには有効であるが，局所的トリガ数え上げへの適用は容易でない．局所的トリガ数え上げ問題アルゴリズムのメッセージ数の改善は今後の課題である．

参考文献

[1] Ram Keralapura, Graham Cormode, and Jeyashankher Ramamirtham. Communication-efficient distributed monitoring of thresholded counts. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pp. 289–300, 2006.

[2] Rahul Garg, Vijay K.Garg, and Yogish Sabharwal. Scalable algorithms for global snapshots in distributed systems. In *Proceedings of the 20th annual international conference on Supercomputing(ICS)*, pp. 269–277, 2006.

[3] Venkatesan T.Chakaravarthy, Anamitra R.Choudhury, Vijay K.Garg, and Yogish Sabharwal. An efficient decentaralized algorithm for the distributed trigger counting problem. In *12th International Conference on Distributed Computing and Networking(ICDCN)*, pp. 53–64, 2011.

[4] Sushanta Karmakar, A. Chandrakanth Reddy, and Yogish Sabharwal. Improved algorithm for the distributed trigger counting problem. *IEEE International Parallel and Distributed Processing Symposium(IPDPS)*, pp. 515–523, 2011.

[5] Venkatesan T.Chakaravarthy, Anamitra R.Choudhury, and Yogish Sabharwal.

An improved algorithm for distributed trigger counting in ring. *The Computer Journal*, 2013.

[6] David Peleg. *Distributed Computing: a locality-sensitive approach.* Society for Industrial and Applied Mathematics, 1987.

Algorithm 1 SIMPLE_COUNTING（p_i の動作）

const

w：検出トリガ数

p_j：任意のプロセッサ $(1 \leq j \leq n)$

$Child_j(p_i)$：全域木 T_j^ρ における p_i の子プロセッサの集合

var

$C(p_i) \leftarrow 0$：各カウンタを初期化

function

1: **while** 各カウンタ $C(p_i) \neq w$
2: **begin**
3: **if**（トリガが発生） **then**
4: $C(p_i) \leftarrow C(p_i) + 1$：カウンタをインクリメント
5: T_i^ρにおける p_iの子 $Child_i(p_i)$ へ $TRIGGER_{p_i}$ メッセージを送信
6: **else if**（$TRIGGER_{p_j}$ メッセージを受信）**then**
7: $C(p_i) \leftarrow C(p_i) + 1$：カウンタをインクリメント
8: T_j^ρにおける p_iの子 $Child_j(p_i)$ へ $TRIGGER_{p_j}$ メッセージを転送
9: **end**
10: w 個のトリガを検出しアルゴリズムを終了する

Algorithm 2 SERVER_COUNTING（サーバ以外のプロセッサ $p_i(1 \leq i \leq n)$ の動作）

const

w：検出トリガ数

$parent(p_i)$：全域木 T_G における p_i の親プロセッサ

$Child(p_i)$：全域木 T_G における p_i の子プロセッサの集合

L：w 個のトリガを検出したプロセッサの識別子の集合

function

1: **while** $ALARM(L)$ が未受信
2: **begin**
3: **if** （トリガが発生） **then**
4: p_i の親 $parent(i)$ へ $TRIGGER(p_i)$ メッセージを送信
5: **else if** （$TRIGGER(j)$ メッセージを受信）**then**
6: p_i の親 $parent(i)$ へ $TRIGGER(j)$ メッセージを転送
7: **end**
8: **if** （$i \in L$）**then**
9: S_i^ρ が検出トリガ数に達したことを検出
10: p_i の子 $Child_j(p_i)$ へ $ALARM(L)$ メッセージを転送
11: アルゴリズム終了

Algorithm 3 SERVER_COUNTING（サーバ s の動作）

const

w：検出トリガ数

$parent(p_i)$：全域木 T_G における p_i の親プロセッサ

$Child(p_i)$：全域木 T_G における p_i の子プロセッサの集合

$L = \{p_l | C(p_l) = w\}$：w 個のトリガを検出したプロセッサの識別子の集合

var

$C(p_i) \leftarrow 0$：各 $p_i (1 \leq i \leq n)$ に対しカウンタを初期化

function

1: **while** （任意の $p_i (1 \leq i \leq n)$ に対し $C(p_i) \neq w$）
2: **begin**
3: **if** （トリガが発生） **then**
4: **for** $\{p_i \mid s \in S_i^{\rho}\}$ /* s を含む ρ-近傍のカウンタを増やす */
5: $C(p_i) \leftarrow C(p_i) + 1$
6: **else if** （$TRIGGER(i)$ メッセージを受信）**then**
7: **for** $\{p_j \mid p_i \in S_j^{\rho}\}$ /* p_i を含む ρ-近傍のカウンタを増やす */
8: $C(p_j) \leftarrow C(p_j) + 1$
9: **end**
10: **for** 各 $p_i (1 \leq i \leq n)$ に対し
11: **if** （$C(p_i) = w$） **then**
12: プロセッサの識別子の集合 L に p_i を追加
13: サーバ s の子 $Child(s)$ へ $ALARM(L)$ メッセージを送信
14: アルゴリズム終了

Algorithm 4 $CLUSTER\_COUNTING$（サーバ以外のプロセッサ $p_i(1 \leq i \leq n)$ の動作）

const

w：検出トリガ数

$Server(p_i) = \{s_j \mid p_i \in T_j\}$：$p_i$ を含むクラスタのサーバの集合

$parent_j(p_i)$：全域木 U_j における p_i の親プロセッサ $(s_j \in Server(p_i))$

$Child_j(p_i)$：全域木 U_j における p_i の子プロセッサの集合 $(s_j \in Server(p_i))$

L：w 個のトリガを検出したプロセッサの識別子の集合

function

1: **while** どのサーバ $s_j(1 \leq j \leq m)$ からも $ALARM_{s_j}(L)$ が未受信
2: **begin**
3: **if**（トリガが発生）**then**
4: **for** 各 $s_j \in Server(p_i)$ /* s_j ：p_i を含むクラスタのサーバ */
5: U_j における p_i の親 $parent_j(p_i)$ へ $TRIGGER_{p_i-s_j}$ メッセージを送信
6: **else if**（$TRIGGER_{p_k-s_j}$ メッセージを受信）**then**
7: U_j における p_i の親 $parent_j(p_i)$ へ $TRIGGER_{p_k-s_j}$ メッセージを転送
8: **end**
9: **if**（$i \in L$）**then**
10: S_i^{ρ} が検出トリガ数に達したことを検出
11: U_j における p_i の子 $Child_j(p_i)$ へ $ALARM_{s_j}(L)$ メッセージを転送
12: アルゴリズム終了

Algorithm 5 $CLUSTER\_COUNTING$（サーバ s_k の動作）

const

w：検出トリガ数

$Server(s_k) = \{s_j \mid s_k \in T_j\}$：$s_k$ を含むクラスタのサーバの集合

$parent_j(s_k)$：全域木 U_j における s_k の親プロセッサ ($s_j \in Server(s_k)$)

$Child_k$：全域木 U_k における s_k の子プロセッサの集合

$L = \{p_l | C(p_l) = w\}$：w 個のトリガを検出したプロセッサの識別子の集合

var

$S_i^\rho \subseteq T_k$なる各 p_iに対し $C(p_i) \leftarrow 0$：各カウンタを初期化

function

while （$S_i^\rho \subseteq T_k$なる各 p_iに対し $C(p_i) \neq w$）**and**
（どのサーバ s_jからも $ALARM_{s_j}(L)$ が未受信）
begin
  if （トリガが発生） **then**
    for $s_k \in S_i^\rho$ なる 各 p_i に対し　/* s_k を含む ρ-近傍のカウンタを増やす */
      $C(p_i) \leftarrow C(p_i) + 1$
    for 各 $s_j \in Server(s_k)$　/* s_j ：s_k を含むクラスタのサーバ */
      U_jにおける s_k の親 $parent_j(s_k)$ へ $TRIGGER_{s_k - s_j}$ メッセージを送信
  else if （$TRIGGER_{p_i - s_j}$ メッセージを受信）**then**
    if （$s_j = s_k$） **then**　/* 自分宛のメッセージの場合 */
      for $s_j \in S_i^\rho$ なる 各 p_i に対し　/* p_i を含む ρ-近傍のカウンタを増やす */
        $C(p_i) \leftarrow C(p_i) + 1$
    else
      U_jにおける s_k の親 $parent_j(s_k)$ へ $TRIGGER_{p_i - s_j}$ メッセージを転送
end
if （$S_i^\rho \subseteq T_k$ なる 各 p_i に対し $C(p_i) = w$） **then**
  プロセッサの識別子の集合 L に p_i を追加
  $Child_k$ へ $ALARM_{s_k}(L)$ メッセージを送信し p_i に通知
else if （$ALARM_{s_j}(L)$ 受信）**then**
  $S_{s_k}^\rho$ が検出トリガ数に達したことを検出
  U_j における p_i の子 $Child_j(s_k)$ へ $ALARM_{s_j}(L)$ メッセージを転送
アルゴリズム終了

トピック別資源発見問題について

清水 与也　　泉 泰介

近年, P2P ネットワーク, ソーシャルネットワークなどの大規模分散ネットワークなどがインターネットにおけるコミュニケーションの新しい形として注目を集めている. 大規模分散ネットワークにおいて重要な基本問題の一つとして資源発見問題がある. 資源発見問題はシステムに参加しているノードの ID 値を全ノードが収集する問題として定義される. あるノードが別のノードの ID を保有することをオーバーレイネットワークにおける論理リンクを生成すると解釈した場合, この問題は全ノード上で完全グラフオーバーレイネットワークを構成する問題とみなすことができる.

本研究では, 従来の資源発見問題を拡張したトピック別資源発見問題を提案しその効率的な解放の検討を行う. トピック別資源探索問題では, 各ノードがトピックと呼ばれる自身の嗜好を示す情報を保有している. より形式的には, ノード集合を V, 全てのトピック集合を T としたとき, ノードの嗜好を表す関数 Int : $V \to 2^T$ が定義されているものとする. トピック別資源探索問題は, 任意の $t \in T$ について, $t \in Int(v)$ であるような頂点 v すべてからなる集合により誘導されるネットワークのトポロジがクリークとなるように オーバレイを構成する問題と定義される.

トピック別リソース発見問題を解くナイーブな手法として考えられるものは, 各トピックについて従来のリソース発見の手法を並列に動作させることである. しかしながら, このアプローチは以下にあげる問題が生じる.

- １つ目は, 帯域幅の問題である. これは, トピックごとに各ノードがリソース発見のプロセスを実行するため, トピックの種類の数だけ各ノードはネットワークで通信を行う情報が増えてしまう. このことにより, システムに参加するノードが増えるにつれてトピック数だけ通信を行う回数が増えることになり, スケーラビリティの点で問題が生じる.
- ２つ目に, 同じトピックのグループが複数の部分的なグループに分断される可能性があることである. これはトピックごとに従来の手法を適用するとき, , 同じトピックを持つノード対の距離が 3-hop 以上である場合はそのノード対にエッジが追加されないため, 同じトピックでありながら別のグループに所属してしまうことになる.

本研究では, 単位時間の通信帯が O(log n) ビット (n はノードの総数) に制限されたモデル上で, 上記の問題を解決したトピック別資源探索問題のための新しいアルゴリズムの実現可能性について検討する. 検討に際して, 資源発見問題における既存のゴシップに基づくアルゴリズム [1] に注目する. このアルゴリズムは各時刻においてランダムに自身が保有する ID を隣接ノードに伝えることで資源発見問題を解く単純なアルゴリズムである. 本研究では, このアルゴリズムに適用するために, どのような改変が必要かを検討し, アルゴリズム設計の道筋を明らかにすることを目指す.

前述のアルゴリズムに基づく, 本研究のトピック別資源発見問題を解くアルゴリズムを以下に挙げる.

まず, 各ノードは, ID・トピック・生存時間 (カウンタ) からなる情報をリストに持つ. 初期状態として, 自身の ID・トピックの情報をリストに持つ.

- 情報を送るノードは, 各ラウンドでリストからランダムにID・トピック・生存時間の情報を一つ選択し, ランダムに選択した隣接ノードに配布する. その後, 送った情報はリストから削除する (自身のトピックに関する情報は残す).
- 情報を受け取ったノードは, 自身の持っているトピックと一致する場合, エッジを追加する. その後, カウンタを一つ下げてリストに追加する. その時に, 残り生存時間が”0”になった場合は, 削除する. 上記以外の場合は, リストにその情報を追加する (カウンタを一つ下げる). これも同様にカウンタが”0”になる場合は削除.

このアルゴリズムの主な狙いとしては, 生存時間だけ離れたノードに自信のトピック情報を送ることができることである. つまり, 各ノードは, 自身のトピック情報に加えて, 生存時間分離れたノードのトピック情報を持つことになる. そのため, 所持している情報が増えるにつれて擬似的に所持トピック数が増えていき, 一度に行われる通信で新たにエッジが追加される確率を増やすことができる. このことにより, 高い確率でトピック別にクリークを作成することができることが予測される.

参考文献

[1] Bernhard Haeupler, Gopal Pandurangan,David Peleg, Rajmohan Rajaraman and Zhifeng Sun. Discovery through Gossip. arXiv: 1202.2092, 2012.

[2] B. Doerr, T. Friedrich, and T. Sauerwald. Quasi-random rumor spreading. In SODA, pages 773-781, 2008.

[3] Giakkoupis. Tight bounds for rumor spreading in graphs of a given conductance. In STACS, pages 57-68, 2011.

[4] M. Harchol-Balter, T. Leighton, and D. Lewin. Resource discovery in distributed networks. In PODC, pages 229-237, 1999.

[5] Damon Mosk-Aoyama and Devavrat Shah. Computing separable functions via gossip. In PODC, pages 113-122, 2006.

[6] M. Jelasity, A. Montresor, and O. Babaoglu. T-man: Gossip-based fast overlay topology construction.Computer networks, 53:2321-2339, 2009.

[7] S. Kutten, D. Peleg, and U. Vishkin. Deterministic resource discovery in distributed networks. In SPAA, 2001.

自律分散ロボットによる線分上の被覆問題について

後藤　千香行

近年，自律分散ロボット群の協調動作のためのアルゴリズム設計が注目を集めている．本研究では，視野の制限された自律分散ロボット群による線分上の被覆問題について検討する．この問題では，各ロボットは，他のロボットの位置を感知できるセンサー，および，侵入者を感知できるセンサーの 2 種類を備えており，それぞれが異なる検出可能範囲を持つ．問題の最終的な目標は，与えられた線分上の全ての点を各ロボットの侵入者感知センシング領域で被覆するように，侵入者を検出できる障壁範囲を形成することである．

各ロボットは，範囲 r の侵入者を検出できるセンサーと，他のロボットを感知することができる視界 R を持つ．また，ロボットは固有の ID，および過去の行動履歴を記録するためのメモリを持たないものとする．各ロボットは初期状態において，被覆対象となる線分上の任意の位置に配置されており，移動は線分に沿った移動のみが可能である．このとき，ロボットは，視野内の他のロボットの位置を確認し，アルゴリズムを用いて自分の移動先を計算し，計算した場所に移動する．

被覆線分の長さを L とすると，各ロボットが，被覆できる領域は高々$2r$ であるので，線分被覆問題を解くためには少なくとも $\lceil L/2r \rceil$ 台のロボットが必要である．また，被覆問題の可解性はその他にも (1) 視野範囲 R の大きさ，(2) 余剰ロボットの台数，(3) 各ロボットが左右の方向付けに関して合意が取れているかどうか，(4) ロボットの同期モデル，等の因子により影響される．これらの因子と被覆問題の可解性についてこれまでに知られている結果を表 1 に示す [1]．なお，ここで線分被覆問題が ϵ-近似で可解とは，被覆されていない連続領域の最大長さが高々ϵ となる (すなわち，高々ϵ の長さ隙間が生じる) ような被覆を達成する問題である．また，SSYNC モデルは，各単位時刻において一部のロボットが同期的に動作するモデルであり，移動に掛かる時間を考えないものとし，ASYNC モデルは，ロボットの動作タイミングに関して一切の仮定を置かないモデルである．

表 1: 被覆問題の可解性

視野	余剰台数	同期モデル	方向付け	可解性
$R \geq 2r$	0	ASYNC	あり	可解
$R \geq 2r$	0	SSYNC	なし	不可能
$R > 2r$	0	SSYNC	なし	ϵ-近似

上記の研究結果においては，いずれもロボットの台数に関して余剰のないケースを考えていた．本研究では，タイトな視野，および方向付けのないロボット群の可解性について，以下の結果を得た．

定理 1 各ロボットの視野が $R > 2r$ のとき，線分被覆問題が可解であるならば，視野 $R = 2r$ のロボットの余剰台数 *1* で線分被覆問題を解くことができる．

参考文献

[1] M. Eftekhari, P. Flocchini, L. Narayanan, J. Opatrny, and N. Santoro. Distributed Barrier Coverage with Relocatable Sensors. Structual Information and Communication Complexity, pages 235-249, 2014.

A Local Information Based Distributed Algorithm Constructing 3-Nodes Rectilinear Steiner Tree in Virtual Grid Networks

Yonghwan Kim† Fukuhito Ooshita† Hirotsugu Kakugawa† Toshimitsu Masuzawa†

†Graduate School of Information and Technology, Osaka University, Osaka, Japan

Abstract

In this paper, we present a local information based distributed algorithm which constructs a rectilinear steiner tree among 3 nodes in virtual grid networks. A virtual grid network is obtained by virtually dividing a wireless network into a grid of geographical square regions called *cells*. A single node is selected as a *router* at each cell and inter-cell communication is realized by using the routers. Other nodes in the cell have no responsibility for inter-cell communication and can become inactive to save energy consumption.

We suppose one special node (named a *home* node) and several moving nodes (named *target* nodes) in virtual grid networks. And we consider maintenance of inter-cell communication paths to each target node from the a *home* node. In this paper, we propose an optimizing protocol in virtual grid networks, which can transform arbitrary given set of paths (from a home node to each target node) to rectilinear steiner tree in order to save energy consumption.

1 Introduction

Recently, wireless networks, such as MANETs (Mobile Ad-hoc NETworks) or WSNs (Wireless Sensor Networks), become popular and important in the distributed systems. In the wireless networks, nodes are deployed on a two-dimensional plane, and each node can directly communicate only with nodes within its communication range. If the destination node (the node receives the message) is outside of the communication range of the source node (the node sends the message), the message should be relayed to the destination node.

The topology of wireless networks can be changed frequently because of the mobility property of a node or node failures. Moreover, each node in the wireless networks has resource scarcity, for example, processing power, energy, and storage. Therefore, the key issues of the wireless routing protocols contain adaptability to the network dynamics and reduction of resource consumption[1, 2].

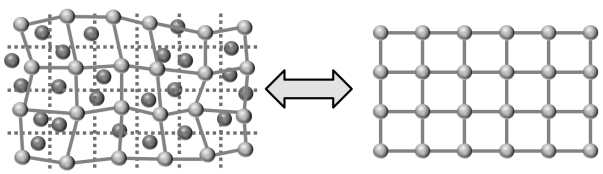


Figure 1: A Virtual Grid Network

Fig. 1 represent a virtual grid network[3], which is obtained by virtually dividing a wireless network into a grid of geographical square regions called *cells* of the same size. The size of the cells is determined so that any nodes in the same cell or in the neighboring cells can directly communicate with each other.

A single node is selected as a *router* at each cell and inter-cell communication is realized by using the routers. Other nodes in the cell have no responsibility for inter-cell communication and can become inactive (e.g., sleep) to save energy consumption. Since energy efficiency is one of the most critical issues in wireless networks, a router of each cell should be periodically reselected.

We suppose one special node (named a *home* node) and several moving nodes (named *target* nodes) in virtual grid networks. And we assume that the set of paths, which consists of each path to each target node from a home node, are given. This implies that if there are T target nodes in the virtual grid network, T paths (from a home node to each target node) are given.

In this paper, we consider maintenance of inter-cell communication paths to each target node from the node. As we mentioned above, the power consumption is one of the important issue in wireless sensor networks, therefore, we consider that our goal is the construction of a rectilinear steiner tree (means the steiner tree on the grid topology) which connects a home node to all target nodes. Steiner tree ensures that the total number of edges between routers from its definition. However, construction of rectilinear steiner tree in grid networks is known as *NP-hard* problem[4]. This implies that the devising of an distributed algorithm, which can find an optimal solution with local information only, is nearly impossible. Thus, in this paper, we consider only three nodes, two target nodes and one home node, in a virtual grid network. We propose an optimizing protocol in virtual grid networks, which can transform arbitrary given two paths (from a home node to two target nodes) to rectilinear steiner tree for saving energy consumption.

2 3-nodes Rectilinear Steiner Tree Problem

In this Section, we define 3-nodes rectilinear steiner tree (RST) problem in detail, and introduce some characteristics of this problem.

We suppose one special node (named a *home* node) and two moving nodes (named a *target* nodes) in virtual grid

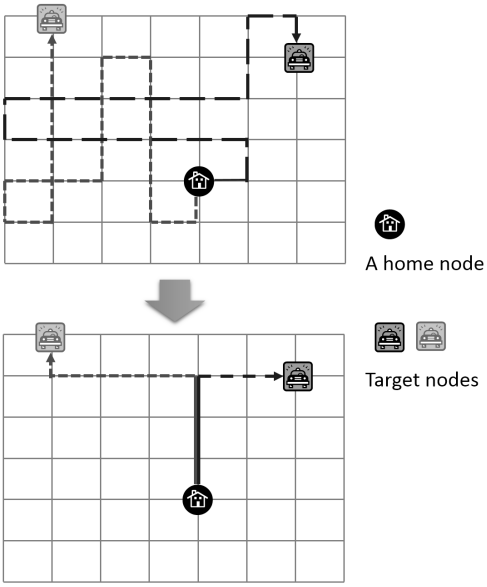


Figure 2: An example of 3-nodes RST problem

networks. And we assume that two arbitrary paths, which are the paths to two target nodes from a home node, are given. The goal of our protocol is the construction of the minimum rectilinear steiner tree among these three nodes. Fig. 2 illustrates an example of 3-nodes RST problem. Initially, two paths between each target node and a home node are given. Certainly, these path are not the shortest path between them and can be updated by the moving of a target node. Our proposed protocol construct the minimum rectilinear steiner tree like the lower part of Fig. 2 using local information and local updates only.

3-nodes RST problem can be changed to shortest-path tree problem. This implies that if the construction of a shortest-path tree with minimum total edge weight (in the case of a virtual grid network, the minimum number of edges) is available, that tree also becomes a rectilinear steiner tree. This can be allowed when there are only 3 nodes in a virtual grid network.

To help to understand, we use the cartesian coordinate plane (two-dimension) and we suppose a home node is exist on origin (0,0) of this plane. And we can represent the positions of two target nodes as a pair of x and y value. Fig.

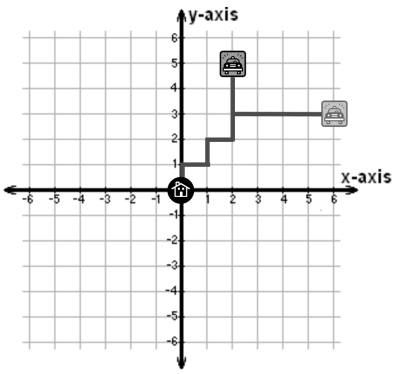


Figure 3: A steiner tree when two target nodes are in the same quadrant

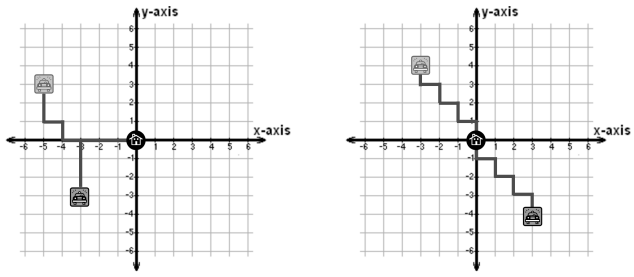


Figure 4: The other exmaples of steiner trees

3 represent an example of a steiner tree when two target nodes are in the same quadrant of a cartesian coordinates. In this case, we can simply construct a steiner tree as follow steps (if we can know global information).

1. Make shortest path to $(min(|x_1|, |x_2|), min(|y_1|, |y_2|))$ from a home node $(0, 0)$. (where two target nodes are located on (x_1, y_1) and (x_2, y_2)).
2. Make shortest (becomes straight line) path to each target node from $(min(|x_1|, |x_2|), min(|y_1|, |y_2|))$.

Likewise, we can easily construct a steiner tree in other two cases: two target nodes are located in the neighboring quadrants, and in the opposite quadrants. Fig. 4 represent an example of a steiner tree in the above two cases. Note that each (spanning) trees in Fig. 3 and 4 becomes not only a shortest-path tree but a steiner tree.

Our protocol adopts the routing protocol *Zigzag*[5], which can transform any given inter-cell path to a shortest (or minimum-hop) one by repeatedly applying local updates on the path (will be introduced in Section 3). In our protocol, we construct a shortest path to each target node from a home node using the protocol *Zigzag*, and, if possible, we combine some parts of two paths to reduce the number of edges. Our protocol can be operated on each router locally and adaptively (unaffected by target nodes' moves) in a virtual grid network using local information only.

3 Our proposed protocol

As we mentioned, we adopt a routing protocol named *Zigzag* to make a shortest path to each target node from a home node, and we propose some new protocols to combine some parts of two paths for reduction the number of edges contained in tree.

3.1 A routing protocol *Zigzag* and its modification

Zigzag is a local-information-based self-optimizing routing protocol in virtual grid networks. Protocol *Zigzag* find a shortest path between two nodes by repeatedly applying local updates to the path until it converges to a shortest path. *Zigzag* detects a redundancy of the recent path locally with making *zigzag-based* path.

Zigzag defines only three local updates on the node p_i as Fig. 5.

We consider the combining of two paths which are transformed (or been transforming) by *Zigzag*. However, the

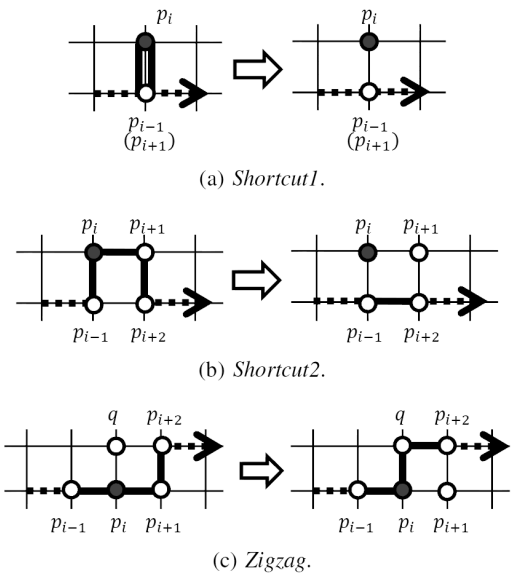


Figure 5: Three local updates to a path in *Zigzag*

converged shortest path can be different depending on the initial path, even if the positional relation between two nodes is exactly same. Fig. 6 represent two possibilities of the converged shortest path by *Zigzag*. When a target node is located in the first quadrant, a shortest path can be the one of two possible paths after convergence. In the case of Fig. 6, a shortest path can be started with up direction or right direction. This causes some difficulties when we find the relation between two paths. For example, if two target nodes are in the same quadrant, the overlapping of some parts of two paths can be expected. However, two converged paths can be completely different depending on starting directions.

Therefore, we modify the protocol *Zigzag* slightly. If a home node finds some specific starting directions, a home node updates those directions. Table 1 shows the detail rules of starting directions.

This modification makes it an open possibility to combine two paths. We introduce our protocol in the Section 3.2.

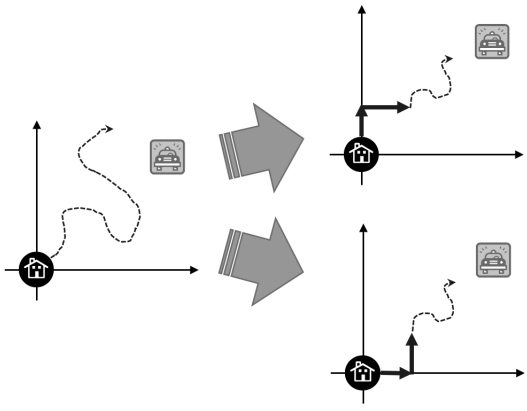


Figure 6: Two possible converged paths by *Zigzag*

Table 1: Rule for fixing the starting directions

2-hop directions from home	modified directions
{UP, RIGHT}(1st quadrant)	{RIGHT, UP}
{LEFT, UP}(2nd quadrant)	{UP, LEFT}
{DOWN, LEFT}(3rd quadrant)	{LEFT DOWN}
{RIGHT, DOWN}(4th quadrant)	{DOWN, RIGHT}

3.2 Path Combining Protocol

In this Section, we introduce our protocol to reduce the number of edges combining two paths. In the previous Section, we modified the protocol *Zigzag* thus we can easily find the positional relationship between two paths. We explain how to combine two paths in detail.

3.2.1 Two target nodes are in the neighboring quadrants

At first, we consider the case when two target nodes are in the neighboring quadrants (e.g., one target node is in the 1st quadrant, and another one is in the 2nd quadrant).

Fig. 7(a) shows an example case when two target nodes are in the neighboring quadrants, 1st and 2nd quadrants. In this case, we can find *U-shaped* path, from $(0, 1)$ to $(1, 1)$, on a home node (the origin). From this U-shaped path, a home node recognizes that two target nodes are in the neighboring quadrants and two paths can be combined. Note that, this recognition might be incorrect because *Zigzag* is not converged yet, but a home node can decide it at the current moment and our protocol can resolve this local miss.

Fig. 7(b) presents the situation after combining detected U-shaped path. Total number of edges is less than Fig. 7(a), however we cannnot find more combining points although this is not optimal solution. Therefore, we introduce a new virtual node named *virtual home* node (*vHome*). *vHome* is located on the home node initially, and after combining two path on U-shaped path as Fig. 7(b), *vHome* moves one hop along the combined path. In the case of Fig. 7(b), *vHome* is located on $(0, 1)$. *vHome* operates exactly same as a home node, this changes the *Zigzag* paths on Fig. 7(b). The path to left target changes its starting directions {UP, LEFT} due to the modification of *Zigzag* (Section 3.1). The zigzag part of the path to right target moves left by *Zigzag* protocol.

Fig. 8(a) illustrates the path after *vHome* is located on $(0, 1)$. Because of local updates of *Zigzag* protocol and its modification, our protocol can find the combining point again. A virtual home moves repeatedly and this protocol can be eventually converged as Fig. 8(b).

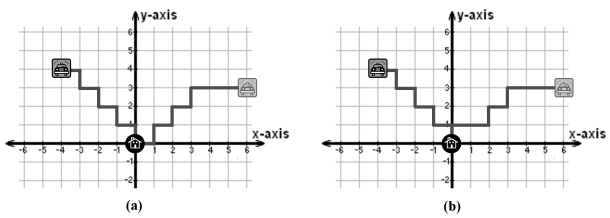


Figure 7: An example case of neighboring quadrants

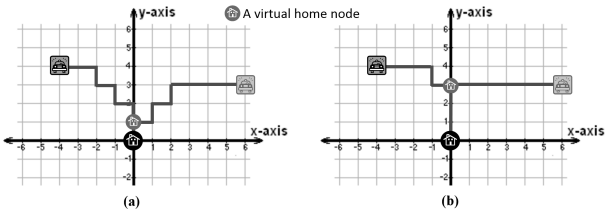


Figure 8: Coordinating using a virtual home

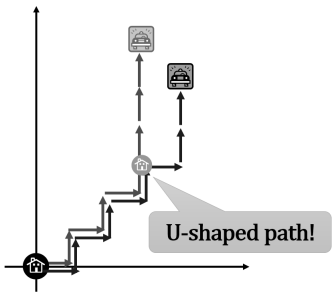


Figure 9: When two target nodes are in the 1st quadrant

3.2.2 Two target nodes are in the same quadrants

In the case when two target nodes are in the same quadrants (e.g., both of them are located in the third quadrant), some of paths are overlapped because of *Zigzag* protocol and its modification.

Fig. 9 shows an example when both of two target nodes are located in the first quadrant. We can easily find the overlapped (common) path from a home node (to ($min(|x_1|, |x_2|)$, $min(|y_1|, |y_2|)$). In this case, *vHome* can move along this common path and we can find a new U-shaped path (if exists) for combining two paths as we explained in Section 3.2.1.

3.2.3 Two target nodes are in the opposite quadrants

If two target nodes are in the opposite quadrants (e.g., one is in the 1st quadrant and another is in the 3rd quadrant), there is no U-shaped path or common path. Therefore, any combining of paths are never occurred.

3.2.4 Supplement of our protocol

Our protocol using *vHome* basically updates the path, if it finds U-shaped path. However, we cannnot know whether a protocol *Zigzag* is converged or not using local information only. As we mentioned in Section 3.2.1, our protocol's local update (combining) is sometimes incorrect. However, our protocol allows not only moving forward but also moving backward of *vHome* when it finds U-shaped path.

Fig. 10 shows the case that *vHome* moves backward (toward a home node) because of detecting U-shaped path. At this moment, *vHome* is located on $(0, 3)$ because of detecting miss (this might be occurred when *Zigzag* protocol is not converged yet, or a target node moves after combining). However, in this case, *vHome* moves backward repeatedly, and our protocol is converged when *vHome* is located on $(0, 1)$. Even if *vHome* is moved by detecting

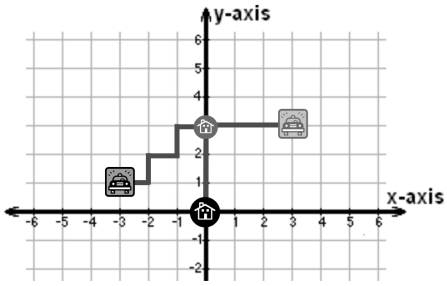


Figure 10: The case that *vHome* moves backward

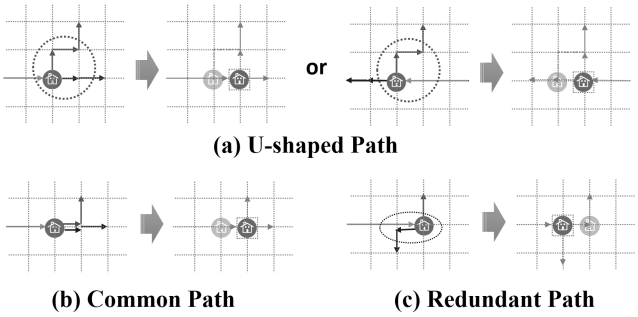


Figure 11: Three rules of our protocol

some common path as we introduced in Section 3.2.2., it can be move backward due to detecting of U-shaped path.

Finally we introduce one more simple rule to our protocol. By our protocol, the path to *vHome* from a home node is created. When *vHome* is not on the origin, if the direction just after *vHome* is the reverse direction of one just before *vHome*, *vHome* moves backward with one hop because the last hop of common path is redundant trivially.

Fig. 11 summarizes the local update's rule of our protocol.

3.2.5 A Problem of our proposed protocol and its solution

In this Section, we introduce a problem of our protocol and its solution. Fig. 12 shows an example of our protocol's incorrect convergence. The difference of convergence time between two protocol, our proposed protocol and *Zigzag* causes this incorrect convergence. However our proposed protocol is operated on each node locally, thus all nodes never recognize this miss convergence problem.

We can resolve this problem easily adopting *Zigzag* pro-

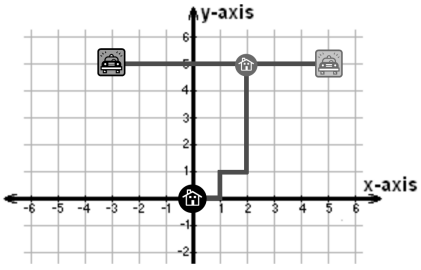


Figure 12: Incorrect convergence of our protocol

tocol between a home node and *vHome*. The important point of this adopting is the direction of *Zigzag* protocol. In this *Zigzag* protocol, *vHome* is resposible for a source node and a home node becomes a destination node. In the case of Fig. 12, if *vHome* adopts *Zigzag* toward a home node, we can find U-shaped path on *vHome* because the *zigzag-path* is created near *vHome* like $\{(2,5),(2,4),(1,4),(1,3)...\}$.

4 Summary and Future Works

In this paper, we proposed a protocol using local information only, which can construct 3-node rectilinear steiner tree. Our protocol uses a protocol *Zigzag* to make a shortest path to each target node from a home node. We modify *Zigzag* protocol slightly, and this enables the detecting some paths which can be combined. We introduce the notion of a virtual home node and some local update rules for combining path. We also explain a problem of our protocol and its simple solution.

However, in order to realize the correct convergence, our protocol assumes that a virtual home node is located in correct position and the path to a virtual node home from a home node correctly. Therefore, we consider the modification of our protocol to determine a virtual home node with only local information.

Moreover, we has to prove that our protocol creates the optimal steiner tree. Before a theoretical proof, we implement a simulator of our protocol, and evaluate our protocol on the various size of virtual grid networks. As a result of our simulator, we can find our protocol is operated correctly.

Finally, we should analyze the convergence time of our protocol.

References

[1] S. Giordano, Mobile Ad-Hoc Networks. Wiley, 2000.

[2] J. Zheng and A. Jamalipour, Wireless Sensor Networks : A Networking Perspective. Wiley-IEEE Press, 2009.

[3] Y. Xu, J. Heidemann, and D. Estrin, “Geography-informed energy conservation for ad hoc routing,” in Proceedings of the 7th Annual International Conference on Mobile Computing and Networking (MobiCom ’01), 2001, pp. 70—84.

[4] F. K. Hwang, “On Steiner Minimal Trees with Rectilinear Distance”, SIAM Journal on Applied Mathematics, Vol.30, No.1, pp.104-114, 1976.

[5] S. Takatsu, F. Ooshita, H. Kakugawa, and T. Masuzawa, ”Zigzag: Local-information-based self-optimizing routing in virtual grid networks,” Proceedings of the 33rd International Conference on Distributed Computing Systems (ICDCS) , pp. 358-368, July 2013.

自己安定平衡探索木構成アルゴリズム

伊藤 瑠美　大下福仁　角川裕次　増澤利光
大阪大学　大学院情報科学研究科

概要　大規模分散システムにおいて，大量の資源を管理するため適切な分散データ構造を用いることが重要である．データを効率的に探索するために有効なデータ構造として平衡探索木が挙げられる．また，システムの拡張性を保証するために故障やデータの挿入・削除など状況の変化に対応する必要がある．動的なシステムの保守には，任意の初期状況から望ましいシステム状況へと自律的に収束することを保証する自己安定性の適用が有効である．そこで，本稿では与えられた任意の探索木から平衡探索木を構成する自己安定アルゴリズムを提案する．

1　はじめに

近年，P2P システムやグリッド・コンピューティングのような大規模**分散システム**の利用が進んでいる．分散システムでは，ノードの離脱や参加，あるいは，故障や復旧などによるシステムの変化が多数発生すると考えられる．分散システムの変化に対する高度な適応性を実現する手法として**自己安定性**[1] が近年注目されている．自己安定性とは，任意の状況からアルゴリズムの実行を開始しても，問題の要求を満たす状況へ到達して安定するという性質である．

大規模分散システムにおいて，大量の資源を管理するため適切な**分散データ構造**を用いることが重要である．分散データ構造は，物理ネットワーク上に論理リンクを用いて**オーバーレイネットワーク**として実現する．仮想的な論理リンクを用いることによって，設計者が物理ネットワークの制約を受けずに，望ましい性質を持つ分散データ構造を構成することができる．データを効率的に探索するために有効なデータ構造として**平衡探索木**が挙げられる．代表的な平衡探索木として **B 木**(図 1) が挙げられる．B 木は各ノードが最大 m 個の子をもつ m 分木であり，さらに，すべての葉ノードのレベルが等しいという平衡条件を満たす．データは葉ノードにのみ格納され，各内部ノードが探索用のラベルを保持する．B 木の各内部ノードは m 個のラベルを保持するので，m を十分に大きく設定することにより，B 木の各ノードを 1 台の計算機で実現する分散データ構造に適すると考えられる．

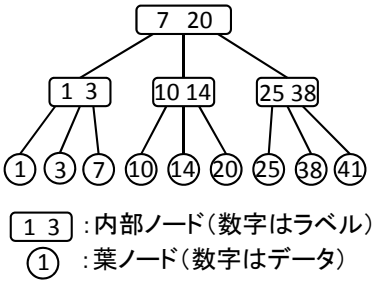


図 1　B 木

B 木に 1 データを挿入・削除すると，ノードのこの数に関する制約と平衡条件を満たすために，B 木の構造変更が必要なことがある．この構造変更は，挿入・削除を行った葉ノードから根ノードまで及ぶことがあり，分散データ構造では，大域的な構造変更を伴うことがある．一方，分散システムのデータ構造では，断続的なデータの挿入・削除や通信遅延などの障害が発生する可能性があり，大域的な構造変更を伴うデータの挿入・削除操作を逐次的に実行するのは非効率的である．そこで，分散 B 木では，データの挿入・削除操作を継続して受付可能とし，一時的には，子の数に関する制約や平衡条件が不成立となることを許して，局所的な構造変更のみで対応することが効率的であると考えられる (図 2)．継続的なデータ追加・削除を局所的に実行すると，平衡条件や次数の制約，データの順序，ラベルの内容など，B 木の厳しい制約が大きく破綻することがある．そこで，本稿では，与えられた任意の初期探索木から分散 B 木を構成する問題を解く自己安定アルゴリズムを提案する (図 3)．本稿で提案するアルゴリズムは，ラベル，データの配置，平衡状態が任意の初期木から局所的な情報のみを用いて自律的に分散 B 木を構成するアルゴリズムである．

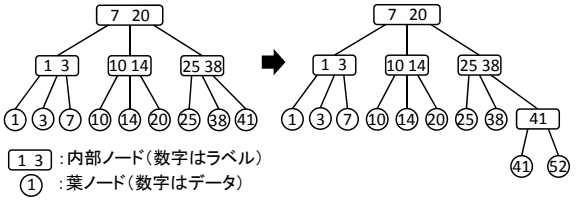


図 2　局所的なデータ挿入

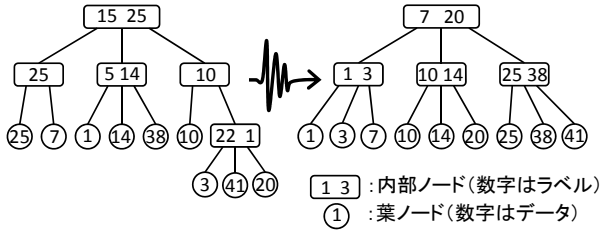


図 3　自己安定 B 木構成アルゴリズム

関連研究

これまで，分散データ構造を構成する様々な自己安定アルゴリズムが提案されてきた [3, 4, 5]．[3] では，ノード数 n に対して収束時間 $O(n)$，空間計算量 $O(\log n)$ で二分探索木を構成する自己安定アルゴリズムを提案している．構成される二分探索木は，初期構造に依存したものとなる．[4] では，与えられた任意の探索木を平衡化する自己安定アルゴリズムを提案している．[4] のアルゴリズムは，平衡化のため探索木内の辺を交換する操作を行う．辺の交換は各ノードが保持するラベルの変更を伴うため，探索の非効率化を引き起こす虞がある．

本稿の構成

本稿の構成は以下のとおりである．まず 2 節において諸定義を行う．次に，3 節において提案アルゴリズムを説明し，最後に 4 節において本研究のまとめと今後の課題を述べる．

2 諸定義

ある定数 m に対して，B 木において，根または葉以外の各ノードは，$\lceil m/2 \rceil$ 以上 m 以下の子をもつとする．また，根は 2 以上 m 以下の子をもつとする．根から各葉ノードまでの距離（深さ）は全て等しいという平衡条件を満たす．データは葉ノードにのみ格納され，各葉ノードは，1 つのデータを格納している．全ての葉ノードがデータが昇順となるように整列している．各内部ノードは複数のラベルを格納している．あるノード u の i 番目のラベルは，u に接続する i 番目と $i+1$ 番目の子を根とする部分木に含まれるデータの境界値である．

分散 B 木は，各ノードをプロセスに対応させることにより実装する．本稿では，分散 B 木のノード数の変化に応じて，プロセスの追加・削除が可能であるとする．各ノードは隣接ノードとのみ通信可能である．通信モデルとして，**局所共有メモリモデル**[2] を用いる．局所共有メモリモデルでは，変数の読み込みによって隣接ノードと通信を行う．各ノードは自身の変数に書き込みが可能で，隣接ノードの変数を読み込み可能である．

3 分散 B 木構成自己安定アルゴリズム

本節では，任意の初期探索木から分散 B 木を構成するアルゴリズムを紹介する．ただし，初期状況において各内部ノードがもつ子の数は m 以下であるとする[1]．

提案アルゴリズムにおいて，各ノード u は以下の変数を保持する．

- $u.child[i]$: u の i 番目の子のノード ID を格納する．
- $u.label[i]$: u の i 番目のラベルを格納する．
- $u.children$: u の子の数を格納する．
- $u.height$: u の高さ (子孫の葉ノードまでの最大距離) を格納する．

ノード u の子の数が $u.children$ のとき，記憶すべき部分木間の境界値は $u.children$　1 個となる．提案アルゴリズムでは，各ノードを根とする部分木内の最大データを記憶しておく必要がある．そこで本稿では，ノード u を根とする部分木内の最大データを $u.children$ 番目のラベルとして $u.child[u.chiledren]$ に格納しておくこととする．

さらに，ノード u に対して以下の述語を定義する．

- $h\_correct(u)$: u の高さが子の高さと矛盾しないとき $true$ を返す．
- $balanced(u)$: u を根とする部分木が平衡なとき $true$ を返す．
- $ordered(u)$: u を根とする部分木中のデータが昇順に整列されているとき $true$ を返す．
- $labeled(u)$: u のラベルが子のラベルと矛盾しないとき $true$ を返す．

このアルゴリズムは，次の 3 つの自己安定副アルゴリズムから構成される．

1. **データ再配置副アルゴリズム**：探索木に含まれるデータの並びが昇順になるように再配置する．
2. **ラベル調整副アルゴリズム**：各ノードのラベルを正しく設定する．
3. **平衡化副アルゴリズム**：部分木の高さの差を減少させ，平衡探索木を構成する．

提案アルゴリズムでは，これら 3 つの自己安定副アルゴリズムを並行動作させる．各ノード u の動作 (Algorithm1) は次の通りである．

1. ノード u の子の数が m を超えている場合，適切なラベルを親に転送し，自ノードを分割する (図 4)．
2. ノード u の子を根とする各部分木においてデータが昇順に整列され，根が正しいラベルを保持している場合，
   - ノード u を根とする部分木においてデータが昇順に整列されていない場合，データ再配置副アルゴリズム (Algorithm2) を実行する．

[1] 子の数をこのように制限しても，平衡条件を緩めているため，挿入や削除を局所的に実行可能である．

- ノード u のラベルが u の子のラベルと矛盾する場合，ラベル調整副アルゴリズム (Algorithm3) を実行する．

3. ノード u が正しい高さ値を保持し平衡化でなく，かつ正しい高さ値を保持し平衡な v が u の子として存在する場合，平衡化副アルゴリズム (Algorithm4) を実行する．

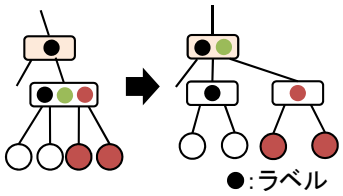


図 4　子が m を超えた場合の処理

3.1　データ再配置副アルゴリズム

データ再配置副アルゴリズムでは，葉ノードに格納されたデータが昇順となるように，データの再配置をおこなう．

各ノード u の動作 (Algorithm2) は次の通りである．各ノードは i 番目と $i+1$ 番目の子を参照する (1　i　$u.children$　1)．i 番目の子の最後のラベル，つまり i 番目の子を根とする部分木の最大値と，$i+1$ 番目の子の最後のラベルを比較し，前者の方が大きい場合に，該当するデータの交換を実行する (図 5)．

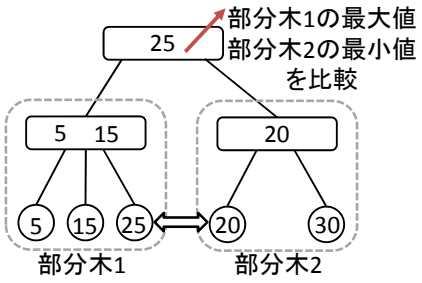


図 5　データ再配置副アルゴリズム

3.2　ラベル調整副アルゴリズム

ラベル調整副アルゴリズムでは，各ノードが正しくラベルを設定する．各ノード u の動作 (Algorithm3) は次の通りである．

各ノードは i 番目の子を参照する (1　i　$u.children$)．i 番目の子の最後のラベル，つまり i 番目の子を根とする部分木の最大値を，自身の i 番目のラベルとして設定する (図 6)．

Algorithm 1 分散 B 木構成自己安定アルゴリズム：ノード u の動作

Constants:

m: 子の数の上限

$u.id$: u の ID

Variables:

$u.child$: u の子の集合

$u.child[i]$: u の i 番目の子

$u.label[i]$: u の i 番目のラベル

$u.children$: u の子の数

$u.height$: u の高さ

Predicates:

$h\_correct(u)$: u の高さが子と矛盾ないとき $true$ を返す

$balanced(u)$: u を根とする部分木が平衡なとき $true$ を返す

$ordered(u)$: u を根とする部分木中のデータが整列されているとき $true$ を返す

$labeled(u)$: u のラベルが子と矛盾ないとき $true$ を返す

Function:

repeat forever do

1: **if** $u.children > m$ **then**
2:　ラベルを親に転送し，自ノードを分割;
3: **if** $\forall v \in u.child : (ordered(v) = true$ **and** $labeled(v) = true)$ **then**
4:　**if** $ordered(u) = false$ **then**
5:　データ再配置副アルゴリズム (Algorithm2) を実行;
6:　**else if** $labeled(u) = false$ **then**
7:　ラベル調整副アルゴリズム (Algorithm3) を実行;
8: **if** $h_correct(u) = false$ **then**
9:　**if** u は内部ノード **then**
10:　$u.height \leftarrow 1 + \max v.height$;
11:　**else**
12:　$u.height \leftarrow 0$;
13: **else if** $balanced(u) = false$ **and** $\exists v \in u.child : (balanced(v) = true$ **and** $h\_correct(v) = true$ **then**
14:　平衡化副アルゴリズム (Algorithm4) を実行;

Algorithm 2 データ再配置副アルゴリズム：ノード u の動作

Function:

1: **for** $i = 1 \ldots u.children \quad 1$ **do**
2: **if** $(u.child[i]).label[(u.child[i]).children]$ $(u.child[i+1]).label[1] > 0$ **then**
3: i 番目の部分木の最大データと $i+1$ 番目の部分木の最小データを交換;

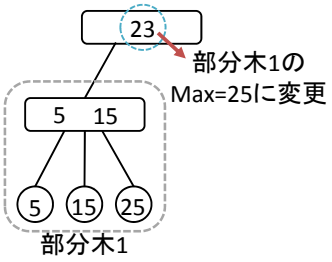


図 6　ラベル調整副アルゴリズム

Algorithm 3 ラベル調整副アルゴリズム：ノード u の動作

Function:

1: **for** $i = 1 \ldots u.children$ **do**
2: $u.label[i] \leftarrow (u.child[i]).label[(u.child[i]).children]$;

3.3　平衡化副アルゴリズム

平衡化副アルゴリズムでは，各ノードが高さの異なる子をもつ場合，高さの差を減少させるように探索木の変形を行う．各ノード u の動作 (Algorithm4) は次の通りである．

ノード u の子のうち，平衡かつ高さの正しいすべてのノード v に対して，ノード v のラベルを u のラベルとして追加する．さらに，v の子を u の子として追加する (図 7)．

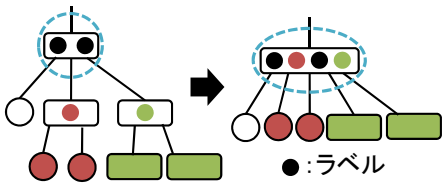


図 7　平衡化副アルゴリズム

Algorithm 4 平衡化副アルゴリズム：ノード u の動作

Function:

1: **for all** $v \in u.child$(ただし，v は $balanced(v) = true$ かつ $v \neq \arg\min_{w \in v.child} w.height$ を満たす) **do**
2: v のラベルを u 自身のラベルとして追加;
3: v の各子を u 自身の子として追加;

4　まとめと今後の課題

本稿では，任意の探索木から平衡探索木である分散 B 木を構成する自己安定アルゴリズムを提案した．提案アルゴリズムでは，データ再配置副アルゴリズム，ラベル調整副アルゴリズム，平衡化副アルゴリズムの 3 つの副アルゴリズムを並行して実行するものである．ラベル調整副アルゴリズム及び平衡化副アルゴリズムでは，初期木の高さを h とすると，収束時間が $O(h)$ となることが予測できる．また，データ再配置副アルゴリズムでは，初期木のデータの逆転数を r とすると，収束時間が $O(hr)$ となると予測できる．収束時間や空間計算量に関する正確な評価は今後の課題とする．

また，提案手法では，データ再配置副アルゴリズムにおいて，データの交換方法を定義していない．葉ノード同士をポインタで接続することで，葉ノード間でのデータ交換を直接可能にし，アルゴリズムの簡略化を図りたい．

参考文献

[1] S. dolev, Self-stabilization. The MIT Press, 2000.
[2] E. W. Dijkstra.: Self-stabilizing systems in spite of distributed control. Commun. ACM 17(11), 643-644 (1974).
[3] D. Bein, A. K. Datta, V. Villain.: Snap-stabilizing optimal binary search tree. Proceedings of the 7th international conference on Self-Stabilizing Systems (2005).
[4] E. Bampas, A. Lamani, F. Petit, M. Valero.: Self-stabilizing balancing algorithm for containment-based trees. Proceedings of the 15th international conference SSS (2013).
[5] S. Bianchi, A. K. Datta, P.Felber, M. Gradinariu.: Stabilizing peer-to-peer spatial lters. In: ICDCS, p. 27. IEEE Computer Society (2007)

A hybrid approach of optimization and sampling for robust portfolio selection

Omar Rifki and Hirotaka Ono
Department of Economic Engineering
Kyushu University, Fukuoka, Japan
omar.rifki.910@s.kyushu-u.ac.jp, hirotaka@econ.kyushu-u.ac.jp

***Abstract*—Dealing with ill-defined problems, where the actual values of input parameters are unknown or not directly measurable, is generally not an easy task. In this paper, we propose a hybrid metaheuristic approach, incorporating a sampling-based simulation module, in order to enhance the robustness of the final solutions. Empirical application to the classical mean-variance portfolio optimization problem, which is known to be extremely sensitive to noises in asset means, is provided through a genetic algorithm solver. Results of the proposed approach are compared with that specified by the worst-case scenario.**

***Keywords*-robustness; simulation model; hybridization; evolutionary algorithm; portfolio optimization;**

I. Introduction

Metaheuristics (MH) are a general class of approximate algorithms, particularly useful for solving difficult optimization problems. Usually simpler to implement compared with gradient-based techniques, they selectively guide the search in succeeding iterations to produce high quality solution(s). The prominent classical families of metaheuristics, that have been used through the years, are tabu search, evolutionary algorithms such as genetic algorithms and evolutionary strategies, simulated annealing, iterated local search and ant colony optimization among others. However, in recent years metaheuristic algorithms combining algorithmic ideas from diverse metaheuristics and a variety of different disciplines like computer science, operations research and artificial intelligence, have been widely and frequently reported. Such algorithms, so-called hybrid metaheuristics, do not restrict their attention solely to the classical metaheuristic families, but extend the scope of its applicability to wide algorithmic areas. The goal is typically to make meaningful improvements either in solution quality or in terms of running time. According to the famous no free lunch theorem (NFL), roughly stated as follows "averaged over the space of all the possible problems all search algorithms perform equally", no single algorithm will outperform random search in average. That is to say, a standard metaheuristic will eventually gain in performance when combined with efficient heuristics and handy problem-specific knowledge. Thereby, hybridization could be a promising tool in order to improve the relative efficiency of the MH in hand for the target optimization problem. The following figure[1] depicts this view for unmixed, hybrid MHs and problem-oriented methods. The last ones have the best efficiency since they are specifically designed for the purpose of solving the target problem. The first ones are instead quite monotonous involving little problem-specific knowledge.

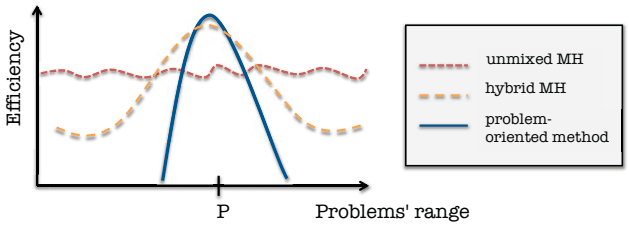


Figure 1. Current view of hybrid MHs

The main focus of MH hybridization process is basically turned to finding efficient good approximate solutions. However, finding robust solutions, which are less sensitive to small changes in the problem variables, can be highly important for effective reasons. In fact, from a practical point of view, solutions of real-world optimization problems are expected to be, not only optimal, but also insensitive to small changes that affect problem variables, whether endogenous or exogenous. Otherwise, a sensitive solution may not be attainable in practice, due to the difficulty of meeting the theoretical assumptions. In this paper, we propose a hybrid metaheuristic approach for solving optimization problems where some exogenous parameters are unknown or unkownable. The intent of hybridization is to enhance the robustness of the final solutions, i.e. the design variables, to changes in those parameters. In other terms, solutions that do not exhibit larger departures when slight changes affect exogenous parameters will be favoured.

The conception of our hybrid MH embeds two phases, in addition to the MH itself, i.e. the algorithmic part, a simulation procedure is performed. Theoretically, any MH, independently of its structure, e.g. population-based or not, related to local search or not, can be utilized within the

[1]This figure is redrawn from [1] and [2] who addressed the case of evolutionary algorithms.

proposed approach. The combination between the MH and the simulation is actually of high level nature, allowing each part to retain its own identity. The aim of the simulation procedure is to identify quality robust solutions among a set of solutions. The overall process comprises, first heuristically solving the problem for several instances of the uncertain parameters, thereafter, evaluating the solutions' performance under a large enough number of samples of the uncertain parameters, slightly and randomly derived from a nominal case. Percentage of being top-ranked solution across the scenarios, plus the across-scenario average of *performance ratio*, i.e. ratio between the solution evaluation and the top-ranked evaluation for the related scenario, are taken as measures of robustness.

The simulation model is described in the next section, while the process of hybridization is presented in Section III. A practical application involving a hybrid genetic algorithm to the financial problem of mean-variance portfolio optimization is provided in Section IV. Section V concludes this study.

II. Simulation Module

This section briefly introduces the notation used in the paper, and subsequently describes the simulation module to be integrated later in the hybrid MH. Special emphasis is given to finding a minimum sample size to guarantee reliable estimates, in subsection B.

A. Notation and model specification

Consider the following constrained optimization problem,

$$\max_x f(\tilde{\theta}, x) \quad \text{subject to} \quad x \in \mathcal{C} \subseteq \mathbb{R}^n,\ \tilde{\theta} \in \Delta \subseteq \mathbb{R}^l, \quad (1)$$

where x denotes a vector of design variables constrained to a set $\mathcal{C} \subseteq \mathbb{R}^n$ and $\tilde{\theta} \in \Delta \subseteq \mathbb{R}^l$ is a vector of random variables representing uncertain exogenous parameters, while $f(\tilde{\theta}, x) : \Delta \times \mathcal{C} \to \mathbb{R}$ is an objective function assumed to be scalar-valued. This problem is not well-defined, since it involves the random parameters $\tilde{\theta}$ that lead to an ambiguous function $f(\tilde{\theta}, x)$. We make it well-defined by assuming that the parameters $\tilde{\theta}$ vary within an uncertainty set Θ, defined as a norm-bounded ball with a center θ_0,

$$\Theta \equiv \{\theta \in \Delta : ||\theta - \theta_0||_p \leq \xi\} = B_\xi(\theta_0) \subseteq \mathbb{R}^l,$$

where $||.||_p$ denotes the l_p norm in $\mathbb{R}^l$, e.g. $p = 1, 2, \infty$. We shall call θ_0 the *nominal value* of $\tilde{\theta}$.

The focus of our simulation is to detect optimal design variables, that have desirable robustness properties within the uncertainty set Θ. To this purpose, given a set of optimal solutions $\mathcal{X} = \{x_1, x_2, x_3, ..., x_k\}$, we proceed by studying their behaviour in terms of performance throughout a number of randomly generated samples of $\tilde{\theta} \in \Theta$, that we shall call *scenarios*. N independent and identically distributed (i.i.d.) scenarios of $\tilde{\theta}$ are drawn, $\theta_1, \theta_2, ..., \theta_N \in \Theta$. To measure robustness of an instance $x \in \mathcal{X}$, we have chosen two measures $R_1^{\mathcal{X}}(x)$ and $R_2^{\mathcal{X}}(x)$. The first one reports the percentage across scenarios of being *top-ranked solution*, i.e. having the best f value compared with the other solutions, for the corresponding scenario. Using the following indicator function,

$$I_{\mathcal{X}}(\theta, x) = \begin{cases} 1, & \text{if } f(\theta, x) = \max\limits_{y \in \mathcal{X}} f(\theta, y) \\ 0, & \text{otherwise} \end{cases}$$

$$\forall \theta \in \{\theta_1, \theta_2, ..., \theta_N\},\ \forall x \in \mathcal{X},$$

we can obtain an estimate of $R_1^{\mathcal{X}}(x)$,

$$\widehat{R_1^{\mathcal{X}}}(x) = \frac{1}{N} \sum_{i=1}^{N} I_{\mathcal{X}}(\theta_i, x) \quad \forall x \in \mathcal{X}, \quad (2)$$

which is equivalent to,

$$\widehat{R_1^{\mathcal{X}}}(x) = \frac{M_{\mathcal{X}}(x)}{N} \quad \forall x \in \mathcal{X}, \quad (3)$$

with $M_{\mathcal{X}}(x)$ is the number of scenarios θ, such that $x \in \arg\max\limits_{y \in \mathcal{X}} f(\theta, y)$. The second measure $R_2^{\mathcal{X}}(x)$ starts by computing for each scenario θ_i the ratio: $f(\theta_i, x)/f_{max}^{\mathcal{X}}(\theta_i)$. $f(\theta_i, x)$ is the evaluation of x correspondingly to θ_i, and $f_{max}^{\mathcal{X}}(\theta_i) = \max\limits_{y \in \mathcal{X}} f(\theta_i, y)$ is the maximum function value reached for θ_i. Afterwards, these ratios are averaged over all scenarios. An estimate can be written as,

$$\widehat{R_2^{\mathcal{X}}}(x) = \frac{1}{N} \sum_{i=1}^{N} \frac{f(\theta_i, x)}{f_{max}^{\mathcal{X}}(\theta_i)}.$$

$f_{max}^{\mathcal{X}}(\theta_i)$ can be also expressed as,

$$f_{max}^{\mathcal{X}}(\theta_i) = \sum_{y \in \mathcal{X}} I_{\mathcal{X}}(\theta_i, y) f(\theta_i, y).$$

The reason of using the first measure is intuitive. It relates the robustness of a solution to the number of times it is optimum (according to $\mathcal{X}$) for a number of random samples $\tilde{\theta}$. The advantage of the second measure is however to take into account the relative span of each solution x to the top-ranked solution across scenarios. Indeed this ratio may be desirable to know if a solution is rarely ranked best, but frequently exhibits a small gap to the top-ranked solutions.

	$\mathcal{X}$				
	x_1	x_2	...	x_k	
(I_1) scenario θ_1	$f(\theta_1, x_1)$	$f(\theta_1, x_2)$	...	$f(\theta_1, x_k)$	$\rightarrow f_{max}^{\mathcal{X}}(\theta_1)$
(I_2) scenario θ_2	$f(\theta_2, x_1)$	$f(\theta_2, x_2)$	...	$f(\theta_2, x_k)$	$\rightarrow f_{max}^{\mathcal{X}}(\theta_2)$
...	...	...	...	...	
(I_N) scenario θ_N	$f(\theta_N, x_1)$	$f(\theta_N, x_2)$	...	$f(\theta_N, x_k)$	$\rightarrow f_{max}^{\mathcal{X}}(\theta_N)$
$R_1^{\mathcal{X}}(x)$	-	-	...	-	
$R_2^{\mathcal{X}}(x)$	-	-	...	-	

Table I. Description of the simulation module

Table I illustrates the overall simulation module. In performing the simulation, three elements need to be set:

- $\mathcal{X}$, the set of optimal solutions to be compared. This set will be generated according to the hybridization process described in Section 3,
- N, the number of randomly generated samples of $\tilde{\theta}$, which will be discussed in the following section,
- and the distribution of the random draw of $\tilde{\theta} \in \Theta$. This includes the shape, e.g. Gaussian, uniform, etc, and the magnitude, i.e. the parameter ξ of $\Theta = B_\xi(\theta_0)$, of the noise distribution. The related choice will be mentioned in the application example.

B. Minimum sample size

The estimate (3) of $R_1^{\mathcal{X}}$, formally known as *relative frequency*, is empirically derived from the actual data, i.e. the used scenarios. A crucial question is to know how many samples of scenarios have to be drawn in our simulation in order to obtain reliable estimates of $R_1^{\mathcal{X}}$ with a high probability. In other words, we are seeking a minimum number of N such that the probability value of the difference $|\widehat{R_1^{\mathcal{X}}}(x) - R_1^{\mathcal{X}}(x)| \leq \epsilon$ with $\epsilon \in (0,1)$, is sufficiently high. Given a *margin error* $\epsilon \in (0,1)$ and a *confidence level* $\delta \in (0,1)$, we will make use of this inequality,

$$\mathbb{P}(|\hat{p}_{\mathcal{X}}(x) - p_{\mathcal{X}}(x)| \leq \epsilon) \geq 1 - \delta, \tag{4}$$

with $\hat{p}_{\mathcal{X}}(x) = \widehat{R_1^{\mathcal{X}}}(x)$ and $p_{\mathcal{X}}(x) = R_1^{\mathcal{X}}(x)$. Since for all $i \in [|1, N|]$ the θ_i's are drawn independently, the event whether a solution x is top-ranked solution or not for a scenario θ_i, which can be expressed as $I_{\mathcal{X}}(\theta_i, x)$, is viewed as an independent Bernoulli trial. Thus, the estimated probability of the global process, which is a binomial process, i.e. sum of independent Bernoulli trials, can be expressed in accordance with (2) as $\hat{p}_{\mathcal{X}}(x) = \frac{1}{N}\sum_{i=1}^{N} I_{\mathcal{X}}(\theta_i, x)$. The expectation and the variance of the general process are,

$$\mathbb{E}(\hat{p}_{\mathcal{X}}(x)) = \frac{\mathbb{E}(\sum_{i=1}^{N} I_{\mathcal{X}}(\theta_i, x))}{N} = p_{\mathcal{X}}(x),$$
$$Var(\hat{p}_{\mathcal{X}}(x)) = \frac{Var(\sum_{i=1}^{N} I_{\mathcal{X}}(\theta_i, x))}{N^2} = \frac{\sigma^2_{\mathcal{X}}(x)}{N},$$

where $\sigma^2_{\mathcal{X}}(x)$ is the variance of the parameter $I_{\mathcal{X}}(\theta_i, x)$. Note that, $p_{\mathcal{X}}(x)$ is actually the probability of success in each trial $I_{\mathcal{X}}(\theta_i, x)$. By directly applying Chebyshev's inequality, which is formally stated as follows, given a random variable Y,

$$\mathbb{P}(|Y - \mathbb{E}(Y)| \geq \epsilon) \leq \frac{Var(Y)}{\epsilon^2}, \ \forall \epsilon \in (0,1),$$

we obtain the following inequality,

$$\mathbb{P}(|\hat{p}_{\mathcal{X}}(x) - p_{\mathcal{X}}(x)| \leq \epsilon) \geq 1 - \frac{\sigma^2_{\mathcal{X}}(x)}{N\ \epsilon^2}$$
$$\geq 1 - \frac{p_{\mathcal{X}}(x)\ (1 - p_{\mathcal{X}}(x))}{N\ \epsilon^2}.$$

The corresponding confidence level is then,

$$\delta = \frac{p_{\mathcal{X}}(x)\ (1 - p_{\mathcal{X}}(x))}{N\ \epsilon^2}.$$

Taking into account, $p_{\mathcal{X}}(x)\ (1 - p_{\mathcal{X}}(x)) \leq \frac{1}{4}$, the sample size is bounded, as follows,

$$N \geq \frac{1}{4\epsilon^2\delta}. \tag{5}$$

This bound is known as *Bernoulli bound*. Its expression is actually given in the well-known Bernoulli's theorem of weak law of large number. A tighter bound derived from Hoeffding's inequality[2], called *Chernoff bound*, improves the minimum sample size. Hoeffding's inequality in our case is written as,

$$\mathbb{P}(\Big|\frac{1}{N}\sum_{i=1}^{N} I_{\mathcal{X}}(\theta_i, x) - \mathbb{E}(\frac{1}{N}\sum_{i=1}^{N} I_{\mathcal{X}}(\theta_i, x))\Big| \leq \epsilon) \geq 1 - 2e^{-2\epsilon^2 N}.$$

The confidence level here is,

$$\delta = 2\exp(-2\epsilon^2 N),$$

which subsequently leads to the expression of the Chernoff bound,

$$N \geq \frac{ln(2/\delta)}{2\epsilon^2}. \tag{6}$$

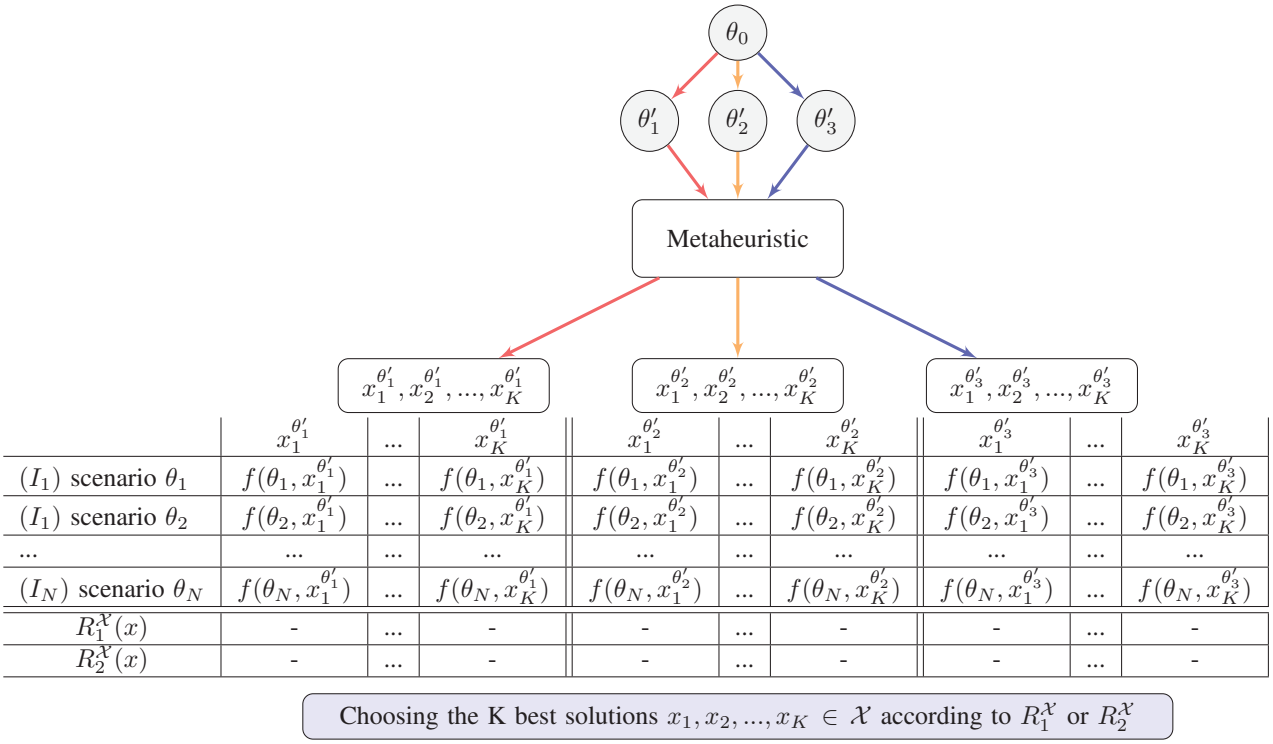
The bound (5) is specified for any random variable, while the bound (6) requires a sum of random variables. The main disadvantage of (5) is that, it requires a large minimum sample size. For instance, Bernoulli and Chernoff bound for $\epsilon = \delta = 0.1\%$ are respectively $N = 2.5\ 10^8$ and $N \approx 3.80\ 10^6$. Tight and rigorous bounds as Chernoff bound are more desirable, since they allow for exact approximation of the minimum value of N necessary for reliable estimates.

III. HYBRIDIZATION

Hybrid MHs are classified according to several aspects. The *level of hybridization*: deep combination or high level cooperation, the *order of execution*: batch, interleaved or parallel and the *control strategy*: integrative or collaborative, are among others classification criteria of heuristic systems[3]. Control strategy concerns the nature of relationship between the hybrid MH parts. It is integrative if one part is subordinate to the other, as in memetic algorithms where the local search improvement is embedded into genetic algorithm procedures. In the collaborative way however, there is an exchange of information in a synergetic manner without any subordination. Our approach falls into the second class. On one side the simulation technique presented in the previous section, and separately on the other side, the MH algorithm. Before running the simulation, the MH will be performed several times for several instances of the uncertain parameters. Let V denote the number of those runs. For

[2]The formal proof of this inequality can be found in [3].

[3]This taxonomy of classification is mentioned with respect to [4].



	$x_1^{\theta'_1}$	...	$x_K^{\theta'_1}$	$x_1^{\theta'_2}$	...	$x_K^{\theta'_2}$	$x_1^{\theta'_3}$	...	$x_K^{\theta'_3}$
(I_1) scenario θ_1	$f(\theta_1, x_1^{\theta'_1})$	...	$f(\theta_1, x_K^{\theta'_1})$	$f(\theta_1, x_1^{\theta'_2})$	...	$f(\theta_1, x_K^{\theta'_2})$	$f(\theta_1, x_1^{\theta'_3})$	...	$f(\theta_1, x_K^{\theta'_3})$
(I_1) scenario θ_2	$f(\theta_2, x_1^{\theta'_1})$	...	$f(\theta_2, x_K^{\theta'_1})$	$f(\theta_2, x_1^{\theta'_2})$	...	$f(\theta_2, x_K^{\theta'_2})$	$f(\theta_2, x_1^{\theta'_3})$	...	$f(\theta_2, x_K^{\theta'_3})$
...	...	...	...	...	...	...	...	...	...
(I_N) scenario θ_N	$f(\theta_N, x_1^{\theta'_1})$	...	$f(\theta_N, x_K^{\theta'_1})$	$f(\theta_N, x_1^{\theta'_2})$	...	$f(\theta_N, x_K^{\theta'_2})$	$f(\theta_N, x_1^{\theta'_3})$	...	$f(\theta_N, x_K^{\theta'_3})$
$R_1^{\mathcal{X}}(x)$	-	...	-	-	...	-	-	...	-
$R_2^{\mathcal{X}}(x)$	-	...	-	-	...	-	-	...	-

Figure 2. General scheme of the proposed hybrid algorithm

example, in Figure 2 $V = 3$, the MH is run three times for three randomly generated variables $\theta'_1, \theta'_2, \theta'_3 \in \Theta$ with θ_0 the nominal value. The overall solutions outputted by the MH runs are grouped to be considered as inputs for the simulation module. The set of input solutions $\mathcal{X}$ considered here according to Figure 2 is,

$$\mathcal{X} = \{x_1^{\theta'_1}, x_2^{\theta'_1}, ..., x_K^{\theta'_1}, x_1^{\theta'_2}, x_2^{\theta'_2}, ..., x_K^{\theta'_2}, x_1^{\theta'_3}, x_2^{\theta'_3}, ..., x_K^{\theta'_3}\}.$$

If we suppose that the MH generates $K \geq 1$ solutions, the final output of the hybrid MH will be also of size K, more specifically the K best solutions sorted according to $R_1^{\mathcal{X}}$ or $R_2^{\mathcal{X}}$. Notice that, it is obviously possible to consider the nominal value θ_0 side by side with $\theta'_1, \theta'_2, \theta'_3$, and the simulation can be computationally costly if the number of required scenarios is quite large. Each new scenario adds at least a cost of $V \times K$ evaluations. Thus, finding a mechanism that may reduce the number of sampled scenarios is of high interest, even if it is not considered in this paper.

Facing optimization under uncertainty, one well-applied approach is the worst-case analysis, known also as max-min, and extensively used in robust optimization. It is based on optimizing the original problem under the worst realization of the uncertain parameters contained within a specially constructed set, called the *uncertainty set*. This approach is preferred to other optimization approaches under uncertainty, as dynamic and stochastic methodologies, since it is more tractable computationally, i.e. the optimization is performed only one time. We shall compare our simulation result: solutions ranked according to $R_1^{\mathcal{X}}$ or $R_2^{\mathcal{X}}$, to the results indicated by the the worst-case scenario θ_{wc} among the set $\{\theta_1, \theta_2, ..., \theta_N\}$ seen here as an uncertainty set. To spot θ_{wc}, we seek given the set of the θ_i's, a lower value of the following expression[4],

$$\sum_{x \in \mathcal{X}} f(x, \theta_i), \quad \forall i \in [|1, N|]. \tag{7}$$

This comparison will allow us to evaluate further the differences between the results of our approach and those suggested by the worst-case view.

IV. Application

In this section, we apply our hybrid metaheuristic construction to the problem of portfolio optimization. Originally proposed by Markowitz in 1952 [5], the problem of finding the best financial investment strategy can be pinned down mathematically, according to Markowitz' model, as an optimization problem based on two criteria, minimizing the risk while maximizing the expected return of an investment which consists of several financial assets, e.g. stock securities, bonds, real estate investment, options, etc. It is called the mean-variance model (MV), as the

[4]If our problem is a minimization problem, we will seek a higher value.

investment risk is measured by variances and covariances of asset returns. The model considered in this section is the standard single-period MV problem, stated for n risky assets as,

$$\begin{aligned} \underset{w}{\text{maximize}}\ \{EU(w)\} &= \{\mu' w - \lambda w' \Sigma w\} \qquad (8) \\ &= \{\sum_{i=1}^{n} w_i \mu_i - \lambda \sum_{i=1}^{n}\sum_{j=1}^{n} w_i w_j \sigma_{ij}\} \\ \text{subject to}\ \sum_{i=1}^{n} w_i &= 1, \quad \text{and} \quad \forall i \in [|1,n|],\ 0 \le w_i \le 1 \end{aligned}$$

where $w \in \mathbb{R}^n$ is the n-vector of weights corresponding to the decision variable, each w_i is the fraction held in the i-th asset. The expected return (mean) of the i-th asset and the covariance between the return of the i-th and j-th assets are respectively denoted by μ_i and σ_{ij}, such that $\sigma_{ii} = \sigma_i^2$ is the variance of the i-th asset. Accordingly, $\mu = (\mu_i)_i$ and $\Sigma = (\sigma_{ij})_{i,j}$ are the n-vector of means and the $n \times n$ covariance matrix in the same order. The objective function is designated by the expected utility function $EU(w)$, and the parameter λ indicates the degree of the investor's risk aversion. The higher λ the more the investor is risk-averse. Nevertheless, the MV analysis suffers from the drawback of extreme sensitivity to input's errors, especially concerning asset means μ. This is an issue because the model inputs are predominantly based on statistical estimations from historical data, which may induce potential and significant errors in asset means and return covariances. Actually, this sensitivity issue is considered a major barrier for POP to be effective in real-life situations. Several studies have addressed this problem: Jobson and Korkie [6], Michaud [7], Best and Grauer [8], Chopra and Ziemba [9] and Broadie [10].

A. Genetic Algorithm solver

Concerning the MH choice, we apply Genetic Algorithm (GA), which is a well-known population-based metaheuristic grounded on the darwinian natural selection principle – *survival of the fittest*. In GA, the population of individuals evolves through three main operators to fit a goal formulated by the fitness function, a function that evaluates the accuracy of the evolved individuals to the problem in hand. The first operator is *selection*, which selects parent individuals intended to generate the next generation via two evolution operators which are *crossover* that merges features of the parents and *mutation* that slightly changes them. GA is typically applied to global optimization problems, especially solving complex nonlinear problems where exact solutions are difficult to obtain. The choice of GA, in the current application, is due to its superior performance reported for solving MV models in comparison with other MHs. For instance, Chang *et al.* [11] compared three MHs, namely Tabu search, Simulated Annealing and GA, for solving a cardinality constrained MV optimization problem. They found that in the unconstrained case, i.e. without cardinality constraints, which corresponds to our target problem (8), GA gives the best approximate solutions with an almost zero mean percentage error. For more information about GA based applications to portfolio optimization, the reader is referred to the surveys [12] [13] [14] which cover single and multi-objective GA-based applications.

Since the optimization problem (8) is continuous, we adopt a real-valued representation rather than the original binary-string representation that is well suited for discrete problems. Simulated binary crossover (SBX) and polynomial mutation (PM) are equally employed. These two operators, proposed respectively by Deb and Agrawal [15] and Deb [16] are specifically designed for real-valued encodings. SBX is the real domain analog of single-crossover operator of binary GA in terms of search power. It uses for offspring production a probability distribution similar to that of binary single-crossover. PM, on the other hand, is a nonuniform mutation where the underlying distribution is similar to that of SBX, i.e, polynomial. Both operators have been shown to yield improved results over real-valued encodings, according to [15] and [16]. For the selection procedure, we used a binary tournament operator with elitism. The binary tournament compares the fitness of a set of randomly chosen individuals, it retains after the best ranked solution, i.e. the one with the highest fitness. This process is repeated until the mating pool, from which the next generation will be drawn, is complete. The elitism is applied by keeping the two fittest solutions of the population for the next generation. The following table summarizes the GA operators and the parameter values used in the current application.

GA Parameter	Value
Population size	100
Generations	300
Selection	tournament (size = 2)
Crossover	SBX
Mutation	polynomial
Crossover probability	0.25
Mutation probability	0.01

Table II. GA parameters

B. Data

We use the following dataset, taken from Michaud [17, p. 16]. It describes monthly sampled expected returns and covariances for six developed countries. The data is given over a period of 18 years ($T = 216$), from January 1978 to December 1995.

		Asset means	Standard deviation
1	Canada Equity	.39	5.50
2	France Equity	.88	7.029
3	Germany Equity	.53	6.220
4	Japan Equity	.88	7.039
5	U.K. Equity	.79	6.010
6	U.S. Equity	.71	4.300
7	U.S. Bonds	.25	2.010
8	Europe Bonds	.27	1.558

	Correlation matrix							
	1	2	3	4	5	6	7	8
1	1							
2	.4099	1						
3	.2999	.6199	1					
4	.2500	.4200	.3500	1				
5	.5799	.5401	.4798	.3999	1			
6	.7099	.4399	.3402	.2200	.5599	1		
7	.2596	.2200	.2703	.1399	.2500	.3598	1	
8	.3300	.2600	.2805	.1603	.2903	..4207	.9191	1

Table III. Historical means and return correlation matrix

C. Methodology and results

We focus on errors in asset means, since their impact can be 10 times or more important than those in variances and covariances, in terms of cash equivalent loss [9]. Consequently, the uncertain parameter here is the n-vector of asset means $\tilde{\theta} = \tilde{\mu}$. By taking a margin error $\epsilon = 0.5\%$ and a confidence level $\sigma = 0.5\%$, the corresponding Chernoff bound, according to (6), is $1.198\,10^5$. For the entire experiment, we fix the sample size at $N = 1.5\ 10^5$, and the risk-aversion parameter at $\lambda = 2$. About the scenarios' random draw, we opt for a uniformly iid sampling from the set,

$$\left\{\mu \in \mathbb{R}^n : |\mu_i - \mu_{0\,i}| \leq \xi,\ \forall i \in [|1, n|]\right\} = B_\xi(\mu_0),$$

where μ_0 represents the nominal vector of asset means given by Table III. Two noise magnitudes ξ are considered, a low magnitude $\xi = 0.01$ and a large one $\xi = 0.3$. To investigate the impact of the initial choice of asset means[5], ten instances of μ are considered (from μ_0 to μ_9); nine are randomly sampled according to the same sampling process described above, plus the nominal value μ_0. For each case of ξ, 100 runs of the hybrid GA are performed, the average value are reported thereafter. For each run of the hybrid GA, the following solution set $\mathcal{X}$ is constituted,

$$\mathcal{X} = \{w_i^{\mu_j}, \forall i \in [|1, 100|], \forall j \in [|0, 9|]\},$$

with $w_i^{\mu_j}$ is the portfolio number $1 \leq i \leq K = 100$, generated using the asset mean μ_j, $0 \leq j \leq 9$. K refers to the GA population size. The set $\mathcal{X}$ is slightly changed in our program by removing identical portfolios $w_i^{\mu_j}$ that are found using different μ_j. Only the first occurrence of these duplicates are retained. It is attributed to the initial μ_j that finds them.

[5]The corresponding initial choice of Figure 2 is θ'_1, θ'_2 and θ'_3.

In our setting, we compare three populations induced from the final output, namely the first K portfolios sorted according to $R_1^{\mathcal{X}}$ and to $R_2^{\mathcal{X}}$ measures and to the worst-case scenario (wc), identified by expression (7). Following the results obtained for $\xi = 0.01$.

1) Low noise magnitude: Figure 3 compares the three populations (according to wc, $R_1^{\mathcal{X}}$ and $R_2^{\mathcal{X}}$) averaged over 100 runs of the hybrid GA. The μ_j in the figure's legend represents the instance of asset means used in the portfolios generation. In general, the three populations do not differ much. However, the ones according to the worst-case scenario and $R_1^{\mathcal{X}}$ are quite similar, expect very small differences between slots which does not exceed 0.4% (larger difference for μ_0). On the other hand, the $R_2^{\mathcal{X}}$ population, even similar, allows more space for the nominal portfolios (generated using μ_0), for almost 21.35%. Table IV gives an example of the best ten rankings across the populations, for one run among the 100 used runs. It shows that, in this particular case, portfolios generated using μ_1 are more adapted to the worst-case scenario, which corresponds to the scenarios number $38\,886$ of this instance run. Also, portfolios generated using μ_4 have better $R_2^{\mathcal{X}}$ measure, averagely performing better across scenarios, i.e. higher performance ratio. The first rankings by $R_1^{\mathcal{X}}$ are however more diverse, including portfolios yielded by μ_6, μ_5, μ_4 and μ_1 among others. In summary, while the 10^{th} first rankings of the populations can be completely different, the overall repartition according to wc scenario and $R_1^{\mathcal{X}}$ looks much alike.

	wc population		$R_1^{\mathcal{X}}$ population		$R_2^{\mathcal{X}}$ population	
ranking	portfolio	μ	portfolio	μ	portfolio	μ
1^{st}	$w_1^{\mu_1}$	μ_1	$w_1^{\mu_6}$	μ_6	$w_1^{\mu_4}$	μ_4
2^{nd}	$w_2^{\mu_1}$	μ_1	$w_1^{\mu_5}$	μ_5	$w_2^{\mu_4}$	μ_4
3^{rd}	$w_{15}^{\mu_1}$	μ_1	$w_1^{\mu_4}$	μ_4	$w_{10}^{\mu_4}$	μ_4
4^{th}	$w_{19}^{\mu_1}$	μ_1	$w_1^{\mu_1}$	μ_1	$w_{12}^{\mu_4}$	μ_4
5^{th}	$w_{20}^{\mu_1}$	μ_1	$w_1^{\mu_8}$	μ_8	$w_{13}^{\mu_4}$	μ_4
6^{th}	$w_{21}^{\mu_1}$	μ_1	$w_1^{\mu_0}$	μ_0	$w_{14}^{\mu_4}$	μ_4
7^{th}	$w_{55}^{\mu_1}$	μ_1	$w_1^{\mu_7}$	μ_7	$w_{15}^{\mu_4}$	μ_4
8^{th}	$w_{58}^{\mu_1}$	μ_1	$w_1^{\mu_9}$	μ_9	$w_{18}^{\mu_4}$	μ_4
9^{th}	$w_{59}^{\mu_1}$	μ_1	$w_{95}^{\mu_9}$	μ_9	$w_{19}^{\mu_4}$	μ_4
10^{th}	$w_{78}^{\mu_8}$	μ_8	$w_1^{\mu_3}$	μ_3	$w_{21}^{\mu_4}$	μ_4

Table IV. An instance of best portfolios by population for $\xi = 0.01$

Figure 4 shows the distribution of rankings across 100 runs of the hybrid MH. For each top ranked portfolio, which has a rank 1, of each population (according to wc, $R_1^{\mathcal{X}}$ or $R_2^{\mathcal{X}}$), we examine its analogous ranking in the other wc, $R_1^{\mathcal{X}}$ and $R_2^{\mathcal{X}}$ populations. This is done 100 times (100 runs of hybrid MH) in order to construct the box plots of Figure 4. Rankings

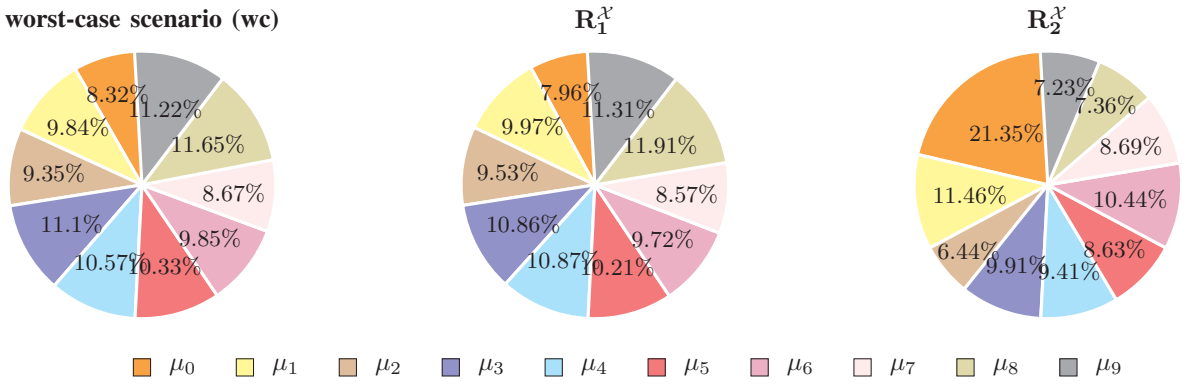


Figure 3. Repartition of final populations, according to wc, $R_1^{\mathcal{X}}$ and $R_2^{\mathcal{X}}$, in terms of asset means instances μ_i (for $\xi = 0.01$)

below the red line designate portfolios that can be part of the final population, i.e. having a ranking less or equal than $K = 100$. From the figure, we observe that $R_1^{\mathcal{X}}$ measure assigns high rankings to the top portfolios according to wc (the first box plot from the left), which suggests a similarity between rankings in wc and $R_1^{\mathcal{X}}$ populations. However, this is not reciprocal, the best portfolios according to $R_1^{\mathcal{X}}$ have spread ranks in wc population (the third box plot). We can similarly notice that $R_1^{\mathcal{X}}$ measure also allocates high ranks to the best $R_2^{\mathcal{X}}$ individuals (the sixth box plot). First ranked portfolios by $R_1^{\mathcal{X}}$ cover large area of rankings in populations of both measures $R_2^{\mathcal{X}}$ and wc, according to the two blue box plots of Figure 4. All the box plot means are above the red line, except the first and sixth ones.

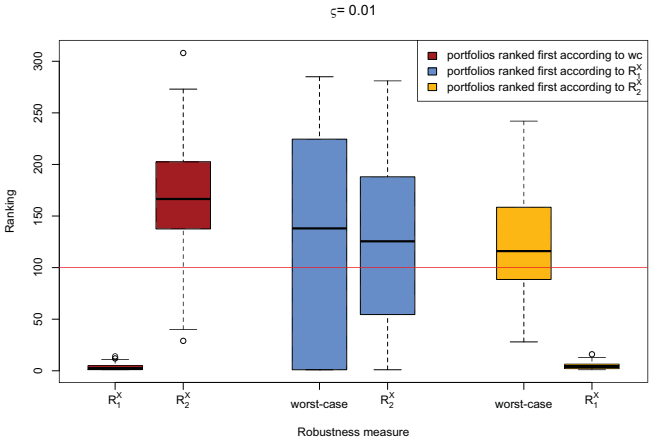


Figure 4. Distribution of the analogous rankings of the best portfolios according to wc, $R_1^{\mathcal{X}}$ and $R_2^{\mathcal{X}}$ (for $\xi = 0.01$)

2) Large noise magnitude: The second part of the experimentation concerns the magnitude of $\xi = 0.3$. Figures 5 and 6 are respectively the equivalent results of Figures 3 and 4. Same observations concerning the rankings distributions of law magnitude can be made here also. The overall populations' repartitions follow likewise, although the populations according to $R_2^{\mathcal{X}}$ gives more room to the nominal portfolios, for around 28.31%. Notice that the nominal asset means μ_0 based portfolios take, according to Figure 5, negligible proportions in wc and $R_1^{\mathcal{X}}$ populations, for around 4.1%. This is consistent with the high risk of having unlimited belief in nominal values of uncertain parameters.

The implementation of the overall hybrid MH, GA and simulation module, is made in the Java environment[6]. The various tests are done on a machine with an Intel Core i7 CPU at 3GHz and 8GB DDR3 RAM, using JDK 1.7.0.

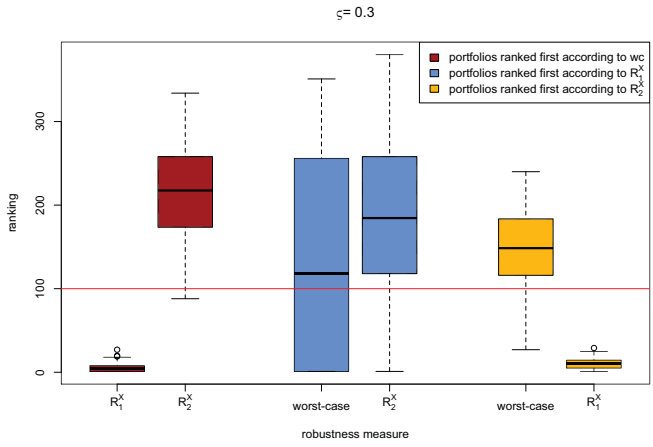


Figure 6. Distribution of the analogous rankings of the best portfolios according to wc, $R_1^{\mathcal{X}}$ and $R_2^{\mathcal{X}}$ (for $\xi = 0.3$)

V. Conclusion

In order to capture the randomness of input parameters for ill-defined problems, we proposed a hybrid metaheuristic (MH) approach incorporating a simulation module. This

[6]The code is available upon request by email.

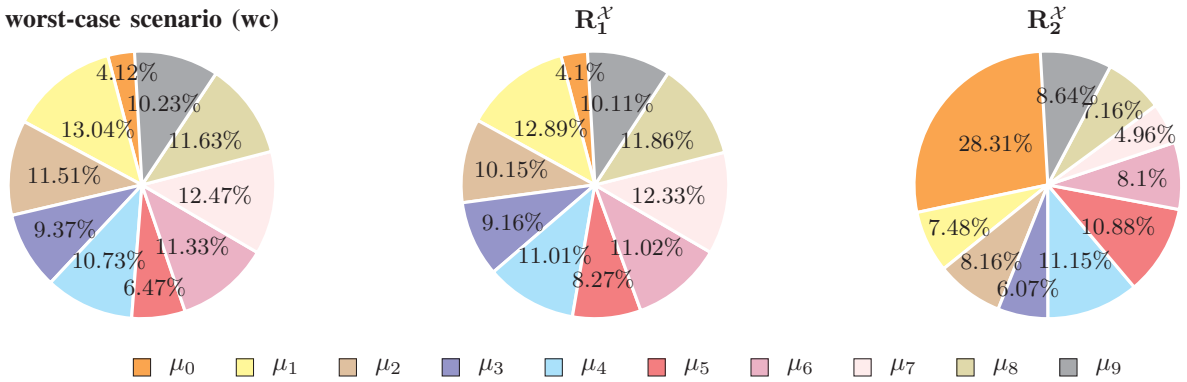


Figure 5. Repartition of final populations, according to wc, $R_1^{\mathcal{X}}$ and $R_2^{\mathcal{X}}$, in terms of asset means instances (for $\xi = 0.3$)

module is created for the purpose of finding less sensitive solutions to perturbations affecting some uncertain input parameters. Two measures of robustness are used within the simulation module :

1) ($\mathbf{R_1^{\mathcal{X}}}$) the percentage of being top-ranked solution throughout randomly sampled scenarios of the uncertain parameters,
2) ($\mathbf{R_2^{\mathcal{X}}}$) the average across scenarios of the solution performance ratio, which corresponds to the solution evaluation over top-ranked evaluation for the related scenario.

Empirical experimentations are conducted for the problem of portfolio optimization against noises in the expected returns of assets, which are inputs of the model. The emphasis of the application is on comparing ranks of the final solutions induced by three criteria: both robustness measures ($R_1^{\mathcal{X}}$ and $R_2^{\mathcal{X}}$) plus ranks implied by the worst-case scenario (wc).

Results have shown, for both cases of high and low magnitude perturbations, that the final population to the $R_1^{\mathcal{X}}$ measure is highly similar to that suggested by the **wc** scenario, meanwhile $R_1^{\mathcal{X}}$ population performs better when it comes to ranking the top-ranked portfolios according to the other measures $R_2^{\mathcal{X}}$ and wc.

In future work, we intend to extend the hybrid approach by embedding in it a selection mechanism of the sampled scenarios. Instead of treating scenarios equally, the user can have more control over the process.

REFERENCES

[1] A. Eiben and J. Smith, *Introduction to evolutionary computing*. springer, 2008.

[2] Z. Michalewicz, *Genetic algorithms+ data structures*. Springer, 1996.

[3] K. Sridharan, "A gentle introduction to concentration inequalities," 2009, available: http://ttic.uchicago.edu/ karthik/ concentration.pdf.

[4] G. R. Raidl, "A unified view on hybrid metaheuristics," in *Hybrid Metaheuristics*. Springer, 2006, pp. 1–12.

[5] H. Markowitz, "Portfolio selection," *The journal of finance*, vol. 7, no. 1, pp. 77–91, 1952.

[6] J. D. Jobson and B. Korkie, "Estimation for markowitz efficient portfolios," *Journal of the American Statistical Association*, vol. 75, no. 371, pp. 544–554, 1980.

[7] R. Michaud, "The markowitz optimization enigma: is 'optimized' optimal?" *Financial Analysts Journal*, pp. 31–42, 1989.

[8] M. Best and R. Grauer, "On the sensitivity of mean-variance-efficient portfolios to changes in asset means: some analytical and computational results," *Review of Financial Studies*, vol. 4, no. 2, pp. 315–342, 1991.

[9] V. Chopra and W. Ziemba, "The effect of errors in means, variances, and covariances on optimal portfolio choice," *The journal of portfolio management*, vol. 19, no. 2, pp. 6–11, 1993.

[10] M. Broadie, "Computing efficient frontiers using estimated parameters," *Annals of Operations Research*, vol. 45, no. 1, pp. 21–58, 1993.

[11] T. Chang, N. Meade, J. Beasley, and Y. Sharaiha, "Heuristics for cardinality constrained portfolio optimisation," *Computers and Operations Research*, vol. 27, no. 13, pp. 1271–1302, 2000.

[12] M. Castillo Tapia and C. Coello, "Applications of multi-objective evolutionary algorithms in economics and finance: A survey," in *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*. IEEE, 2007, pp. 532–539.

[13] F. Schlottmann and D. Seese, "Financial applications of multi-objective evolutionary algorithms: Recent developments and future research directions,"," *Applications of Multi-Objective Evolutionary Algorithms*, pp. 627–652, 2004.

[14] O. Rifki and H. Ono, "A survey of computational approaches to portfolio optimization by genetic algorithms," 2012, available: http://hdl.handle.net/ 2324/ 25317.

[15] R. B. Agrawal, K. Deb, and R. B. Agrawal, "Simulated binary crossover for continuous search space," 1994.

[16] K. Deb *et al.*, *Multi-objective optimization using evolutionary algorithms*. John Wiley & Sons Chichester, 2001, vol. 2012.

[17] R. O. Michaud and R. O. Michaud, "Efficient asset management: a practical guide to stock portfolio optimization and asset allocation," *OUP Catalogue*, 2008.

A Fixed-Parameter Algorithm for Max Edge Domination

Tesshu Hanaka and Hirotaka Ono

Department of Economic Engineering, Kyushu University,
Fukuoka 812-8581, Japan

Abstract. In a graph, an edge is said to *dominate* itself and its adjacent edges. Given an undirected and edge-weighted graph $G = (V, E)$ and an integer k, *Max Edge Domination* problem (MaxED) is to find a subset $K \subseteq E$ with cardinality at most k such that total weight of edges dominated by K is maximized. MaxED is NP-hard due to the NP-hardness of the minimum edge dominating set problem. In this paper, we present fixed-parameter algorithms for MaxED with respect to treewidth ω. We first present an $O(\omega^2 \cdot 2^{6\omega+3\omega^2} \cdot k^3 \cdot n)$-time algorithm. We then improve the running time into $O(\omega \cdot 2^{6\omega} \cdot k^3 \cdot n)$, whose exponent is a linear of ω. This improvement enables us to design a subexponential fixed-parameter algorithm of MaxED for apex-minor-free graphs, which is a graph class that includes planar graphs.

Keywords: max edge domination, fixed-parameter algorithm, bounded treewidth, subexponential FPT

1 Introduction

Let $G = (V(G), E(G))$ be an undirected and positive edge-weighted graph, where $V(G)$ is the set of n vertices and $E(G)$ is the set of m edges. These $V(G)$ and $E(G)$ are simply denoted by V and E, respectively. For an edge $e = \{u, v\} \in E$, its weight is denoted by w_e or w_{uv}. For $E' \subseteq E$, we denote by $V(E')$ the set of vertices that appear in E', that is, $V(E') = \bigcup_{e \in E'} e$. An edge is said to *dominate* itself and its all adjacent edges. We denote by $D_G(e)$ the set of edges dominated by an edge e, that is, $D_G(e) = \{e' \in E(G) \mid e' \cap e \neq \emptyset\}$. For a set E' of edges, we denote by $D_G(E')$ the set of edges dominated by an edge in E', that is, $D_G(E) = \{e \in E(G) \mid e \cap V(E') \neq \emptyset\}$. In these notations, we may omit the subscript G if it is clear.

Given $G = (V, E)$ and an integer k, *Max Edge Domination* problem (MaxED) is to find a subset $K \subseteq E$ with cardinality at most k such that total weight of edges dominated by K is maximized. This problem is formulated by the following optimization problem:

$$\max_{K \subseteq E, |K| \leq k} \sum_{e \in D(K)} w_e.$$

In a sense of the decision problem, MaxED for an unweighted graph is equivalent to the well-known *Minimum Edge Dominating Set* (EDS), that is, the problem to find a minimum subset of E' dominating all edges in E. Due to the NP-hardness of EDS, MaxED is NP-hard, and several approximability (or inapproximability) results are known. For example, MaxED is APX-hard [20], and a greedy algorithm achieves approximation ratio $\max\{1-1/e, k/s\}$, where s is the size of maximal matching [16].

In this paper, we consider fixed-parameter tractability of MaxED. Given a problem with input size n and a parameter γ, the problem is said to be *fixed-parameter tractable* (FPT, for short) if it can be solved in $f(\gamma) \cdot n^{O(1)}$ time, where f is a certain function that depends only on parameter γ. An algorithm that achieves the above running time is called a fixed-parameter algorithm. Particularly, if $f(\gamma) = 2^{o(\gamma)}$, the problem is called *subexponential fixed-parameter tractable*. For general concepts of fixed parameter tractability and related topics, see [8, 11, 21]. It is known that EDS is FPT with respect to the solution size [9], but this does not imply the fixed parameter tractability of MaxED with respect to k, because the solution size of EDS can be much larger than k in general. In fact, MaxED with parameter k has shown to be $W[1]$-hard even for unweighted bipartite graphs [15]. This implies that there unlikely exists a fixed-parameter algorithm for MaxED with parameter k.

In this paper, we show that (1) MaxED with respect to treewidth ω is FPT, and (2) MaxED with respect to k is subexponential FPT for apex-minor-free graphs, which is a graph class that includes planar graphs. For the former result, we first present a basic $O(\omega^2 \cdot 2^{6\omega+3\omega^2} \cdot k^3 \cdot n)$-time algorithm for MaxED, and then improve the running time into $O(\omega \cdot 2^{6\omega} \cdot k^3 \cdot n)$. The fixed-parameter tractability of MaxED with respect to treewidth is rather straightforward, but the improved running time plays a key role of the latter result.

There are many combinatorial optimization problems that have subexponential fixed-parameter algorithms for superclasses of planar graphs. A powerful meta-theorem to design a subexponential fixed-parameter algorithm is known for problems having bidimensionality ([4, Theorem 8.1]). Roughly speaking, if a problem has bidimensionality, the treewidth of a planar graph (or a graph in some superclasses of planar graphs) is bounded by $O(\sqrt{k^*})$, where k^* is the optimal value of the problem. By combining this with $2^{O(\omega)}n^{O(1)}$-time algorithm, a subexponential fixed-parameter algorithm can be obtained. Although EDS with respect to solution size is an example of problems having bidimensionality, MaxED with respect to k is unfortunately not. Instead, we try to choose a special K^* among all the optimal solutions. In this strategy, K^* and its neighbors are localized so that the treewidth of the subgraph of G induced by K^* and its neighbors is bounded by $O(\sqrt{k})$. Then, we can expect a similar speeding-up effect. The points become (i) how we localize K^*, and (ii) the design of a fixed-parameter algorithm whose exponent is linear of ω. This scheme is proposed by [13] to design a subexponential fixed-parameter algorithm of *Partial Vertex Cover* with respect to k, which is also not a bidimensional problem, subexponential fixed-parameter algorithms with respect to k for the partial dominating

set and the partial vertex cover of apex-minor-free graphs. Another example of employing this scheme is found in [18]. To apply the scheme, we utilize a generalized version of EDS, say r-EDS, and investigate the approximability. Based on these together with the faster algorithm mentioned in the previous paragraph, we show that there is an algorithm solving MaxED for apex-minor-free graphs in $2^{O(\sqrt{k})} \cdot n^{O(1)}$ time.

1.1 Related work

As mentioned above, MaxED is strongly related to EDS. EDS is the problem of finding a minimum subset $S \subseteq E$ such that all edges $e \in E \setminus S$ are adjacent to at least one edge in S. EDS is also known as *Minimum Maximal Matching*. There are many studies for EDS from the viewpoint of (in)approximability, parameterized complexity and exact algorithms. For example, EDS is 2-approximable in polynomial time [14], NP-hard to approximate within any factor better than 7/6 [3], and can be exactly solved in $O^*(1.3160^n)$ time, where O^*-notation suppresses all polynomially bounded factors [23]. EDS is also known to be fixed-parameter tractable with respect to several parameters, e.g., the solution size of EDS, treewidth, and so on. For example, an $O^*(1.821^\tau)$-time algorithm of EDS [22] and an $O^*(2.1479^{k^*})$-time algorithm of EDS for cubic graphs are proposed, where τ is the solution size of the minimum vertex cover, and k^* is the solution size of EDS.

As mentioned before, EDS with solution size is known to be a bidimensional problem. By using the bidimensonality theory, a subexponential fixed-parameter algorithm for apex-minor-free graphs can be designed [5].

Compared with EDS, MaxED itself is less studied. MaxED is a special case of *Maximum Coverage Problem* (MaxC): Given n elements x_i with positive weight w_i, $i = 1, 2, \ldots, n$, sets of $S_1, S_2, \ldots, S_m \subseteq \{x_1, x_2, \ldots, x_n\}$ and a positive integer k, find a set $C \subseteq \{1, 2, \ldots, m\}$ such that $|C| \leq k$ and $\sum_{x_i \in \bigcup_{j \in C} S_j} w_i$ is maximized. Since MaxC is known to be $(1 - 1/e)$-approximable in polynomial time [7, 17], so is MaxED. Though the approximation ratio is tight for MaxC under P$\neq$NP ([10]), MaxED is just known to be APX-hard [20]. As for the parameterized complexity, MaxED with respect to k has been recently shown to be W[1]-hard even for unweighted bipartite graphs [15].

This paper is organized as follows. In Section 2, we introduce notations and definitions. In Sections 3 and 4, we present two fixed-parameter algorithms for MaxED. We first present a basic algorithm in Section 3, and then improve the running time in Section 4. Finally, we show that a $2^{O(\sqrt{k})} \cdot n^{O(1)}$-time algorithm of MaxED for apex-minor-free graphs in Section 5.

2 Preliminaries

Let $G = (V, E)$ be an undirected and edge-weighted graph. For $V' \subseteq V$, let $G[V']$ denote a subgraph of G induced by V'. For $E' \subseteq E$, we simply denote $G[V(E')]$ by $G[E']$.

Here, we introduce the following Lemma 1, whose proof is shown in Appendix.

Lemma 1. There is an optimal solution K^* for MaxED, where K^* forms a matching, that is, every two $e, e' \in K^*$ satisfy $e \cap e' = \emptyset$.

By Lemma 1, we assume that our algorithms for MaxED finds an optimal solution that forms a matching.

2.1 Tree Decomposition

Our algorithms that will be presented in Sections 3 and 4 are based on dynamic programming on tree decomposition. In this subsection, we give the definition of tree decomposition.

A *tree decomposition* of a graph $G = (V, E)$ is defined as a pair $\langle \mathcal{X}, T \rangle$, where $\mathcal{X} = \{X_1, X_2, \ldots, X_N \subseteq V\}$, and T is a tree whose nodes are labeled by $1, 2, \ldots, N$, such that

1. $\bigcup_{i \in I} X_i = V$.
2. For $\forall \{u, v\} \in E$, there exists X_i such that $\{u, v\} \subseteq X_i$.
3. For all $i, j, k \in \{1, 2, \ldots, N\}$, if j lies on the path from i to k in T, then $X_i \cap X_k \subseteq X_j$.

In the following, we call T a decomposition tree, and we use term "nodes" (not "vertices") for T to avoid a confusion. The width of a tree decomposition $\langle \mathcal{X}, T \rangle$ is is defined by $\max_{i \in \{1,2,\ldots,N\}} |X_i| - 1$, and the treewidth of G, denoted by $\mathbf{tw}(G)$, is the minimum width over all tree decompositions of G. We sometimes use ω to represent $\mathbf{tw}(G)$.

In general, computing $\mathbf{tw}(G)$ of a given G is NP-hard [1], but fixed-parameter tractable with respect to the treewidth [2]. In the following, we assume that a decomposition tree with the minimum treewidth is given.

2.2 r-Edge Dominating Set

We define a new problem by extending the notion of domination. We first define *distance* between two edges $e_1 = \{u_1, v_1\}$ and $e_2 = \{u_2, v_2\}$ as the shortest path length among (u_1, u_2)-path, (u_1, v_2)-path, (v_1, u_2)-path and (v_1, v_2)-path, which we denote by $d(e_1, e_2)$. *r-Edge Dominating Set* (r-EDS) is the problem of finding an edge set $S \subseteq E$ with minimum size such that for every $e \in E \setminus S$, $d(e, e') < r$ holds for some edge $e' \in S$. This problem is clearly a generalization of EDS, because 1-EDS is equivalent to EDS. To design a subexponential fixed-parameter algorithm in Section 5, we design a constant-factor approximation algorithm for 2-EDS.

3 Basic Algorithm Based On Dynamic Programming

In this section, we present a dynamic programming (DP) algorithm based on a decomposition tree. By the assumption above, we are already given a decomposition tree with treewidth of ω. We assume that a decomposition tree T is a rooted binary tree without loss of generality. The algorithm first prepares $k+1$ DP tables for each X_i, so $(k+1) \cdot N$ tables in total. The algorithm runs in the bottom-up manner; it fills tables from leaf nodes to the root node. For simplicity, we assume that the indices of X_i correspond to the order that the algorithm visits X_i; the algorithm fills tables of $X_1, X_2, \ldots, X_N$ in this order.

We further give several assumptions for the tree decomposition. We define a mapping g from E to X_i. For an edge $e \in E$, there exists at least one bag X_i such that $e \subseteq X_i$ by the definition of tree decomposition. If $|\{X_i \in \mathcal{X} \mid X_i \cap e \neq \emptyset\}| = 1$, we define $g(e) = X_i$ such that $X_i \cap e \neq \emptyset$. If $|\{X_i \in \mathcal{X} \mid X_i \cap e \neq \emptyset\}| > 1$, we define $g(e) = X_i$ where i is the smallest index such that $X_i \cap e \neq \emptyset$. By defining g, we make clear in which node we handle e. Based on g, we partition E into $E_1, E_2, \ldots, E_N$, where $E_i = \{e \in E \mid g(e) = X_i\}$. We then define a subgraph $G_i = (V_i, E_i)$ of G, where $V_i = X_i$.

3.1 DP table and its computation

In the basic algorithm, we prepare DP table $A_i^{(r)}$, $r = 0, 1, 2, \cdots, k$, for a bag X_i. Here, r represents the number of edges selected as a part of a solution at the moment. See Table 1 as an example of $A_i^{(r)}$. In the table, $|V_i| = n_i$ and $|E_i| = m_i$. Table $A_i^{(r)}$ consists of $n_i + m_i + 1$ columns and $4^{n_i} \cdot 2^{m_i}$ rows. The first $n_i + m_i$ columns represent the statuses of vertices and edges in G_i. The last column represents the evaluation value ξ of the corresponding statuses. We call the $n_i + m_i$ statuses of a row a *status vector*.

Table 1. $A_i^{(r)}$

v_{i_1}	v_{i_2}	$\cdots$	$v_{i_{n_i}}$	e_{i_1}	e_{i_2}	$\cdots$	$e_{i_{m_i}}$	ξ
0	0	$\cdots$	0	**N**	**N**	$\cdots$	**N**	10
1	0	$\cdots$	0	**N**	**N**	$\cdots$	**N**	11
$\vdots$	$\vdots$		$\vdots$	$\vdots$	$\vdots$			$\vdots$
1	1	$\cdots$	1	**Y**	**Y**	$\cdots$	**Y**	25
2	0	$\cdots$	0	**N**	**N**	$\cdots$	**N**	13
$\vdots$	$\vdots$		$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$
2	2	$\cdots$	2	**Y**	**Y**	$\cdots$	**Y**	25
3	0	$\cdots$	0	**N**	**N**	$\cdots$	**N**	13
$\vdots$	$\vdots$		$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$
3	3	$\cdots$	3	**Y**	**Y**	$\cdots$	**Y**	25

Now we explain the statuses of vertices and edges. As a status of vertex v_j, we define four statuses **FUT**, **CUR**, **PAST** and **NULL**. Status **FUT** represents

that v_j does not touch any edge that is or was selected as a part of solution, but touches an edge that will be selected as a part of solution. A vertex of status **FUT** should appear in a parent node. Note that the root node cannot take status **FUT**. Status **CUR** represents that v_j touches an edge in G_i that is selected as a part of solution. Status **PAST** represents that v_j touches an edge that was selected as apart of solution in its offspring node. The remaining vertices take status **NULL**. In a DP table $A_i^{(r)}$, we use $3, 2, 1, 0$ instead of **FUT**, **CUR**, **PAST** and **NULL**, respectively.

For edge statuses, we define two statuses **Y** and **N**, where **Y** represents that the edge is selected as a part of solution and **N** represents that the edge is not selected as a part of solution. Recall that each edge appears in exactly one G_i. Therefore, the vertex status **CUR** and the edge status **Y** are correspond.

Let j_L and j_R be two children nodes of i on T. We define the evaluation value of status vector $\mathbf{p}$ as follows:

$$\xi(A_i^{(r)}(\mathbf{p})) = \sum_{\substack{u\in\{v_i|v_i\neq 0\}\\ \{u,v\}\in E_i}} w_{uv} + \max_{s+t+|K_i(\mathbf{p})|=r} \{\xi(A_{j_L}^{(s)}(\mathbf{p}^L)) + \xi(A_{j_R}^{(t)}(\mathbf{p}^R))\}, \quad (1)$$

where $\mathbf{p}^L$ (resp., $\mathbf{p}^R$) is a status vector of $A_{j_L}^{(s)}$ (resp., $A_{j_R}^{(s)}$) that is consistent to $\mathbf{p}$, and $K_i(\mathbf{p})$ is the set of edges in E_i whose status is **Y**. In (1), s and t are the numbers of edges selected as a part of solutions in the offsprings of X_{j_R} and X_{j_L} respectively. The first term represents the edge weight dominated in X_i under $\mathbf{p}$. The second term represents the maximum sum of the evaluation values derived from two children nodes. After filling up the tables of root node, we can obtain an optimal value, and also can obtain an optimal solution by reversely traversing the tables.

Probably a bit tricky part of the algorithm is to define the status **FUT**. This status is introduced to avoid overlooking of the weight of edges that appear in an offspring node. This should be considered because we use the partition $(E_1, E_2, \ldots, E_N)$ of E.

Another important notion is *consistency* of $\mathbf{p}, \mathbf{p}^L$ and $\mathbf{p}^R$. We call $\mathbf{p}$ is inconsistent if it contains impossible statuses. For example, for $e = \{u, v\} \in E_i$, $\mathbf{p}$ is inconsistent if both of the statuses of u and v are **NULL** and $e \in K_i(\mathbf{p})$. For $\mathbf{p}^L$ and $\mathbf{p}^R$, we need to pay an attention for $v \in V_{J_L} \cap V_{j_R}$. Let α_L and α_R be the statuses of v in $\mathbf{p}^L$ and $\mathbf{p}^R$, respectively. Then, the following gives all possible combinations:

$$(\alpha_R, \alpha_L) \in \{\ (\mathbf{NULL}, \mathbf{NULL}), (\mathbf{FUT}, \mathbf{FUT}), (\mathbf{FUT}, \mathbf{CUR}), (\mathbf{FUT}, \mathbf{PAST}),\\ (\mathbf{CUR}, \mathbf{FUT}), (\mathbf{PAST}, \mathbf{FUT})\}.$$

The cases $(\alpha_R, \alpha_L) = (\mathbf{CUR}, \mathbf{CUR}), (\mathbf{CUR}, \mathbf{PAST}), (\mathbf{PAST}, \mathbf{CUR}), (\mathbf{PAST}, \mathbf{PAST})$ can be excluded, because in these combinations solutions do not form a matching. Moreover, we have to consider *consistency* of the parent node and the children nodes. For example, if v's statuses in j_L and j_R are respectively **FUT** and **CUR**, v's status of i should be **PAST**, because v is selected at j_R (i.e., it is a past of i). Let α be the status of $\mathbf{p}$ (i.e., at node i). For each (α_R, α_L) above,

we see possible statuses of α by thinking its chronological order. We then have the following:

- For $(\alpha_R, \alpha_L) = (\mathbf{NULL}, \mathbf{NULL})$, α is **NULL**.
- For $(\alpha_R, \alpha_L) = (\mathbf{FUT}, \mathbf{FUT})$, α is **FUT** or **CUR**.
- For $(\alpha_R, \alpha_L) = (\mathbf{FUT}, \mathbf{CUR})$ or $(\mathbf{CUR}, \mathbf{FUT})$, α is **PAST**.
- For $(\alpha_R, \alpha_L) = (\mathbf{FUT}, \mathbf{PAST})$ or $(\mathbf{PAST}, \mathbf{FUT})$, α is **PAST**.

Overall, we present the following algorithm.

Basic Algorithm
Step 0. Let $i := 1$.
Step 1. Initialize $A_i^{(r)}$ for $r = 0, \ldots, k$.
- Delete rows such that vertex statuses are not consistent to edge statuses.
- Delete rows such that there is a vertex where $\alpha = \mathbf{FUT}$ in $A_i^{(r)}$, while there is not in the parent node of X_i.

Step 2. For every $\mathbf{p}$, compute (1)
Step 3. If $i = N$, output $\max_{r,\mathbf{p}} \xi(A_N^{(r)}(\mathbf{p}))$. Otherwise, let $i := i+1$ and return to Step 1.

We consider the running time of the basic algorithm. The running time of Step 1 is $O(4^\omega \cdot 2^{\omega^2} \cdot k \cdot \omega^2)$ time, because $n_i = O(\omega)$, $m_i = O(\omega^2)$ and the number of rows is $O(4^\omega \cdot 2^{\omega^2} \cdot (k+1))$.

In Step 2, we check all the combinations $A_{j_L}^{(s)}$, $A_{j_R}^{(t)}$ and $A_i^{(r)}$ for $s = 0, 1, \ldots, k$, $t = 0, 1, \ldots, k$, and $r = 0, 1, \ldots, k$. This takes $O(4^{3\omega} \cdot 2^{3\omega^2} \cdot k^3 \cdot \omega^2)$-time in total. Since the procedure of Steps 2 , 3 and 4 is repeated N times, total running time of this algorithm is $O(4^{3\omega} \cdot 2^{3\omega^2} \cdot k^3 \cdot \omega^2 \cdot n)$ time. As a result, we obtain the following theorem.

Theorem 1. *There is an $O(\omega^2 \cdot 2^{6\omega+3\omega^2} \cdot k^3 \cdot n)$-time algorithm for* MaxED.

4 Improvement of the Running time

4.1 Improved Algorithm

We improve the basic algorithm. For the improved algorithm, we prepare a DP table as a new table instead of Table 1. A new DP table is obtained from the corresponding old one by deleting the edge statuses. The size of this DP table is 4^{n_i}. Notice that the size of an improved DP table is much smaller than a basic one. Due to the lack of information, it might be difficult to keep an explicit solution, but it does not matter because we can still compute the evaluation value. In this setting, what we need to consider is to check the consistency. In the basic algorithm, we can easily check the consistency of $\mathbf{p}$ by referring the status vector, but it is not possible for this case.

Actually, this case is also possible to check the consistency. In a new $\mathbf{p}$, all the vertices whose statuses are **CUR**, say $V_i(\mathbf{p})$, should touch an edge selected as a

part of solution in G_i. Since we assume that the selected edges form a matching by Lemma 1, each vertex in $V_i(\mathbf{p})$ touches exactly one edge selected as apart of solution in G_i. In other word, the selected edges form a perfect matching of $G_i[V_i(\mathbf{p})]$. This is clearly a necessary and sufficient condition of the consistency of $\mathbf{p}$, and it can be checked in polynomial time since the maximum matching problem can be solved in polynomial time [19].

Based on the observation, we can rewrite the evaluation value as follows:

$$\xi(A_i^{(r)}(\mathbf{p})) = \sum_{\substack{u\in\{v_i|v_i\neq 0\}\\(u,v)\in E_i}} w_{uv} + \max_{s+t+\frac{|V_i(\mathbf{p})|}{2}=r} \{\xi(A_{j_L}^{(s)}(\mathbf{p}^L)) + \xi(A_{j_R}^{(t)}(\mathbf{p}^R))\},$$

where $\mathbf{p}^L$ (resp., $\mathbf{p}^R$) is a status vector of $A_{j_L}^{(s)}$ (resp., $A_{j_R}^{(s)}$) that is consistent to $\mathbf{p}$. Note that the scheme of the improved algorithm is same as the one of the basic algorithm. Only the difference is how we check the inconsistency of $\mathbf{p}$.

4.2 The Running time of Improved Algorithm

We explain the running time of the improved algorithm. Since the scheme of the basic and improved algorithms are the same, we can easily amount the running time. In Step 1, we should delete the rows whose status vectors are inconsistent. This can be done by applying a maximum matching algorithm, whose running time is $O(\omega^{2.5})$ [19]. Other parts are similarly analyzed, and the total running time of the improved algorithm is $O(4^{3\omega} \cdot k^3 \cdot \omega \cdot n)$-time.

Theorem 2. *There is an $O(\omega \cdot 2^{6\omega} \cdot k^3 \cdot n)$-time algorithm for* MaxED.

5 Subexponentioal fixed-parameter Algorithm

In this section, we will show the following theorem by presenting a subexponential fixed-parameter algorithm.

Theorem 3. *There exists a $2^{O(\sqrt{k})} \cdot n^{O(1)}$-time algorithm for* MaxED *on apex-minor free graphs.*

Let G be an apex-minor-free graph. If $\mathbf{tw}(G) = O(\sqrt{k})$ holds, then Theorem 2 proves Theorem 3. Otherwise we will remove a set I of *irrelevant* edges from G so that at least one optimal solution is a subset of $E \setminus I$ and optimal also for the problem in $G[E \setminus I]$, and we have $\mathbf{tw}(G[E \setminus I]) = O(\sqrt{k})$. Then, applying Theorem 2 to $G[E \setminus I]$, we obtain Theorem 3. To identify such a set I of irrelevant edges, we introduce the notion of *lexicographically smallest solution*. The ideas follow from the ones given by Fomin et al. [13] as mentioned in Introduction.

Definition 1. *Given an ordering $\sigma = e_1 e_2 \dots e_m$ of E and subsets X and Y of E, we say that X is lexicographically smaller than Y, denoted by $X \leq_\sigma Y$, if X is lexicographically smaller than Y, if $E_\sigma^i \cap X = E_\sigma^i \cap Y$ and $e_{i+1} \in X \setminus Y$ for some $i \in \{0, 1, \dots, m\}$, where $E_\sigma^i = \{e_1, e_2, \dots, e_i\}$ for $i \in \{1, 2, \dots, m\}$ and $E_\sigma^0 = \emptyset$. We call a set $K \subseteq E$ the lexicographically smallest (optimal) solution for* MaxED *if for any other solution K' for the* MaxED *we have that $K \leq_\sigma K'$.*

Let $\sigma = e_1 e_2 \dots e_m$ be an ordering of the edges according to the total weight of the edges dominated by an edge in non-increasing order. For $e \in E$, let $\mu(e) = \sum_{e' \in D(e)} w_{e'}$. In the ordering σ,

$$\mu(e_1) \geq \mu(e_2) \geq \cdots \geq \mu(e_{m-1}) \geq \mu(e_m),$$

holds. Throughout the section, we assume that E is ordered by σ, and may use E_σ instead of E to emphasize this. We also denote $\{e_1, e_2, \dots, e_i\}$ by E^i_σ. We will propose an algorithm that finds not an optimal solution but the lexicographically smallest optimal solution for MaxED, which can make it clear to define a set of irrelevant edges. To this end, we give the following three lemmas, though the proof of Lemma 4 is omitted.

Lemma 2. *Given a graph $G = (V, E_\sigma)$, let $K = \{u_{i_1}, u_{i_2}, \dots, u_{i_k}\}$be the lexicographically smallest solution for* MaxED*, where $u_{i_k} = e_j$ for some j. Then, K is a* 2-EDS *of size at most k for $G[E^j_\sigma]$.*

Proof. Show this by contradiction. Assume that a lexicographically smallest solution K of MaxED is not a 2-EDS for $G[E^j_\sigma]$. This implies that there exists an edge e_i $(1 \leq i \leq j)$ such that $D_2(e_i) \cap K = \emptyset$. Let $K' = K \setminus \{e_j\} \cup \{e_i\}$. Clearly, $|K'| = |K|$. Since any edge $e \in D(e_i)$ is not dominated by K, we have

$$\mu(K') \geq \mu(K) - \mu(e_j) + \mu(e_i) \geq \mu(K),$$

a contradiction. □

Lemma 3. *Let G be an apex-minor-free graph. If G has an r-*EDS *of size at most k, $\mathbf{tw}(G) = O(r\sqrt{k})$.*

Proof. If G has an r-EDS of size k, then it has $(2k, r)$-center. Therefore, according to Lemma 8 of [18], the treewidth of G is $O(r\sqrt{k})$. □

Lemma 4. *On apex-minor-free graphs, there exists an EPTAS for* r-EDS*.*

Now we are ready to give a subexponential fixed-parameter algorithm. First, we sort $e_1, e_2, \dots, e_m \in E_\sigma$ and scan it from e_m to e_1. We put a stick in the right of e_m and let $s := m$. In an intermediate stage, if $G[E^j_\sigma]$ does not have a 2-edge dominating set of size at most $(1+\epsilon)k$, let $s := j-1$, $N := N \cup \{e_j\}$, and then we move the stick to the left of e_j. Notice that the edges in the left of the stick belong to $E \setminus N$ and the edges in the right are in N. The contraposition of Lemma 2 denotes that the lexicographically smallest solution for MaxED K lies $E \setminus N$, that is, $K \subseteq E \setminus N$. If $G[E^j_\sigma]$ has a 2-edge dominating set of size at most $(1+\epsilon)k$, then we find a subgraph G' such that $\mathbf{tw}(G') = O(\sqrt{k})$ and there exists $K' \subseteq E(G')$ satisfying $\mu(K) = \mu(K')$ for an optimal solution K of G, where $|K'| \leq k$ and $|K| \leq k$.

Given the parameter $(G = (V, E_\sigma), k, \epsilon, \emptyset)$ where $\epsilon > 0$, the algorithm is described as follows.

Subexponential fixed-parameter Algorithm

Step 0. Let $p := m$

Step 1. While there does not exist 2-edge dominating set of size at most $(1+\epsilon)k$ for $G[E^p_\sigma]$, repeat $N := N \cup \{e_p\}$, $p := p - 1$.

Step 2. Let $I = \{e \mid e \in N, D(e) \subseteq N\}$ and $E' = E \setminus I$.

Step 3. Find a tree decomposition of $G' = G[E']$ using the constant factor approximation algorithm of Demaine et al. [6] for computing the treewidth of H-minor-free graph.

Step 4. Apply the algorithm of Theorem 2 to $G[E']$.

The correctness of the algorithm can be shown by following the proof of Theorem 1 of [13]. In Step 1, we identify an edge set N such that $N \cap K = \emptyset$. Let e_p be a maximum index edge in $E \setminus N$. We check whether $G[E^p_\sigma]$ has 2-edge dominating set of size at most $(1+\epsilon)k$ by Lemma 4. If $G[E^p_\sigma]$ does not have it, then $K = \{u_{i_1}, u_{i_2}, \ldots u_{i_k}\}$ where $u_{i_k} = e_p$ is not the lexicographically smallest solution for MaxED by Lemma 2. Therefore, $e_p \notin K$. We will show letter half is valid. Note that edges in N are not candidates. Thus, an edge $e \in N$ adjacent to only edges in N is not dominated by K, that is, the set I is a set of irrelevant edges. Therefore, we delete a set of such edges as I. Let $E' = E \setminus I$. There exists K of size at most k in G such that $\mu(K) = \max_{K \subseteq E, |K| \leq k} \mu(K)$ if and only if there exists $K' \subseteq E'$ in G' such that $|K'| \leq k$ and $\mu(K') = \max_{K' \subseteq E', |K| \leq k} \mu(K')$. Hence, we will find K' in G' where $|K'| \leq k$ by Theorem 2.

We analyze the running time of this algorithm. When the loop in Step 2. is broken out, $G[E \setminus N]$ has 2-edge dominating set of size at most $(1+\epsilon)k$. Let D_2 be 2-edge dominating set of size at most $(1+\epsilon)k$. Then, D_2 is 3-edge dominating set for $G[E']$ because all edges such that $e \in N \cap E'$ are adjacent to edges in $E \setminus N$. Therefore, $\mathbf{tw}(G') = O(3\sqrt{(1+\epsilon)k}) = O(\sqrt{k})$ is shown by Lemma 3. We use the constant factor approximation algorithm of Demaine et al. [6] to compute the treewidth of H-minor-free graph, then we find tree decomposition such that the size of treewidth is $O(\sqrt{k})$ for $G[E']$ in $n^{O(1)}$-time. Finally, we use the algorithm of Theorem 2 to find optimal solution for MaxED in $O(2^{6\omega} \cdot k^3 \cdot \omega \cdot n)$-time. Therefore, our algorithm achieves running time $2^{O(\sqrt{k})} \cdot n^{O(1)}$.

References

1. Arnborg, S., Corneil, D.G., Proskurowski, A.: Complexity of finding embeddings in a k-tree. SIAM Journal on Algebraic Discrete Methods 8(2), 277–284 (1987)
2. Bodlaender, H.L.: A linear-time algorithm for finding tree-decompositions of small treewidth. SIAM Journal on Computing 25(6), 1305–1317 (1996)
3. Chlebík, M., Chlebíková, J.: Approximation hardness of edge dominating set problems. Journal of Combinatorial Optimization 11(3), 279–290 (2006)
4. Demaine, E.D., Hajiaghayi, M.: The bidimensionality theory and its algorithmic applications. The Computer Journal 51(3), 292–302 (2008)
5. Demaine, E.D., Hajiaghayi, M.: Linearity of grid minors in treewidth with applications through bidimensionality. Combinatorica 28(1), 19–36 (2008)

6. Demaine, E.D., Hajiaghayi, M., Kawarabayashi, K.i.: Algorithmic graph minor theory: Decomposition, approximation, and coloring. In: Proceedings of 46th Annual IEEE Symposium on Foundations of Computer Science. pp. 637–646. IEEE (2005)
7. Dobson, G.: Worst-case analysis of greedy heuristics for integer programming with n onnegative data. Mathematics of Operations Research 7(4), 515–531 (1982)
8. Downey, R.G., Fellows, M.R.: Parameterized complexity, vol. 3. Springer-Heidelberg (1999)
9. Escoffier, B., Monnot, J., Paschos, V., Xiao, M.: New results on polynomial inapproximability and fixed parameter approximability of edge dominating set. In: Parameterized and Exact Computation, Lecture Notes in Computer Science, vol. 7535, pp. 25–36. Springer Berlin Heidelberg (2012)
10. Feige, U.: A threshold of ln n for approximating set cover. Journal of the ACM (JACM) 45(4), 634–652 (1998)
11. Flum, J., Grohe, M.: Parameterized complexity theory, vol. 3. Springer (2006)
12. Fomin, F.V., Lokshtanov, D., Raman, V., Saurabh, S.: Bidimensionality and eptas. In: Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 748–759. SIAM (2011)
13. Fomin, F.V., Lokshtanov, D., Raman, V., Saurabh, S.: Subexponential algorithms for partial cover problems. Information Processing Letters 111(16), 814–818 (2011)
14. Fujito, T., Nagamochi, H.: A 2-approximation algorithm for the minimum weight edge dominating set problem. Discrete Applied Mathematics 118(3), 199–207 (2002)
15. Guo, J., Shrestha, Y.: Parameterized complexity of edge interdiction problems. In: Computing and Combinatorics, Lecture Notes in Computer Science, vol. 8591, pp. 166–178. Springer International Publishing (2014)
16. Hanaka, T., Ono, H.: Approximation ratios of greedy algorithms for max edge domination. In: Proceedings of Hinokuni Information Symposium (in Japanese). Information Processing Society of Japan (2013)
17. Hochbaum, D.S.: Approximating covering and packing problems: set cover, vertex cover, independent set, and related problems. In: Approximation algorithms for NP-hard problems. pp. 94–143. PWS Publishing Co. (1996)
18. Ishii, T., Ono, H., Uno, Y.: Subexponential fixed-parameter algorithms for partial vector domination. In: Combinatorial Optimization, pp. 292–304. Lecture Notes in Computer Science, Springer International Publishing (2014)
19. Micali, S., Vazirani, V.V.: An $O(\sqrt{|V|}|E|)$ algoithm for finding maximum matching in general graphs. In: Proceedings of 21st Annual Symposium on Foundations of Computer Science. pp. 17–27. IEEE (1980)
20. Miyano, E., Ono, H.: Maximum domination problem. In: Proceedings of the Seventeenth Computing: The Australasian Theory Symposium-Volume 119. pp. 55–62. Australian Computer Society, Inc. (2011)
21. Niedermeier, R.: Invitation to Fixed-Parameter Algorithms. Oxford University Press (2006)
22. Xiao, M., Nagamochi, H.: Parameterized edge dominating set in cubic graphs. In: Frontiers in Algorithmics and Algorithmic Aspects in Information and Management, Lecture Notes in Computer Science, vol. 6681, pp. 100–112. Springer Berlin Heidelberg (2011)
23. Xiao, M., Nagamochi, H.: A refined exact algorithm for edge dominating set. In: Theory and Applications of Models of Computation, Lecture Notes in Computer Science, vol. 7287, pp. 360–372. Springer Berlin Heidelberg (2012)

A APPENDIX

This appendix provides the proofs of the results that have been omitted due to space reasons. They may be read to the discretion of the program committee.

A.1 Proof of Lemma 1

Proof. When $k \geq s$ where s is the size of a minimum maximal matching, a minimum maximal matching is an optimal solution for MaxED.

Thus we assume that $k < s$. Let the set K be an optimal solution for MaxED and we call an edge $e \in K$ optimal edge. Suppose that there is no optimal solution with a matching structure. Then, there exists at least one vertex incident to at least two one optimal edges. We call such a vertex *heavy* vertex and edges composing it *composing* edges. Suppose that *heavy* vertices touches two optimal edges. Notice that there is at least one edge not dominated by K due to $k < s$. Let an edge e^* be one of such uncovered edge and let a vertex v^* be *heavy* vertex closest to e^*. Then, we consider the shortest path from v^* to e^* through *composing* edges. Let the edge e' be a *composing* edge closer to e^* and let a vertex v' be another endpoint of e', that is, $e' = \{v^*, v'\}$. All edges incident to v' are not optimal edges because v^* is *heavy* vertex. closest to e^*. If there is at least one edge incident to v' not dominated by K except for e', we can replace optimal edge e' with it without changing the total weight dominated by K. Then, this optimal solution has a matching structure(*), thus this is contradiction. Otherwise, let $K' = K \setminus \{e'\} \cup \{e^*\}$. Then, K' dominates at least more e' than K. Therefore, this contradicts to the optimality of K.

In the case that more than three optimal edges compose a *heavy* vertex, K does not have a matching structure in (*). However, we can lead a contradiction by considering the shortest path again. □

A.2 Proof of Lemma 4

To show Lemma 4, we use the result of [12].

For a problem Π, let $\phi_\Pi(G, S)$ be the feasibility constraint returning **true** if S is feasible and **false** otherwise. Let $\kappa_\Pi(G, S)$ be objective function. In most case, $\kappa_\Pi(G, S)$ will return $|S|$. We will only consider problems such that every instance has at least one feasible solution. Let $\mathcal{U}$ be the set of all graphs. For a graph optimization problem Π, let $\pi : \mathcal{U} \to \mathbb{N}$ be the function returning the objective function value of optimal solution of Π on G. For a graph G and a partition of $V(G)$ into $L \biguplus S \biguplus R$ such that $N(L) \subseteq S$ and $N(R) \subseteq S$, we define $G_L(G_R)$ as a graph obtained from G by contracting every connected component of $G[R](G[L])$ into the minimum index vertex in S.

Definition 2 ([12]). *A contraction-bidimensional problem has the separation property if given any graph G and a partition of $L \biguplus S \biguplus R$ such that $N(L) \subseteq S$ and $N(R) \subseteq S$, and given an optimal solution OPT to G, $\pi(G_L) \leq \kappa_\Pi(G_L, OPT \setminus E(G[R])) + O(|S|)$ and $\pi(G_R) \leq \kappa_\Pi(G_R, OPT \setminus E(G[L])) + O(|S|)$.*

Definition 3 ([12]). *A graph optimization problem Π with objective function κ_Π is called* reducible *if there exists a* **MIN/MAX-CMSO** *problem Π' and a functionf $: \mathbb{N} \to \mathbb{N}$ such that*

- *there is a polynomial time algorithm that given G and $X \subseteq V(G)$ outputs G' such that $\pi'(G') = \pi(G) \pm O(|X|)$ and $\mathbf{tw}(G') \leq f(\mathbf{tw}(G \setminus X)$.*
- *there is a polynomial time algorithm that given G and $X \subseteq V(G)$, G' and a vertex (edge) set S' such that holds $P_{\Pi'}(G', S')$, outputs S such that $\phi_\Pi(G, S) = \mathbf{true}$ and $\kappa_\Pi(G, S) = |S'| \pm O(|X|)$*

Then, the following lemma has been shown.

Lemma 5 ([12]). *Let Π be a reducible contraction-bidimensional problem with the separation property and H be a (apex) graph. There is an EPTAS for Π on the class of graphs excluding H as a minor.*

Therefore, there exists EPTAS for r-EDS if it is reducible and has separation property.

Proof of Lemma 4. Let $V(G) = L \uplus S \uplus R$ such that $N(L) \subseteq S$ and $N(R) \subseteq S$. Let Z be an optimal solution of r-EDS for G and Z_L be an optimal solution of r-EDS for G_L. Moreover, in a graph G_L, let $K^*_L[S]$ be an edge set such that $|K^*_L[S]| \leq |S|$ and for arbitrary non-isolated vertices u and v in S there exists at least one edge such that $\{u, v\} \in K^*_L[S]$. Then, $(Z \setminus E(G[R])) \cup K^*_L[S]$ is r-EDS of G_L because the edges dominated by $e \in E(G[R])$ or $(u, v) \in E(G)$ where $u \in R$ and $v \in S$ are dominated by $K^*_L[S]$. Because Z_L is an optimal solution in G_L,

$$\begin{aligned} |Z_L| &\leq |(Z \setminus E(G[R])) \cup K^*[S]| \\ &\leq |(Z \setminus E(G[R]))| + |K^*[S]| \\ &\leq |(Z \setminus E(G[R]))| + |S|. \end{aligned}$$

Therefore, r-EDS has separation property.

Given a graph G and edge set Y, let X be a vertex set of endpoints of edges in Y. Let $G' = G \setminus X$ and $R = D_r(Y) \setminus Y$. Note that $\mathbf{tw}(G') = \mathbf{tw}(G \setminus X)$. The annotated problem Π' is to find the minimum sized edge set $S' \subseteq E(G')$ such that every edge $e \in E(G) \setminus (S \cup R)$ is at distance at most r from some edges in S. Here, for any r-edge dominating set in G of Π, $S \setminus Y$ is a feasible solution in G' of Π'. Conversely, for any r-edge dominating set in G' of Π', $S' \cup Y$ is a feasible solution in G of Π. Because $\pi'(G') \leq |S \setminus Y| \leq |S| - |Y|$ for any S,

$$\pi'(G') \leq \pi(G) - |Y| \leq \pi(G) - \frac{1}{2}|X|.$$

Also, given G, $Y \subseteq E(G)$ and a set $X \subseteq V(G)$ of endpoints of Y, G' and an edge set S', let $S = S' \cup Y$. Then, considering $|Y| \leq |X| \leq 2|Y|$, they holds:

- $\phi_\Pi(G, S) = \mathbf{true}$
- $\kappa_\Pi(G, S) = |S'| + |Y| = |S'| + O(|X|)$.

Therefore, r-EDS is reducible. □

分割されたナップサック制約付き最大被覆問題に対する近似アルゴリズム

山下智大 (Tomohiro Yamashita)* 小野廣隆 (Hirotaka Ono)†

∗: 九州大学経済学部経済工学科
(Department of Economic Engineering, Kyushu University)
†: 九州大学大学院経済学研究院経済工学部門
(Department of Economic Engineering, Kyushu Universuty)

概要

分割されたナップサック制約付き最大被覆問題とは, 与えられた集合 $I = \{1, \ldots, n\}$ の部分集合族 $\mathcal{F} = \{S_j : j \in J\}$ と, 非負の重み w_j, 費用 c_j, 正の整数 $B_i (i = 1, \ldots, l)$, 集合 I の分割 $T_i (i = 1, \ldots, l)$ に対して, 費用 c_j に対する l 本のナップサック制約 $\sum_{j \in T_i} c_j \leq B_i (i = 1, \ldots, l)$ を満たし, X と非空の交わりをもつ S_j の重みの総和を最大にする I の部分集合 X を求めるという問題である. ナップサック制約付き最大被覆問題は NP 困難であるが, pipage rounding を用いたアルゴリズムで性能保証 $\left(1 - (1 - \frac{1}{k})^k\right) (k = \max\{|S_j| : j \in J\})$ を持つことが知られている. 本論文では, ナップサック制約を分割した場合も, pipage rounding を用いたアルゴリズムで同様の性能保証を持つことを示す.

Abstract

Given a family $\mathcal{F} = \{S_j : j \in J\}$ of subsets of a set $I = \{1, \ldots, n\}$ with associated nonnegative weights w_j and costs c_j and positive integers $B_i (i = 1, \ldots, l)$, and partitioned index sets $T_i (i = 1, \ldots, l)$ of a set I, the maximum coverage problem with partitioned knapsack constrains is to find a subset $X \subseteq I$ with $\sum_{j \in T_i} c_j \leq B_i (i = 1, \ldots, l)$ so as to maximize the total weight of the sets in $\mathcal{F}$ having nonempty intersections with X. The maximum coverage problem with a knapsack constraint is NP-hard, and it is $\left(1 - (1 - \frac{1}{k})^k\right)$-approximable by the pipage rounding algorithm where k is the maximum size of sets in the instance. In this paper, we show that the maximum coverage problem with partitioned knapsack constrains has the same approximation ratio by the pipage rounding algorithm.

1 はじめに

最大被覆問題とは, 与えられた集合 $I = \{1, \ldots, n\}$ の部分集合族 $\mathcal{F} = \{S_j : j \in J\}$ $(J = \{1, \ldots, m\})$ と, 非負の重み w_j, 正の整数 p に対して, X と非空の交わりをもつ S_j の重みの総和を最大にする I の部分集合 X $(|X| = p)$ を求めるという問題であり, 次のように定式化される.

$$\max \quad \sum_{j=1}^{m} w_j z_j \tag{1}$$

$$s.t. \quad \sum_{i \in S_j} x_i \geq z_j, \quad j \in J \tag{2}$$

$$\sum_{i=1}^{n} x_i = p \tag{3}$$

$$x_i \in \{0, 1\}, \quad i \in I \tag{4}$$

$$0 \leq z_j \leq 1, \quad j \in J \tag{5}$$

ナップサック制約付き最大被覆問題とは, 与えられた集合 $I = \{1, \ldots, n\}$ の部分集合族 $\mathcal{F} = \{S_j : j \in J\}$ $(J = 1, \ldots, m)$ と, 非負の重み w_j, 費用 c_j, 正の整数 B に対して, ナップサック制約を満たし, X と非空の交わりをもつ S_j の重みの総和を最大にする I の部分集合 X を求めるという問題であり, 次のように定式化される.

$$\max \quad \sum_{j=1}^{m} w_j z_j \tag{6}$$

$$s.t. \quad \sum_{i \in S_j} x_i \geq z_j, \quad j \in J \tag{7}$$

$$\sum_{i=1}^{n} c_i x_i \leq B, \tag{8}$$

$$0 \leq x_i, z_j \leq 1, \quad i \in I, j \in J \tag{9}$$

$$x_i \in \{0, 1\}, \quad i \in I \tag{10}$$

分割されたナップサック制約付き最大被覆問題とは, 集合 $I = \{1, \ldots, n\}$ の部分集合族 $\mathcal{F} = \{S_j : j \in J\}$ $(J = \{1, \ldots, m\})$ と, 非負の重み w_j, 費用 c_j, 正の整数 $B_i (i = 1, \ldots, l)$, ナップサック制約の分割された添字集合 $T_i (i = 1, \ldots, l)$ が与えられたとき, l 本のナップサック制約を満たし, X と非空の交わりをもつ S_j の重みの総和を最大にする I の部分集合 X を求めるという問題であり, 次のように定式化される (以下では, $L = \{1, \ldots, l\}$ とする).

$$\max \quad \sum_{j=1}^{m} w_j z_j \tag{11}$$

$$s.t. \quad \sum_{i \in S_j} x_i \geq z_j, \quad j \in J \tag{12}$$

$$\sum_{i \in T_j} c_{ij} x_i \leq B_j, \quad j \in L \tag{13}$$

$$0 \leq x_i, z_j \leq 1, \quad i \in I, j \in J \tag{14}$$

$$x_i \in \{0, 1\}, \quad i \in I \tag{15}$$

これは, 次のように定式化することもできる.

$$\max \quad F(x) = \sum_{j=1}^{m} w_j \big(1 - \prod_{i \in S_j} (1 - x_i)\big) \tag{16}$$

$$s.t. \quad \sum_{i \in T_j} c_{ij} x_i \leq B_j, \quad j \in L \tag{17}$$

$$0 \leq x_i \leq 1, \quad i \in I \tag{18}$$

$$x_i \in \{0, 1\}, \quad i \in I \tag{19}$$

最大被覆問題は NP 困難であり, 貪欲法によるアルゴリズムでは $\big(1 - (1 - \frac{1}{p})^p\big)$-近似可能 [3], pipage rounding によるアルゴリズムでは $k = \max\{|S_j| : j \in J\}$ としたとき $\big(1 - (1 - \frac{1}{k})^k\big)$-近似可能 [1], $P \neq NP$ の下では $\varepsilon > 0$ に対して $\big(1 - \frac{1}{e} + \varepsilon\big)$-近似不可能であることが知られている [2]. ナップサック制約付き最大被覆問題は, 貪欲法によるアルゴリズムでは $\big(1 - \frac{1}{e}\big)$ 近似可能であり [4], pipage rounding によるアルゴリズムでは $\big(1 - (1 - \frac{1}{k})^k\big)$-近似可能であることが知られている [1].

本論文の構成は以下の通りである. 2 節で計算の準備を行い, 3 節で分割されたナップサック制約付き最大被覆問題に対するアルゴリズムの記述, 4 節でそのアルゴリズムが $\big(1 - (1 - \frac{1}{k})^k\big)$ の性能保証を持つことの証明, 5 節で結論を述べる.

2 準備

$0 \leq y_i \leq 1$, $i = 1, \ldots, k$, $k = \max\{|S_j| : j \in J\}$ に対して, 次の式が成立する.

$$1 - \prod_{i=1}^{k} (1 - y_i) \geq \big(1 - (1 - \frac{1}{k})^k\big) \min\Big\{1, \sum_{i=1}^{k} y_i\Big\} \tag{20}$$

証明. $z = \min\Big\{1, \sum_{i=1}^{k} y_i\Big\}$ とすると, 相加平均 · 相乗平均の関係から次の式が成立する.

$$1 - \prod_{i=1}^{k} (1 - y_i) \geq 1 - (1 - \frac{z}{k})^k \tag{21}$$

また, $0 \leq z = \min\Big\{1, \sum_{i=1}^{k} y_i\Big\} \leq 1$ で $g(z) = 1 - (1 - \frac{z}{k})^k$ は凹関数であるので

$$g(z) \geq \left(1 - (1 - \frac{1}{k})^k\right) z \tag{22}$$

である. (21), (22) より (20) は示された.
$|S_j| = k'(\leq k)$ である, ある j $(1 \leq j \leq m)$ に対しても, (20) より

$$1 - \prod_{i=1}^{k'}(1 - y_i) \geq \big(1 - (1 - \frac{1}{k'})^{k'}\big) \min\Big\{1, \sum_{i=1}^{k'} y_i\Big\}$$

が成立する. さらに, $x \geq 1$ で $(1 - \frac{1}{x})^x$ が単調増加関数であることより

$$1 - \prod_{i=1}^{k'}(1 - y_i) \geq \big(1 - (1 - \frac{1}{k'})^{k'}\big) \min\Big\{1, \sum_{i=1}^{k'} y_i\Big\} \geq \big(1 - (1 - \frac{1}{k})^{k}\big) \min\Big\{1, \sum_{i=1}^{k'} y_i\Big\} \tag{23}$$

であるので, すべての j に対して (23) より

$$1 - \prod_{j \in S_j}(1 - x_i) \geq \big(1 - (1 - \frac{1}{k})^{k}\big) \min\Big\{1, \sum_{i \in S_j} x_i\Big\} \tag{24}$$

が成立する. (24) の両辺に $w_j(> 0)$ を掛けて $j = 1, \ldots, m$ について辺々加えると

$$\sum_{j=1}^{m} w_j \Big(1 - \prod_{j \in S_j}(1 - x_i)\Big) \geq \big(1 - (1 - \frac{1}{k})^{k}\big) \sum_{j=1}^{m} w_j \min\Big\{1, \sum_{i \in S_j} x_i\Big\} \tag{25}$$

が得られる.

3 アルゴリズム

$IP[I_0, I_1]$ で整数計画問題 (6)-(10) を, $LP[I_0, I_1]$ でその線形緩和 (6)-(9) を表すとする (I_0, I_1 はともに交わりを持たない I の部分集合であり, $i \in I_0$ に対して $x_i = 0$, $i \in I_1$ に対して $x_i = 1$ である).
$LP[I_0, I_1]$ に対する実行可能解 $x^{\mathcal{A}}$ を発見するアルゴリズム $\mathcal{A}$ を記述する (([1], Ageev and Sviridenko, 2004) を再掲する).

アルゴリズム $\mathcal{A}$
フェーズ 1. $LP[I_0, I_1]$ に対する最適解 x^{LP} を見つける.
フェーズ 2. 一連の pipage ステップによって x^{LP} を $x^{\mathcal{A}}$ に変形する.

0. $x^{\mathcal{A}} \leftarrow x^{LP}$ とする.
1. $x^{\mathcal{A}}$ の高々 1 つの要素が小数であるとき, 終了.
2. それ以外のとき, $0 < x^{\mathcal{A}}_{i'} < 1, 0 < x^{\mathcal{A}}_{i''} < 1$ である添字 i',i'' を選ぶ.
   $x^{\mathcal{A}}_{i'}(\varepsilon) \leftarrow x^{\mathcal{A}}_{i'} + \varepsilon, x^{\mathcal{A}}_{i''}(\varepsilon) \leftarrow x^{\mathcal{A}}_{i''} - \varepsilon \frac{c_{i'}}{c_{i''}}$ として, $\forall k \neq i', i''$ に対しては $x^{\mathcal{A}}_{k}(\varepsilon) \leftarrow x^{\mathcal{A}}_{k}$ とする.
   $F(x(\varepsilon*)) > F(x^{\mathcal{A}})$ となる, 区間 $\Big[-\min\Big\{x_{i'}, (1 - x_{i''})\frac{c_{i''}}{c_{i'}}\Big\}, \min\Big\{1 - x_{i'}, x_{i''}\frac{c_{i''}}{c_{i'}}\Big\}\Big]$ の端点 $\varepsilon*$ を選ぶ.
   $x^{\mathcal{A}} \leftarrow x(\epsilon*)$ として 1 に戻る.

· フェーズ 2 は, 各 ε に対して $x(\varepsilon)$ が実行可能であることと $F(x(\varepsilon))$ が凸であることから $F(x(\varepsilon*)) > F(x^{\mathcal{A}})$ となる端点 $\varepsilon*$ が存在することを利用している.
· フェーズ 2 の各 pipage ステップで, アルゴリズム $\mathcal{A}$ は $x^{\mathcal{A}}$ の要素の中で小数であるものの個数を少なくとも 1 つずつ減らしていくので, 最終的に高々 1 つの要素が小数である実行可能解 $x^{\mathcal{A}}$ を出力される.
(25) より, $\forall j \in J \setminus J_1$ に対して

$$\sum_{j\in J\setminus J_1} w_j\Big(1-\prod_{j\in S_j}(1-x_i)\Big) \geq \big(1-(1-\frac{1}{k})^k\big)\sum_{j\in J\setminus J_1} w_j \min\Big\{1,\sum_{i\in S_j} x_i\Big\} \tag{26}$$

が成立する. ただし, $J_1 = \{j : S_j \cap I_1 \neq \emptyset\}$ とする.
$\forall j \in J_1$ に対して

$$\sum_{j\in J_1} w_j\Big(1-\prod_{j\in S_j}(1-x_i)\Big) = \sum_{j\in J_1} w_j \tag{27}$$

が成立する.
(16) より

$$\begin{aligned} F(x^{LP}) &= \sum_{j\in J} w_j\Big(1-\prod_{j\in S_j}(1-x_i^{LP})\Big) \\ &= \sum_{j\in J_1} w_j\Big(1-\prod_{j\in S_j}(1-x_i^{LP})\Big) + \sum_{j\in J\setminus J_1} w_j\Big(1-\prod_{j\in S_j}(1-x_i^{LP})\Big) \end{aligned} \tag{28}$$

である.
アルゴリズム $\mathcal{A}$ の構成より, $F(x^{\mathcal{A}}) \geq F(x^{LP})$ であることと, (26), (27), (28) より

$$\begin{aligned} F(x^{A}) &\geq F(x^{LP}) \\ &\geq \sum_{j\in J_1} w_j + \big(1-(1-\frac{1}{k})^k\big)\sum_{j\in J\setminus J_1} w_j \min\Big\{1,\sum_{i\in S_j} x_i^{LP}\Big\} \end{aligned} \tag{29}$$

が成立する.

次に, アルゴリズム全体を記述する (([1], Ageev and Sviridenko, 2004) を元に構成する).
入力:整数計画問題の例 (11)-(15)
出力:その例に対する実行可能解 $\bar{x}$
0. $\sum_{i\in I} x_i \leq 4l-1$ である全ての実行可能解の中で, 目的関数を最大にするものを x^0 とする.
   $\bar{x} \leftarrow x^0$ とする.
1. $|I_1| = 4l$ かつ $\sum_{i\in I_1} \leq B_j,\ \forall j \in J$ である全ての I_1 に対して 2 以降を計算する.
2. $I_0 \leftarrow \emptyset, t \leftarrow 0$ とする.
3. $t = 0$ のとき, アルゴリズム $\mathcal{A}$ を $LP[I_0, I_1]$ の l 本のナップサック制約 (13) それぞれに適用する.
   · 全ての $x_i^{\mathcal{A}}$ が整数のとき, $t \leftarrow 1, \hat{x} \leftarrow x^{\mathcal{A}}$ として終了.
   · l' 個 $(1 \leq l' \leq l)$ の小数解が存在するとき
   $x_i^{\mathcal{A}}$ が小数である添字の集合を L' として, 次の操作をする.
   $\hat{x}_i \leftarrow 0, \forall i \in L'$
   $\forall i \in I \setminus L'$ に対しては, $\hat{x}_i \leftarrow \hat{x}_i^{\mathcal{A}}$ とする.

$I_0 \leftarrow I_0 \cup L'$ とする.
$F(\hat{x}) > F(\bar{x})$ ならば, $\bar{x} \leftarrow \hat{x}$ とする.
3 を繰り返す.

4 性能保証

4.1 アルゴリズムの詳細

与えられた問題例に対する 最適集合を $|X^*|$, それに対応するベクトルを x^* とする. $|X^*| \leq 4l-1$ のとき, このアルゴリズムで最適解が出力されることが分かるので, $|X^*| \geq 4l$ の場合を考える. 一般性を失わず, $X^* = \{1, 2, \ldots, |X^*|\}$, $w_1 \geq w_2 \geq \cdots \geq w_{|X^*|}$ であるとできる. $I_1 = \{1, 2, 3, 4\}$ のときの反復について考える. この反復が q 回 $(4 \leq q \leq n-4)$ で終わるとする. t 回目の反復において, $IP[I_0^t, I_1]$ に対する高々 1 つの要素が小数である実行可能解 $x^t = x^{\mathcal{A}}$ を得るためにアルゴリズム $\mathcal{A}$ が呼び出される $(I_0^t = i_1, \ldots, i_{t-1}$ とする). このとき x^t の全ての要素が整数ならば反復を $t = q$ として終了する. x^t の要素に l' 個 $(1 \leq l' \leq l)$ の小数解が残っているとき, 小数解となっているものの添字集合を L' とすると, $\forall i \in I \setminus L'$ に対しては $\hat{x}_i^t \leftarrow x_i^t$, $\forall i \in L'$ に対しては $\hat{x}_i^t \leftarrow 0$ とする. $F(\hat{x}^t) > F(\bar{x})$ であれば $\bar{x} \leftarrow \hat{x}$ と更新する. $I_0^{t+1} = I_0^t \cup L'$ として次の反復へ.

4.2 性能保証の証明

以下では, ([1], Ageev and Sviridenko, 2004) を元に, 分割されたナップサック制約付き最大被覆問題の性能保証が $\big(1-(1-\frac{1}{k})^k\big)(k = \max\{|S_j| : j \in J\})$ であることを示す.
· $X^* \cap I_0^q = \emptyset$ の場合, x^* は $IP[I_0^q, I_1]$ に対する最適解である. x^q は $LP[I_0^q, I_1]$ の最適解で $\hat{x}^q = x^q$ であることと, (29) より

$$\begin{aligned} F(\hat{x}^q) = F(x^q) &\geq \sum_{j \in J_1} w_j + \big(1-(1-\frac{1}{k})^k\big) \sum_{j \in J \setminus J_1} w_j \min\Big\{1, \sum_{i \in S_j} x_i^q\Big\} \\ &\geq \sum_{j \in J_1} w_j + \big(1-(1-\frac{1}{k})^k\big) \sum_{j \in J \setminus J_1} w_j \min\Big\{1, \sum_{i \in S_j} x^*\Big\} \\ &\geq \big(1-(1-\frac{1}{k})^k\big) \sum_{j \in J} w_j \min\Big\{1, \sum_{i \in S_j} x^*\Big\} \end{aligned}$$

であるので, 性能保証は $\big(1-(1-\frac{1}{k})^k\big)$ である.
· $X^* \cap I_0^q \neq \emptyset$ の場合, I_0^{s+1} を $I_0^1 = \emptyset, \ldots, I_0^q$ の中で初めて X^* と交わりを持つ集合であるとする. 言い換えると, i_s は $i_1, \ldots, i_{q-1}$ の中で初めて X^* に出てくる添字であるとする. 以下では

$$F(\hat{x}^q) \geq \cdots \geq F(\hat{x}^s) \geq \big(1-(1-\frac{1}{k})^k\big) F(x^*)) \tag{30}$$

を証明する.
証明. アルゴリズムの構成より $F(\hat{x}^q) \geq \cdots \geq F(\hat{x}^s)$ は明らか.
関数 F は全ての部分集合 $X \subseteq I$ 上で定義された集合関数 $F(X)$ として扱うことができる. $F(X)$ は劣モジュラ関数であり, 次の性質をもつ.

$$F(X \cup i) - F(X) \geq F(X \cup Y \cup i) - F(X \cup Y), \forall X, Y \subseteq I, i \in I \tag{31}$$

$i \in I, Y \supseteq I_1$ として (31) を用いると, 次の式が成立する.

$$\begin{aligned}
\frac{1}{4l}F(I_1) &= \frac{1}{4l}F(1,2,\ldots,4l) \\
&= \frac{1}{4l}\big(F(1,2,\ldots,4l) - F(1,2,\ldots,4l-1) \\
&\quad + F(1,2,\ldots,4l-1) - F(1,2,\ldots,4l-2) \\
&\quad + \ldots \\
&\quad + F(1,2) - F(1) \\
&\quad + F(1) - F(\emptyset)\big) \\
&\geq \frac{1}{4l}\big(F(1,2,\ldots,4l-1,i) - F(1,2,\ldots,4l-1) \\
&\quad + F(1,2,\ldots,4l-2,i) - F(1,2,\ldots,4l-2) \\
&\quad + \ldots \\
&\quad + F(1,i) - F(1) \\
&\quad + F(i) - F(\emptyset)\big) \\
&\geq F(Y \cup i) - F(Y)
\end{aligned}$$

このように, $\forall Y \supseteq I_1, \forall i \in I$ に対して次の式が成立することが証明された.

$$\frac{1}{4l}F(I_1) \geq F(Y \cup i) - F(Y) \tag{32}$$

$\hat{x}^s$ は x^s の l' 個 $(1 \leq l' \leq l)$ の小数である要素を 0 にしたものであり, x^s において小数解を持つ要素を $x^s_{i_1}, \ldots, x^s_{i_{l'}}$ とすると, 次の式が成立する.

$$F(\hat{X}_s \cup i_1 \cup \cdots \cup i_{l'}) \geq F(x_s) \tag{33}$$

(26), (32), (33) を用いると次の式が得られる.

$$
\begin{aligned}
F(\hat{x}^s) &= F(\hat{X}_s) \\
&= F(\hat{X}_s \cup i_1) - \big(F(\hat{X}_s \cup i_1) - F(\hat{X}_s)\big) \\
&\geq F(\hat{X}_s \cup i_1) - \frac{1}{4l} F(I_1) && (by(32)) \\
&= F(\hat{X}_s \cup i_1 \cup i_2) - \big(F(\hat{X}_s \cup i_1 \cup i_2) - F(\hat{X}_s \cup i_1)\big) - \frac{1}{4l'} F(I_1) \\
&\geq F(\hat{X}_s \cup i_1 \cup i_2) - \frac{2}{4l'} F(I_1) && (by(32)) \\
&\cdots \\
&\geq F(\hat{X}_s \cup i_1 \cdots \cup i_{l'}) - \frac{l'}{4l'} F(I_1) && (by(32)) \\
&= F(\hat{X}_s \cup i_1 \cdots \cup i_{l'}) - \frac{1}{4} F(I_1) \\
&\geq F(x^s) - \frac{1}{4} F(I_1) && (by(33)) \\
&= \sum_{j \in J_1} w_j + \sum_{j \in J \setminus J_1} w_j \big(1 - \prod_{i \in S_j} (1 - x_i^s)\big) - \frac{1}{4} \sum_{j \in J_1} w_j \\
&\geq \frac{3}{4} \sum_{j \in J_1} w_j + \big(1 - (1 - \frac{1}{k})^k\big) \sum_{j \in J \setminus J_1} w_j \min\Big\{1, \sum_{i \in S_j} x_i^{LP}\Big\} && (by(26)) \\
&\geq \big(1 - (1 - \frac{1}{k})^k\big) \Big(\sum_{j \in J_1} w_j + \sum_{j \in J \setminus J_1} w_j \min\Big\{1, \sum_{i \in S_j} x_i^{LP}\Big\} \Big) && (k \geq 2) \\
&\geq \big(1 - (1 - \frac{1}{k})^k\big) \Big(\sum_{j \in J_1} w_j + \sum_{j \in J \setminus J_1} w_j \min\Big\{1, \sum_{i \in S_j} x_i^{*}\Big\} \Big) \\
&= \big(1 - (1 - \frac{1}{k})^k\big) F(x^*)
\end{aligned}
$$

以上より, (30) が成立し, $X^* \cap I_0^q \neq \emptyset$ の場合も性能保証 $\big(1 - (1 - \frac{1}{k})^k\big)$ を持つことが示された.

5 おわりに

本研究では, 分割されたナップサック制約付き最大被覆問題に対する pipage rounding によるアルゴリズムの設計と, そのアルゴリズムが $\big(1 - (1 - \frac{1}{k})^k\big)(k = \max\{|S_j| : j \in J\})$ の性能保証を持つことを示した. ([2], Feige, 1998) では最大被覆問題は $\varepsilon > 0$ に対して $\big(1 - \frac{1}{e} + \varepsilon\big)$-近似不可能であることが示されているので, 本論文で示した分割されたナップサック制約付き最大被覆問題に対する性能保証は有意なものであると言える.

参考文献

[1] A.A. Ageev and M.I. Sviridenko, "Pipage Rounding: A New Method of Constructing Algorithms with Proven Performance Guarantee," Journal of Combinatorial Optimization, 8, pp. 307-328, 2004.

[2] U. Feige, "A threshold of ln n for approximating set cover," J. of ACM., vol. 45, pp. 634-652, 1998.

[3] D.S. Hochbaum, "Approximating covering and packing problems: Set Cover, Vertex Cover, Independent Set, and related problems," in Approximation algorithms for NP-hard problems, D.S.Hochbaum (Ed.), PWS Publishing Company: New York, pp. 94-143, 1997.

[4] S. Khuller, A. Moss, and J. Naor, "The budgeted maximum coverage problem," Information Processing Letters, vol. 70, pp. 39-45, 1999.

計算機アーキテクチャの教育を支援する可視化レベル変更可能な PDP-11 シミュレータ

林　毅
法政大学　大学院
理工学研究科　応用情報工学専攻

和田　幸一
法政大学
理工学部　応用情報工学科

計算機アーキテクチャの学習を支援する PDP-11 シミュレータを考える．提案するシミュレータは PDP-11 のマシンサイクルの各ステージの動作を提示するだけでなく，ステージ毎に内部動作を詳しく提示し，バスの制御のレベルを可視化することができる．このシミュレータは，コンピュータの内部動作をより深いレベルまで学習できることを目的としている．

1.はじめに

法政大学では，計算機アーキテクチャとアセンブリ言語の講義がある．現在アセンブリ言語の講義では，COMET2 に対応したアセンブリ言語 CASL2 のシミュレータを用いて学生がアセンブリ言語を学んでいる． COMET2 はアセンブリ言語の学習用に開発された仮想計算機であるため，アセンブリのプログラムとしてアドレッシングモードなどの機能があるが，実用化されている CPU に比べると種類が少なく，単純であり，計算機アーキテクチャを学ぶにはもの足りないところがある．そのため計算機アーキテクチャの講義では PDP-11 を用いて講義が行われている．そこで，今回はその PDP-11 のシミュレータの提案，実装を行い，計算機アーキテクチャの教育を支援する．また，アセンブリ言語の教育支援に関しては「アセンブリ言語教育支援教材 Sim AI の設計と実装(小林 晴紀)」を参照する．

2. PDP-11

PDP-11 は DEC 社が開発した 16 ビットミニコンピュータである．命令は 1 語 16 ビットで，命令数は 60 種類あるため，オペレーションコードは可変長になっている．番地はバイト毎についているバイトマシンで，1 バイト(8 ビット)のデータに対する演算も用意されている．汎用レジスタ（general register)は R0 から R7 までの 8 本あり，これは 3 ビットで指定する．そのうち R7 はプログラムカウンタ（program counter) PC を兼ねており，また，R6 はスタックポインタ（stack pointer）として使用される．メモリの番地（address）は 16 ビットで指定するので，主記憶は最大でも 32K 語(64kB)までとなる．そのうち，実記憶は 28K 語で，最上位番地の 4k 語は特別な用途に割り当てられている．(参考文献[1])

3.PDP-11 シミュレータ

シミュレータの基本的な機能を述べる．提案するシミュレータではアセンブリ言語の記述が可能で，ユーザーは自由にプログラムの編集，実行が可能である．また，エディターウインドウではレジスタの内容とメモリの内容が確認できる．本シミュレータでは，プログラムを実行したあと，ユーザーはプログラムの実行に対応したマシンサイクルの内部動作をマシンサイクルウインドウで確認できる．

3.1. PDP-11 シミュレータの特徴

このシミュレータではマシンサイクルを動作の抽象度によって，レベルに分けたシミュレートを可能としている．そのレベルを 3 段階に分け，レベル 1～3 とする．レベル 1 では，アドレスの内容やデコード結果,演算結果を表示させて,マシンサイクルの大まかな流れを見せる．レベル 2 では，フェッチされたアドレスの内容やデコードされた命令がどのように処理されているのかを表示させて,各ステージの細かい流れを見せる．レベル 3 では，メモリへのアクセス方法やデコード方法,演算方法をバスレベルでより細かく表示させて,各ステージの処理をより細かく見せる．各レベルで提示するものを表 1 に示す．また，このシミュレータで学習できる計算機アーキテクチャの機能を表 2 に示す．

表 1．各レベルで提示するもの

レベル 1	レジスタ，主記憶，各ステージの入出力を提示させてマシンサイクルの大まかな流れを理解させる．
レベル 2(フェッチステージ)	プログラムカウンタが指すアドレスがどのように保持されて，デコードステージへ渡されるのかを提示する．
レベル 2(デコードステージ)	フェッチステージからもらったデータがどのようなデータであるのかを提示する． 2 語命令以上の場合は，再びフェッチステージに移行することがわかるようにもする．
レベル 2(実行ステージ)	各命令がどのようにして，実行されるのかを図示する． 計算式や条件の分岐を見せる．
レベル 3(フェッチステージ)	割り込み処理の動作を提示する． バスの同期の取り方を理解させる．
レベル 3(デコードステージ)	命令がデコード内でどのようにして決められているのかを，セレクタ回路などを提示させて理解させる．
レベル 3(実行ステージ)	データがどのようにして演算されているのかを，論理回路レベルで図示し， ALU の構成，動作を理解させる．

表 2．各レベルで学習できる計算機アーキテクチャ

レベル 1	CPU の構造 命令サイクル
レベル 2	命令形式，演算コード，アドレッシングモード(直接、即値) オペランド アドレッシングモードの実現 サブルーチンの実現 命令の記号化 バスの構造 論理演算とシフト 算術論理演算命令 加減乗除命令、ロード・ストア命令 分岐命令 機械語命令形式 具体的な計算機の機械語命令 スタックの概念 スタックポインタ スタック操作命令 サブルーチンの基本 サブルーチンの概念
レベル 3	バスの制御(Data in，Data out) ALU の構成 割り込みの概念 割り込み要因・動作

	割り込み用命令と割り込みベクトル

3.1.1. シミュレートレベル 1

シミュレートレベル 1 では以下のものを図示する．このレベルで表示させるマシンサイクルウインドウを図 3.1 に示す．

・アドレスバス(AB)
・データバス(DB)
・CPU の内部バス
・汎用レジスタ R0～R7 の内容
・メモリの内容
・フェッチステージに入力されるメモリのアドレスと内容
・デコードステージで解析された命令のニモニックと機械語
・実行ステージで実行された演算名と演算結果と条件フラグ

シミュレートレベル 1 では，レジスタの内容の変化や各ステージの入出力の結果を表示させて，マシンサイクルの大まかな流れを理解する．

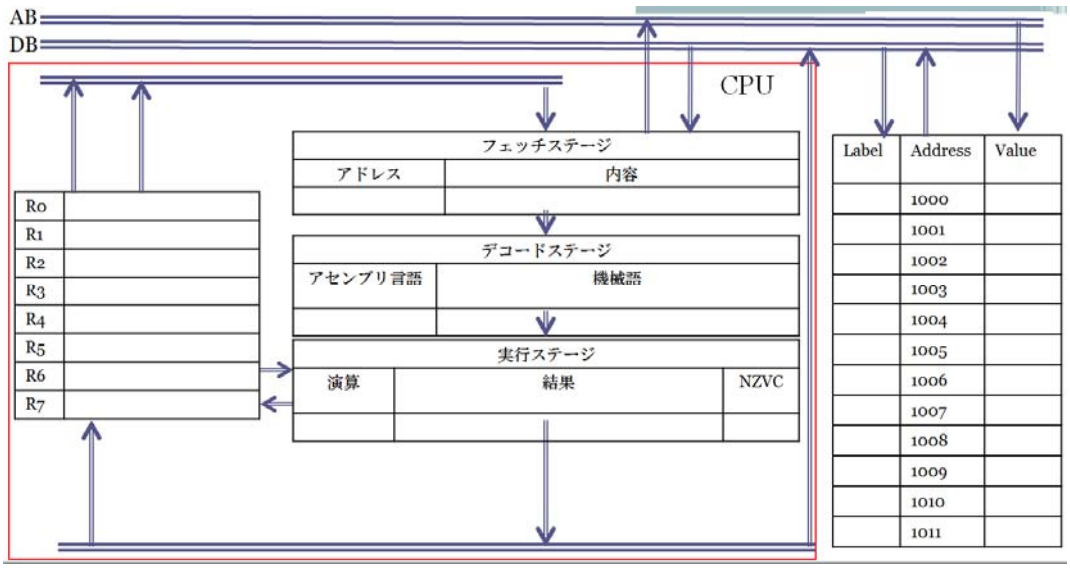


図 3.1.提案するマシンサイクルウインドウ

3.1.2. シミュレートレベル 2

より細かく見たいステージの内部動作をレベル 1 より掘り下げてみることができる．

フェッチステージでは，PC(Program Counter)の内容が内部バスをとおって BAR(Bus Address Register)に格納される．その値が指すアドレスの内容が BDR(Bus Data Register)に格納され，IR(Instruction Register)に送られる．(図 3.2)この一連の動作がフェッチステージのシミュレートレベル 2 となる．

フェッチステージのシミュレートレベル 2 で表示するものをいかに示す．

・アドレスバス(AB)
・データバス(DB)
・CPU の内部バス
・メモリの内容
・汎用レジスタ R7(PC)
・バスアドレスレジスタ(BAR)
・バスデータレジスタ(BDR)

・インストラクションレジスタ(IR)

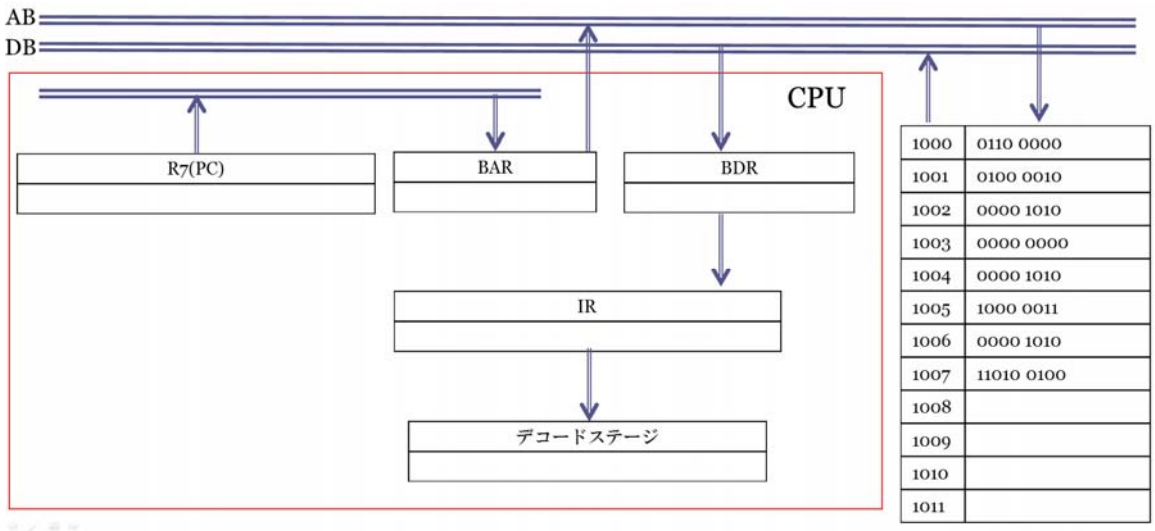


図 3.2.フェッチステージのシミュレートレベル 2

デコードステージでは，IR に格納されている内容の命令の種類，その命令の説明，機能を表示させる．その命令の構成，opc(operation code)，NZVC(条件フラグ)，dst(destination operand),src(source operand)などの情報も加えて表示させる．(図 3.3)命令が 2 語命令以上の場合，フェッチステージに移行することを理解させるために，2word の命令の場合は，図中の fetch の部分が変色する．

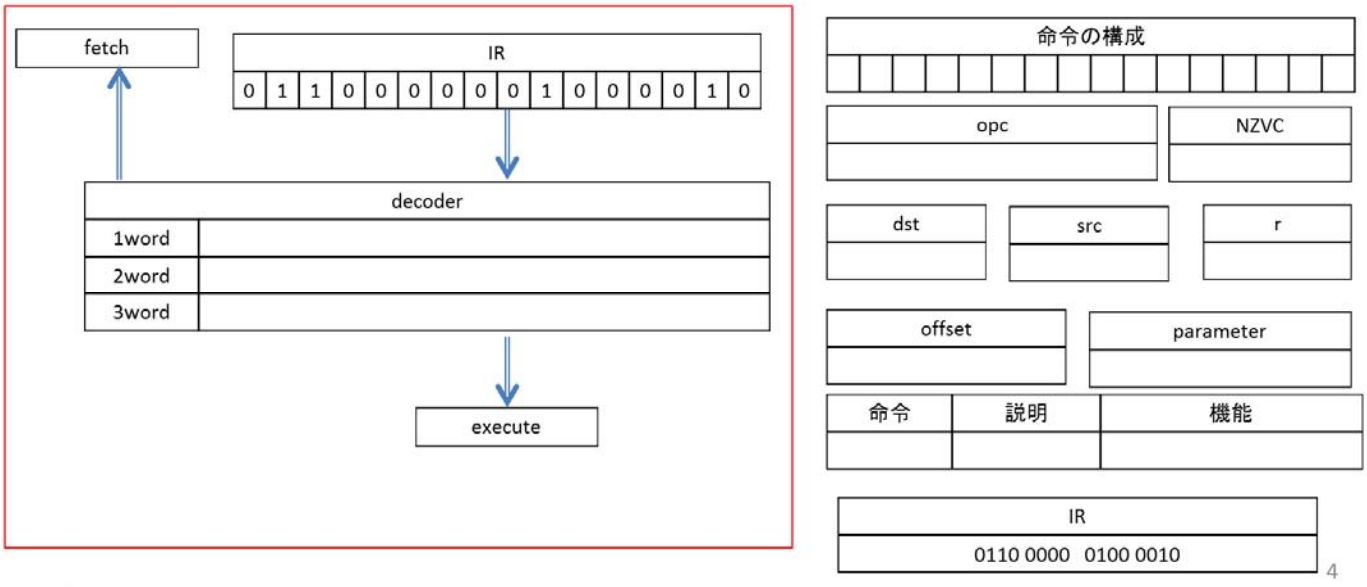


図 3.3.デコードステージのシミュレートレベル 2

実行ステージでは，各命令に対応した形式で図示し，実行時の CPU の内部動作を可視化する．

算術演算命令

16 ビットの 2 の補数表現として演算を行い，結果によって条件コード NZVC を設定する．

表 3. 実行ステージで可視化される算術演算命令

算術演算命令	比較 加算 減算 クリア 1 加算 1 減算 2 の補数 キャリー加算 キャリー減算 テスト

算術演算命令では，表 3 に示す演算命令の動作が可視化される．ウインドウには，IBA，IBB，レジスタ，dst の値，src の値，演算内容，演算結果，PSW が図示されており，これらの値とデータの移動が確認できる(図 3.3)．また，クリア，1 加算命令など，src が使われず決まった

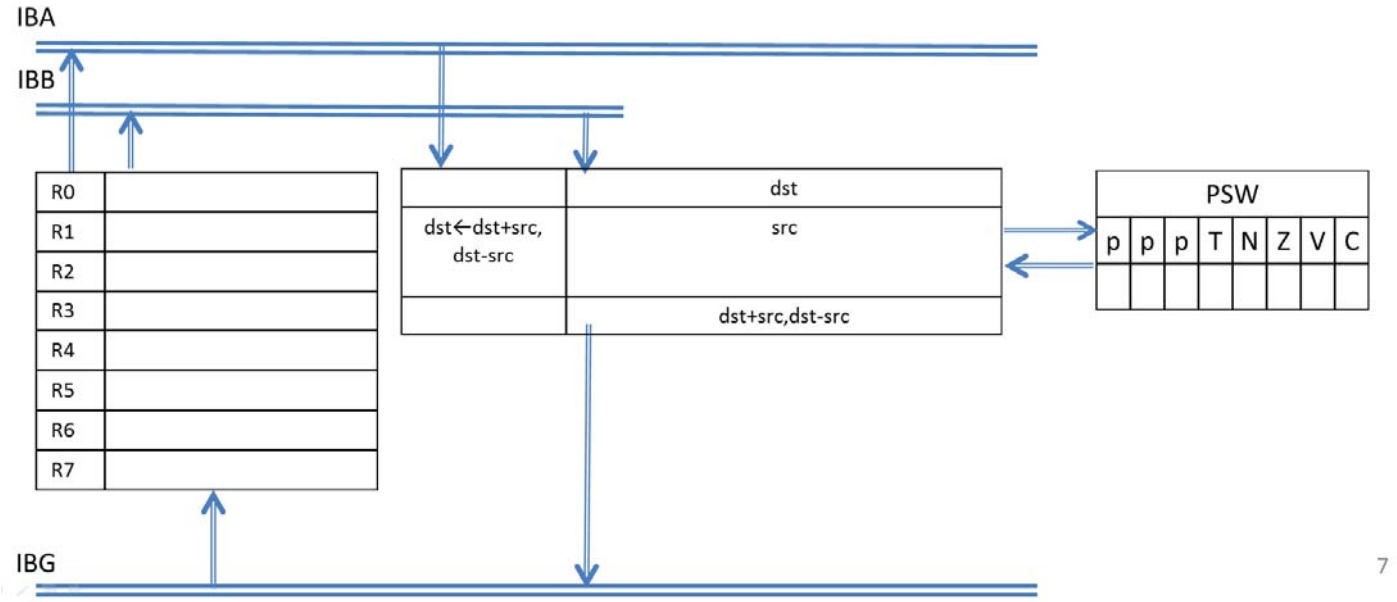


図 3.3. 実行ステージ算術命令(例 1 加算，減算)

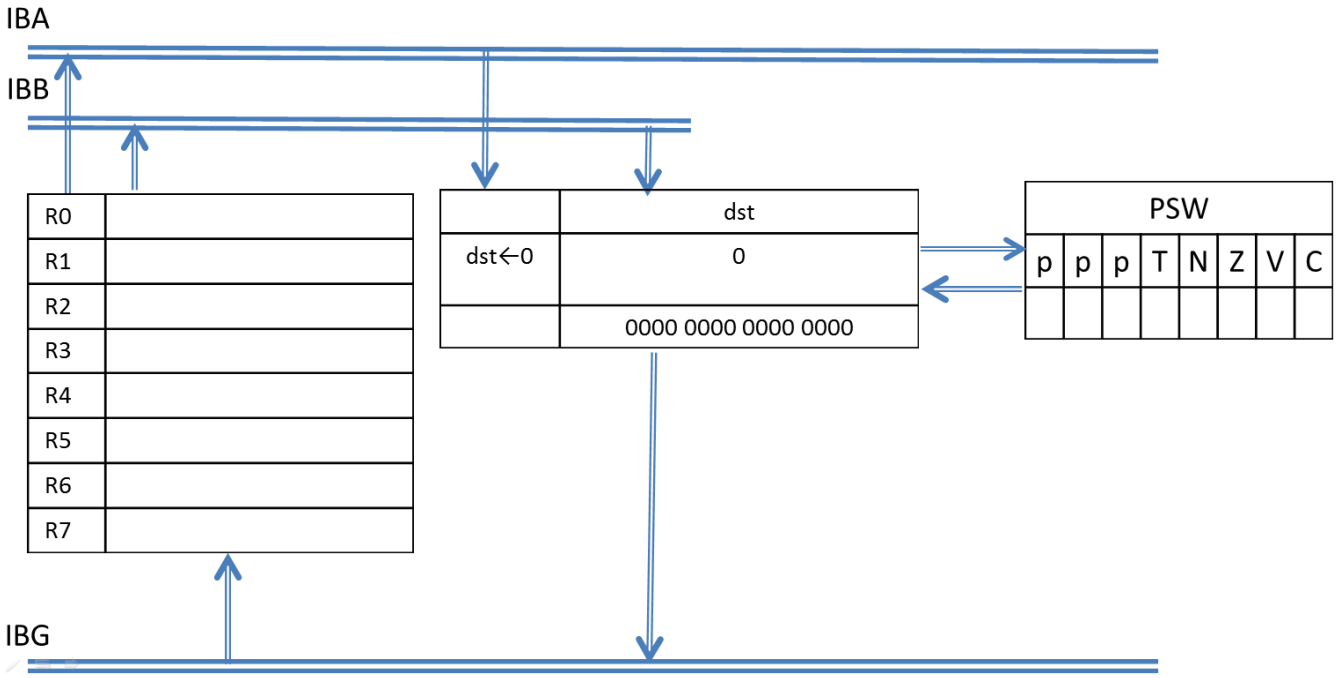


図 3.4. 実行ステージ算術命令(例 2 クリア)

論理演算命令

論理演算命令では，表 4 に示す演算命令の動作が可視化される．

表 4. 実行ステージで可視化される論理演算命令

論理演算命令	転送 ビットテスト ビットクリア ビットセット 反転 バイト交換

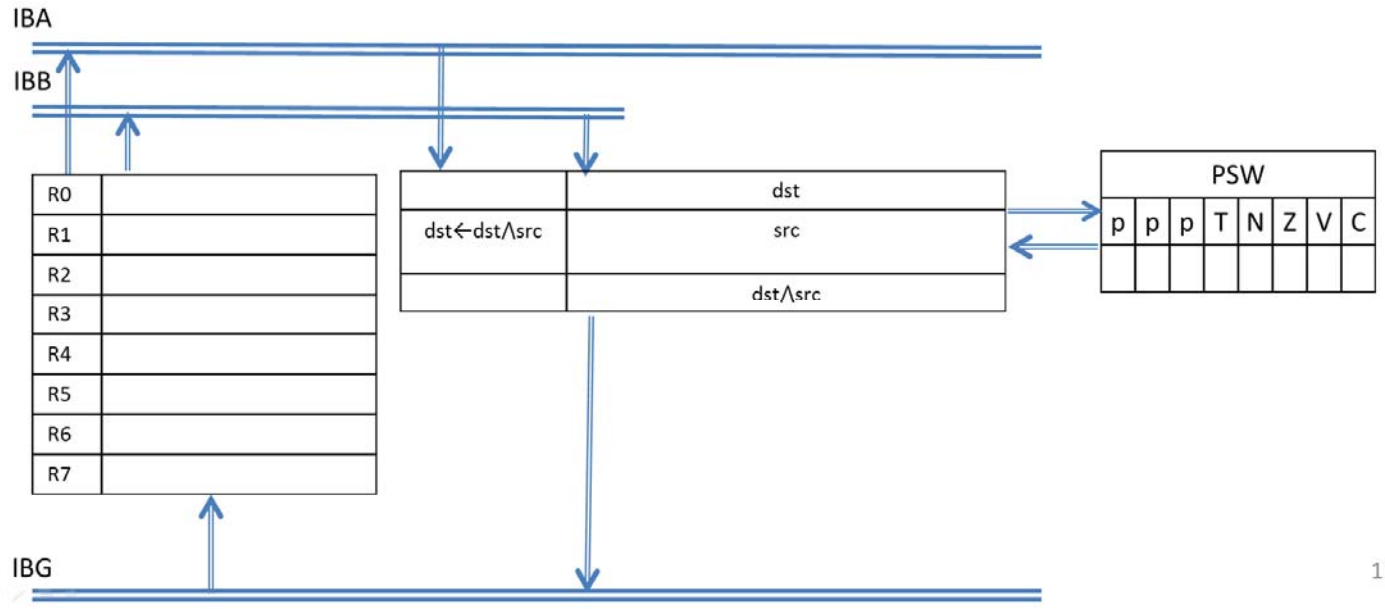


図 3.5. ビットテスト

シフト命令

シフト命令は，データを 1 ビット，右か左に動かすもので，右か左かと，算術データか，論理データかで 4 通りのシフト法がある(表 5)．シフト命令のウインドウ画面では，IBA，IBB，IBG，レジスタ，dst，C，PSW を表示させる．シフト前後のデータの変化をユーザーに理解してもらい，その時の条件コードもわかるようにする．

表 5．実行ステージで可視化されるシフト命令

シフト命令	右巡回シフト 左巡回シフト 算術右シフト 算術左シフト

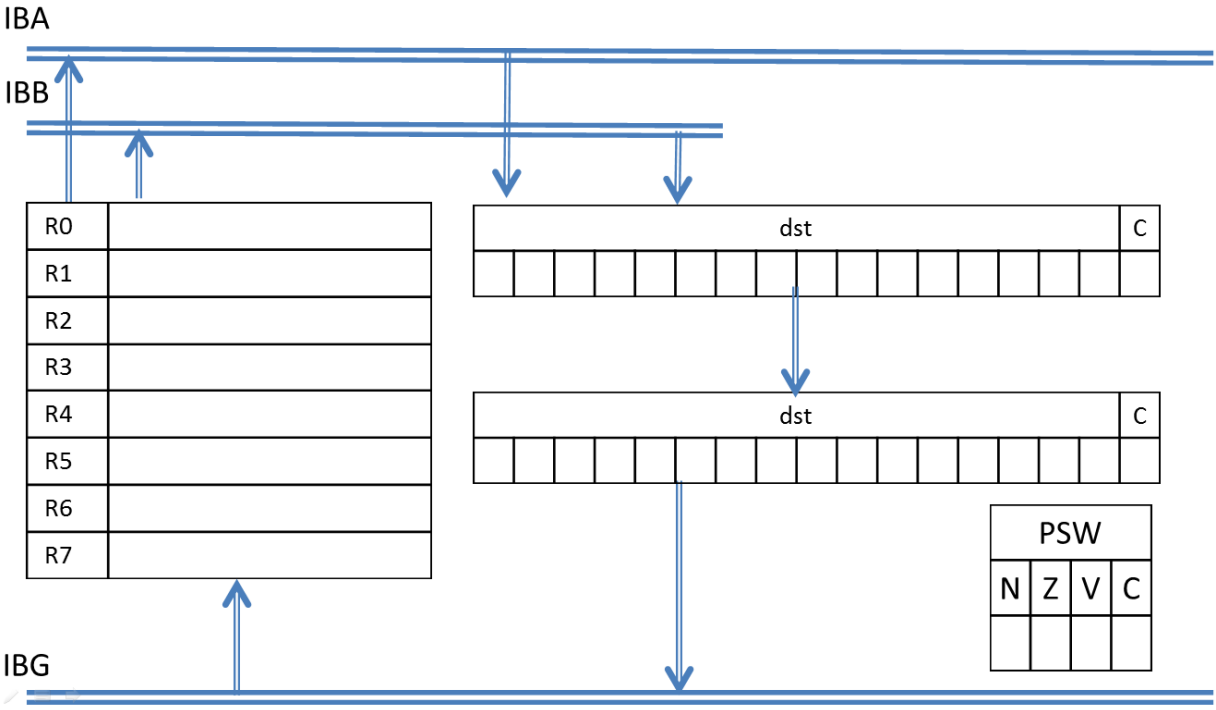


図 3.6．シフト演算

バイト命令

いくつかの命令は対応したバイトデータに対する演算をする命令がある(表 6)．

バイト命令は基本的に，算術演算，論理演算，シフト命令と同じ動作をするためウインドウは上記の図と同じように図示する．しかし，アドレッシングモードのバイト命令の場合，処理するアドレスは下位バイトのみである．

表 6．実行ステージで可視化されるバイト命令

バイト命令	転送 比較 ビットテスト ビットクリア ビットセット クリア 反転 1 加算 1 減算

	2 の補数 キャリー加算 キャリー減算 テスト 右巡回シフト 左巡回シフト 算術右シフト 算術左シフト

分岐命令

分岐命令は，条件コードの値(表 7)によって，PC の値をそのまま使うか，あるいは，オフセットを加算して使うかを切り替える．

表 7．実行ステージで可視化される分岐命令

分岐命令	分岐条件
	True Z=0 Z=1 N=0 N=1 V=0 V=1 C=0 C=1
	符号付分岐
	N=V N≠V Z=1∧N=V Z=1∨N≠V
	符号なし分岐
	C∨Z=0 C∨Z=1 C=0 C=1

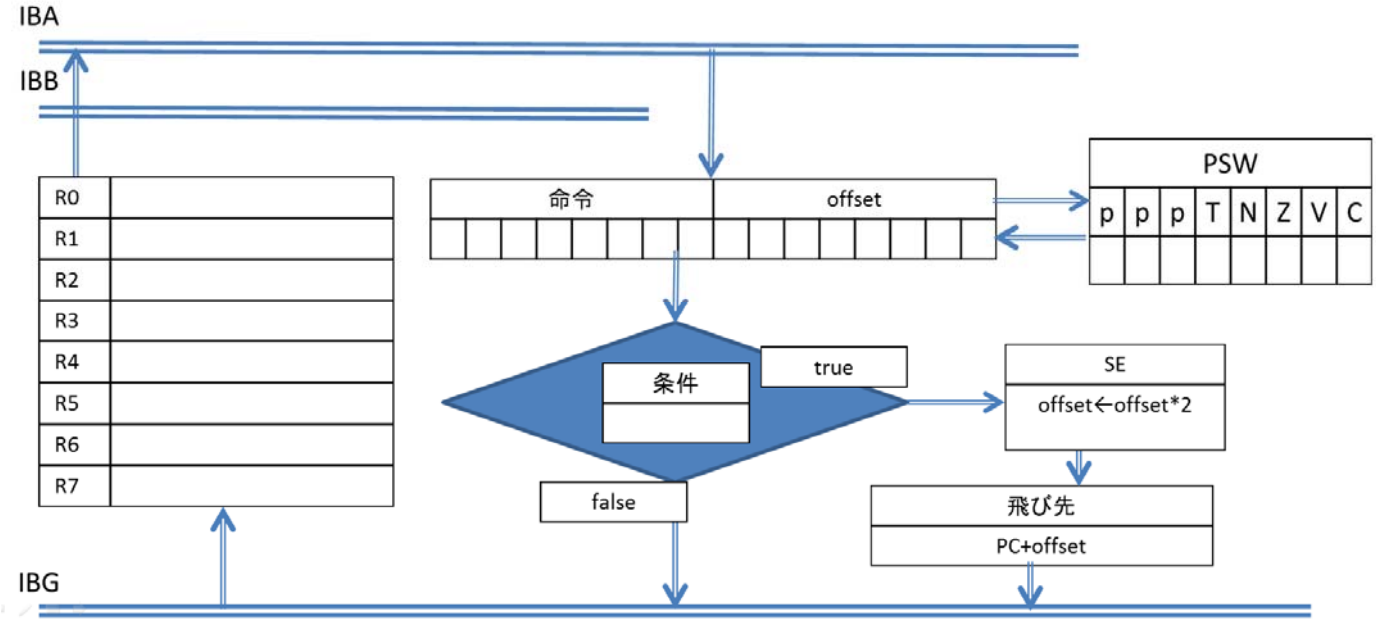


図 3.7. 分岐命令

その他の実行制御命令

その他の実行制御命令では，表 4 に示す演算命令の動作が可視化される．

表 8. 実行ステージで可視化されるその他の実行制御命令

その他の実行制御命令	ジャンプ命令 サブルーチン呼び出し サブルーチンからの戻り ソフトウェア割り込み 割り込みからの戻り リセット，待ち 停止 ソフトウェア割り込み トラップ

アドレッシングモード

PDP-11 の特色のひとつはアドレシングモードである．オペランドの指定では，コンピュータの中のどの場所のデータにアクセスするかを決める．そのデータは，大きく分けると汎用レジスタ，主記憶，命令中に書き込まれているデータである．主記憶はバイト毎に番地が与えられており，16 ビットの番地で指定される．レジスタは 8 個あり，3 ビットで指定できる．R7 はプログラムカウンタ(program counter), R6 はスタックポインタ(stack pointer)として使用する．したがって，アプリケーションプログラムで汎用レジスタとして自由に使えるのは R0 から R5 の 6 つである．ただ，PDP-11 では，PC やスタックポインタを特別扱いせず，汎用レジスタとして平等に扱うことを選択したために，豊富なアドレシングモード（addressing mode）が可能となった．8 つのアドレシングモードがあり，やや複雑に見えるが，実際によく使うのはモード 0，2，4，6 で，モード 3，5，7 はあまり使うことはない．あと，レジスタとして，プログラムカウンタ PC を使うとき，直値，絶対番地，相対番地など重要なアドレシングモードになる．ソースオペランド，デスティネーションオペランドは，機械語では 6 ビットであるが，それはつぎのように，上位 3 ビットがモード，下位 3 ビットがレジスタの指定に使われる．アドレッシングモードの種類を表 7 に示す．

表 9. 実行ステージで可視化されるアドレッシングモード

アドレッシングモード	
直接アドレッシング	レジスタ 自動加算 自動減算 インデックス
間接アドレッシング	レジスタ間接 自動加算間接 自動減算間接 インデックス間接
PC を使う	直値 絶対 相対 相対間接

3.1.3. シミュレートレベル 3

シミュレートレベル 3 では割り込み処理，デコーダーの内部動作，ALU の内部動作を提示する．

割り込み処理

割り込みはコンピュータ内外の状況によって，通常の処理を一時中断し，その状況に応じた処理を優先して実行する仕組みである．シミュレートレベル 3 の割込命令では,新たに MSYN(master synchronization),SSYN(slave synchronization),read,write を表示させる(図 3.8).これらは,MSYN と SSYN で同期をとり,read と write の値を書き換えて割り込みの制御を行う．

シミュレートレベル 3 では,より複雑な計算機アーキテクチャの構造となるので,さらに詳しくみることもできる.割込み命令では,図 3.9 のように,AB,DB,CB(control bus)に乗っているデータを表示させ,CPU とメモリ間でどの用にデータがやりとりされているのか細かくみることができる.また,Data in の図では,割込み制御の同期の取り方の順序を追って確認できる．

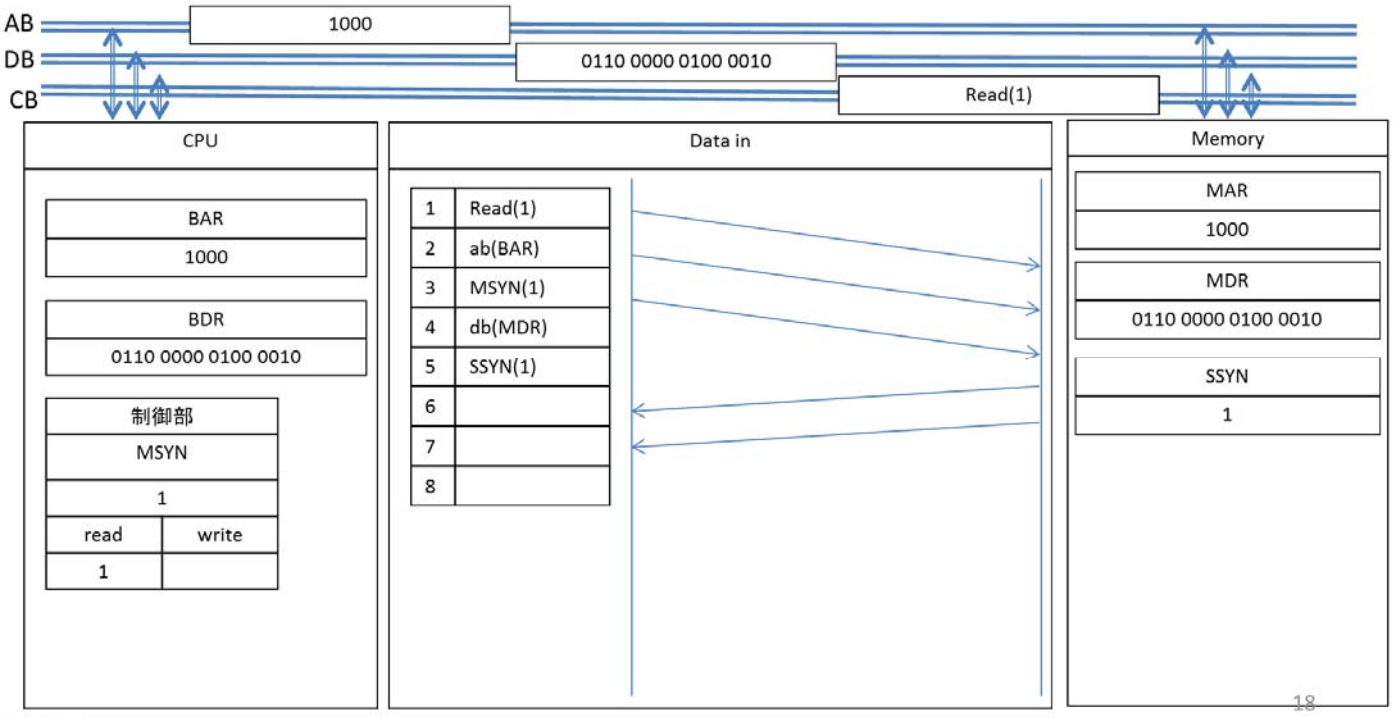


図 3.9. 割り込み命令

デコーダー内部動作

セレクタ回路で命令の解析を行っており，その解析の方法を回路図で提示し，デコーダーの原理を理解してもらう．

ALU 内部動作

算術演算，論理演算，シフト演算などすべての命令での ALU 内部動作を提示し，命令に対して ALU の内部ではどのように演算されているのかを理解してもらう．

4. 計算機アーキテクチャの各項目の実現，今後の課題

現在は提案シミュレータの構成を考えており，レベル 1，レベル 2 のフェッチステージ，デコードステージの提案は実装できる環境にある．レベル 2 の実行ステージでは，各命令に対応したシミュレートが必要なため，今後すべての命令に対応したシミュレータの提案を進める．レベル 3 のフェッチステージでは，割り込み処理の動作の提案は完了しているが，デコーダーの内部動作，ALU の内部動作は，バスレベルでの考案が必要なため，現段階では実装はできない環境である．

5.おわりに

本研究では，計算機アーキテクチャとアセンブリ言語の教育の現状を調査し，シミュレータの要件定義を行った．
今後の課題として，可視化できるレベルをより掘り下げていく．そして，実装を行い学生の講義で使用してもらう．

6.参考文献

[1]PDP-11 Handbook by Digital Equipment Corporation(1979)
[2]法政大学 理工学部 応用情報工学科 計算機アーキテクチャ 講義資料
[3] 法政大学 理工学部 応用情報工学科 計算機ハードウェア 講義資料

アセンブリ言語教育支援教材 Sim AI の設計と実装

小林晴紀†　　和田幸一†
†法政大学大学院

あらまし：本稿は法政大学応用情報工学科のアセンブリ言語教育を支援する CASLⅡシミュレータ「Sim AI」(CASLⅡ Simulator for Applied Infomatics of Hosei University)の設計及び実装について述べる．提案シミュレータは指導者と学習者の双方を支援対象とする．提案シミュレータは指導者の講義解説や学習者の自学自習を支援する上で有効と考えられる計算機内部状態・プログラム動作の可視化機能や，プログラム記述負担を軽減する入力補助機能，OS の仕様に準じた主記憶における不定番地へのプログラムロード機能を有する．

１．はじめに

アセンブリ言語は機械語を人間が読みやすい短いニーモニックで表現した低水準言語である.一般的に，アセンブリ言語の教育はC言語やjava等の高水準言語と比較して難しいとされている．この問題は，アセンブリ言語が機械語に近い単純な命令語で構成されていることに起因する．複雑な命令系が備わっていないため，プログラム作成者はプログラム内で多くの定義をしなくてはならない．また，機械語が計算機内部でどのように処理されるかも把握している必要がある．これは，計算機アーキテクチャの仕組みの理解がなければ不可能である． このようにアセンブリ言語を教育する上では多くの障害がある．

一方で，近年IT分野における教育支援システムの構築が盛んに行われている．この流れの中，計算機アーキテクチャやアセンブリ言語の教材は多数あるにも関わらず，学習環境が充実しているとはいえない．

本研究ではアセンブリ言語教育を支援するためのシステムの開発をした．このシステムでは対象とするアセンブリ言語として CASLⅡ,その仮想計算機として COMETⅡを採用する．CASLⅡは既存のコンピュータの本質の部分を抜き出して実装した仮想計算機 COMETⅡの機械語を記述するためのアセンブリ言語である.COMETⅡは仮想計算機であり実機と比較して，アーキテクチャが単純で分かりやすいという特徴がある．CASLⅡと COMETⅡは基本技術者試験にも採用されている．本研究ではアセンブリ言語教育を支援するCASLⅡシミュレータ「Sim AI」を設計,開発した．シミュレータは講義におけるワークフローの円滑化や学習者の自学自習の便宜を図ることを目標に開発を行う．提案シミュレータは視覚的理解を可能とする計算機内部状態とプログラム動作可視化やユーザの負担を軽減する入力補助機能、不定番地へのプログラムロード機能を実現している．

2 関連研究[1][2][3]

本稿における教育支援システムはアセンブリ言語と計算機アーキテクチャの教育において指導者と学習者のいずれをも支援するためのシステムとする．このシステムがサポートするものとして,学習者のアセンブリプログラミング・学習者同士の情報交換・学習者指導者間の質疑応答，指導者の講義・問題生成・プログラム採点・プログラム評価が挙げられる．

関連研究として，教育用 MIPS プロセッサシミュレータシステム，計算機アーキテクチャ教育支援システム，初等アセンブラプログラミング評価支援システムがある．これらの支援システムの概要を以下に記す．

➢ 教育用 MIPS プロセッサシミュレータシステム[1]

会津大学が開発したプロセッサ内部の動作を可視化するシステム.複数の MIPS マイクロアーキテクチャに対応し，コンピュータ内部の動作や構造を視覚化できる． このシミュレータの課題として，データバスや制御部の定義を変更可能とすることが挙げられている.

➢ 計算機アーキテクチャ教育支援システム[2]

ノイマン型コンピュータ教材 VisuSim を用いた計算機アーキテクチャ教育支援環境. VisuSim は計算機内部の構造・動作の可視化やアセンブリ言語プログラミング支援などを支援する教材である．アプレットとしてもアプリケーションとしても起動可能という特徴を持ち，その特徴を利用したメール機能を組み込むことで学習者指導者間の質疑応答や学習者間の情報交換等の教育支援を円滑にしている.この教材を用いたシステムの課題として前述の基本機能を活かすための補助機能の開発や, LMS との統合が挙げられている.

➢ 初等アセンブラプログラミング評価支援システム[3]

CASL を教材とした授業におけるプログラム評価支援システム．評価対象のプログラムが提示した問題の題意を満たしているかどうかの判定とプログラムに対するアドバイスの作成を行う．このシステムの課題として，冗長命令検出やアドバイス文の精度向上等が挙げられている.

教育支援システムの構成としては１章で列挙した関連研究[1][2][3]を参考とする．完成システムの概要図を次頁に示す.

３．提案シミュレータの設計方針 [4]

３－１．先行調査

シミュレータを新規に開発する上で必要な機能を調査する．はじめに，実際の教育現場で使用されている CASLⅡシミュレータの運用法である．ここでは，例として法政大学のアセンブリ言語講義を挙げ，現状を調査する．もう一つは Eclipse や Visual Studio など他言語の統合開発環境である．アセンブリ言語教育と比較して，高水準言語教育においてデファクトスタンダードとして広く利用されている統合開発環境は数多い．ここでは，アセンブリ言語教育においても有意性を示す統合開発環境の機能及び運用法について調査する．以上 2 つの調査によりシミュレーション使用環境及び求められている機能を明らかにする．最後に，シミュレーション使用環境に基づき既存 CASLⅡシミュレータについて調査・比較を行う．**以下に**比較対象である既存の CASLⅡシミュレータの一覧表を示す.

表１　既存の CASLⅡシミュレータ一覧

1	CaslBuilder	10	CASL & COMET Simulator
2	WCASL-II	11	CCasl II
3	CASL2 Visible Inside	12	CASL2000 for Windows95,98/NT4.0
4	CASL2シミュレータ	13	EduCasl
5	ドリームキャスル	14	Hello CASLII(1ライセンス版)
6	中身が見えるCaslxxxx	15	IPA公式シミュレータ
7	CASL2 cmd	16	InfoCASL version2.0.10
8	CASL2する?	17	Web版キャッスルⅡシミュレータ
9	キャッスルシミュレーター	18	CASL2シミュレータ

既存シミュレータと他言語における統合開発環境の機能・運用法を調査して得られたシミュレーション使用環境及び必要と考えられる機能を表２にまとめる.

表２　先行調査で得られたシミュレーション使用環境

運用法	自学自習，演習，講義解説
主な学習者の特長	プログラムの理解作成に苦労を要する 他分野(計算機アーキテクチャ、C 言語)の知識を有する
指導者	プログラムの視覚的解説
必須機能	計算機内部状態を可視化しつつプログラムをシミュレーションする機能 プログラム入力補助機能

調査の結果，計算機内部状態やプログラム動作可視化機能が効果的なこと，学習者の負荷を軽減する入力補助機能の提供が効率的学習の観点から望ましいなどの結論を得る.

既存シミュレータの中には計算機アーキテクチャとアセンブリ言語の協調学習を目的とするものがあった.(W-CASLⅡ)また，サンプルプログラムを取得・シミュレートするシミュレータもあった.(Hello CASLⅡ)しかし，いずれも可視化シミュレート機能が対象とするアーキテクチャが不十分であり、プログラム入力補助機能を有したものはドリームキャッスルのみであった．特に主記憶の可視化については、格納されているものの説明が不十分で，命令なのか参照している番地なのか分かり辛かった．先行調査の結果，既存シミュレータによって条件を満たすことは難しいと判断する．以上が，新規に CASLⅡシミュレータを開発するに至った要因である.

３－２．設計方針

（１）可視化シミュレート機能

計算機の内部状態を可視化シミュレーションするにも様々な表現方法が考えられる．表現方法を工夫することで指導者の手間を省き，学習者の理解度を高めることが可能となる．例えば，COMETⅡ・CASLⅡではプログラムがロードされる主記憶の領域は不定としている．但し，プログラム中のラベルに対応するアドレス値は OS によって実アドレスに補正される．この一連の仕様は主記憶内部をそれに準じた形（図１）に変形することで表現が可能となる．このような仕様に準じた表現方法を採用する.

```
START    EX
         LD    GR1, DATA1
         LD    GR2, DATA2, GR3
         RET
DATA1    DC    3
DATA2    DC    31
         END
```

実効アドレスはこのように[　]で囲む

０００１	０００１	０００１	０００１
＊＊＊＊	＊＊＊＊	＊＊＊＊	＊＊＊＊(GR1←[****(DATA1)])
０００１	０００１	００１０	００１１
＊＊＊＊	＊＊＊＊	＊＊＊＊	＊＊＊＊（GR2 ← [GR3 + ****(DATA2)]）
１０００	１０００	０００１	０００１
０００１	０００１	０００１	００１０(DATA1)
０００１	０００１	０００１	１１１１(DATA2)

図１　主記憶内部の表現方法例

（２）プログラム入力補助機能

プログラム作成負担を軽減することで効率的学習が期待できる．高級言語の統合開発環境では，シンタックスのハイライト明示や命令記述例の紹介を使用者の入力に即して動的に行うものが存在する．提案シミュレータに取り入れる主な機能例をまとめるとシンタックスハイライト，オートコンプリート，プリティプリンタ，スペルチェッカー等がある．これらの機能はプログラム作成負担を軽減する一方で，学習の妨げとなることも考えられる．自力でプログラムを記述する能力を身につけるために，これらの機能は切替できるようにする．

４．Sim　AI の開発［5］

４－１．シミュレータの概要

提案シミュレータは，アセンブリ言語CASLⅡを記述するためのエディタ・可視化機能を実現するためのプログラム動作表示画面，シミュレーション機能を実現するためのデバッガから構成される．尚，従来シミュレータに見習い，アセンブリ言語と関連性が高い計算機内部の動作や構造をJava Swingを利用して可視化シミュレーションを実現する．また，レジスタ内容の2進, 10進，符号無し10進, 16進，文字での提示も実現した．シミュレータは移植性を考慮してJavaを用いて実現した．

次に提案シミュレータのユーザインターフェース概要を図２に示す．図２は,制御画面であると同時にプログラム動作を可視化する．ユーザインターフェースは上部にファイル操作をはじめとするアクションコマンドが実行されるメニューバーやツールバーがあり，その下にエディタ画面(図 2C)，主記憶表示画面(図 2B)，レジスタ表示画面(図 2A)，インフォメーション画面(図 2D)がある．COMETⅡのシミュレート実行時には画面が制御部やＡＬＵなど COMETⅡの内部情報を表示する画面に切り替わる．

以下，提案シミュレータの特長である計算機内部状態・プログラム動作の可視化機能やユーザ入力補助・主記憶における不定番地へのプログラムロードについて概要を述べる．

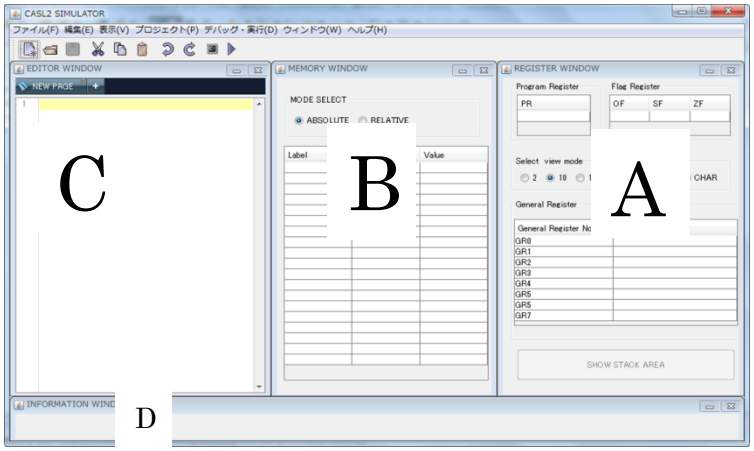


図 2　提案シミュレータの GUI 画面

4－2．ユーザ入力補助機能

コピー，切取，貼付，検索, Undo, Redo 等のエディタとしての基本機能に加えて，プログラムの行番号の表示，オートコンプリート，命令セットの情報表示，スペルチェッカー,シンタックスハイライトを実装する.オートコンプリートに関しては，候補となる命令及び例題コードを表示し，命令を学習者に選択させる．この機能は学習者のコード記述負担を軽減するというメリットがある反面，命令セットの学習面では悪影響となる機能である．そのため, Ctrl + Space というショートカットキーを入力しない限りこの機能を無効状態となるよう設計した．シンタックスハイライトに関しては　命令系，リテラル，コメント，レジスタ，特殊文字によって，ハイライト色を場合わけして施すことによりプログラム構造を理解しやすいものとしている．以下にエディタの機能例を示す.

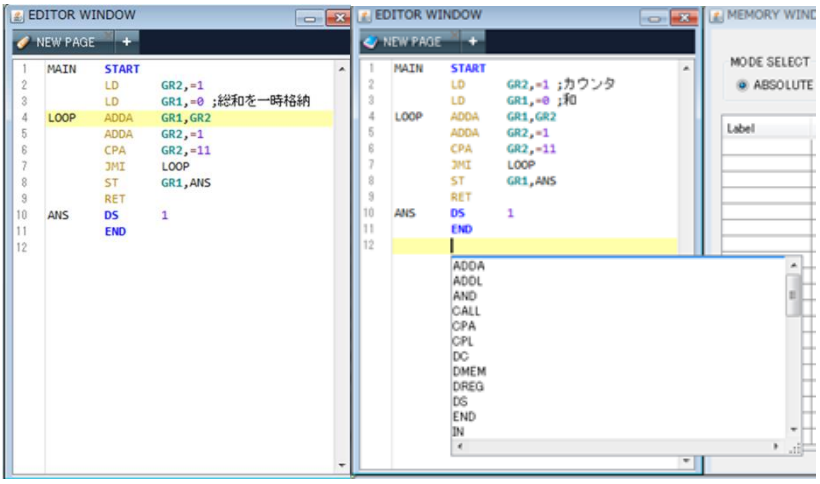


図 3　エディタ画面(左はオートコンプリート無し，右はオートコンプリート有り)

4－3．計算機内部状態・プログラム動作可視化機能

（1）レジスタ表示画面

汎用レジスタとスタックのデータは 2 進数や符号無 10 進数，符号付 10 進数，16 進数の中からユーザが選択した形で提示する．さらに，学習者の理解を助けるため，1 命令実行毎に次に参照されたレジスタを黄色に，更新されたレジスタ内容を赤色に変化させ，使用者の注意を喚起する．また，プログラム

内で文字列またはポインタとしてデータが設定された場合は文字列またはポインタとして提示する．汎用レジスタが指標レジスタとして使われた場合は(GRx + 番地)または(数 + 番地)と注釈する．これは従来の CASLⅡシミュレータにはない新たな機能である．

（2）主記憶表示画面

サポート機能として，二語で構成される命令語(第二語がアセンブラ命令(DC DS)やマクロ命令(IN, OUT, RPUSH, RPOP)，ラベル，リテラルに割り当てられた番地となる命令)に対しては，二語目の番地の内容を注釈する．スタック領域と定数領域，リテラル領域，プログラム領域は，ユーザがみて区別がつくように提示する．尚，番地の内容もレジスタ表示画面と同様の表示形式を選択できるようにする．

（3）シミュレーション機能

プログラム全体を実行した後の値を観察するだけでは，学習者がプログラムの振る舞いを理解する上で不十分な場合がある．その対策として以下の機能を実装した．尚，これらの機能は図 6 上部メニューバーの「デバッグ・実行(D)」から選択可能である．

- トレース実行

  プログラムの実行履歴を別画面で表示する．実行履歴は上から実行順番に従ってソートされる．表示内容はPR, FR, GR, SPの内容・メモリ番地及び内容・命令である.ループ文やサブルーチンの箇所は，ループの繰り返し回数やサブルーチンの実行回数等を横に注釈して表示される．そうすることにより，ループ文やサブルーチンの構造や振る舞いの理解するのを手助けする．

- ステップ実行

  プログラムを1ステップ毎に実行する．ユーザの設定によって繰り返し処理を構成する部分を1ステップごとに実行したり1行ずつ実行したりできるようにする．以下で述べるレジスタ・メモリ内容変更機能と組み合わせることにより，学習者-シミュレータ間のインタラクティブなやり取りを可能とする．また繰り返し処理の回数や初期値を変数と仮定して，レジスタの値等を関数として提示する．

- 手戻り実行

  ステップ実行を行ったプログラムを命令文一行単位で手戻りする．

- ブレークポイントの設定/解除

  ユーザが選んだ任意の行でプログラムを停止する．また，ユーザが指定した範囲内を実行できる.この機能をエラー表示があるときでも可能な限り行えるようにすることでエラーの原因究明の手助けをすることが可能である．

- 停止条件文の追加

  レジスタ・変数・アドレスの値や命令の実行回数を用いて作った停止条件文をプログラムに対して設定する．条件式が真の時や偽の時，式の値が変化した時に実行を中断する．

➢　レジスタ・メモリ内容変更

ステップ実行・手戻り実行・ブレークポイント・停止条件文のようなデバッガによるプログラムの動作停止の際呼び出される．シミュレータ使用者は任意のレジスタ・メモリ内容の値を書き換えることが可能．何も入力しない場合，実行終了となるか，ステップ実行，手戻り実行モードに復帰するか選択可能．尚，ここで書き換えた値はプログラムの記述内容を書き換えるものではないので注意が必要である．

➢　命令の実行回数表示

各行やユーザが指定した一定範囲の各命令実行回数を表示する．分岐命令やサブルーチンコールを含むプログラムの理解の手助けをする．

➢　クロスリファレンス

クロスリファレンステーブルを出力する．プログラムで定義したラベルやサブルーチンの定義された番地や参照している番地を表にまとめて出力する．整列順序としては定義した名前のアルファベット順や定義された番地の若い順から選択可能．

４－３．ユーザ入力補助機能

コピー，切取，貼付，検索，Undo，Redo 等のエディタとしての基本機能に加えて，プログラムの行番号の表示，オートコンプリート，命令セットの情報表示，スペルチェッカー,シンタックスハイライトを実装する.オートコンプリートに関しては，候補となる命令及び例題コードを表示し，命令を学習者に選択させる．この機能は学習者のコード記述負担を軽減するというメリットがある反面，命令セットの学習面では悪影響となる機能である．そのため，Ctrl + Space というショートカットキーを入力しない限りこの機能を無効状態となるよう設計した．シンタックスハイライトに関しては　命令系，リテラル，コメント，レジスタ，特殊文字によって，ハイライト色を場合わけして施すことによりプログラム構造を理解しやすいものとしている．以下にエディタの機能例を示す．

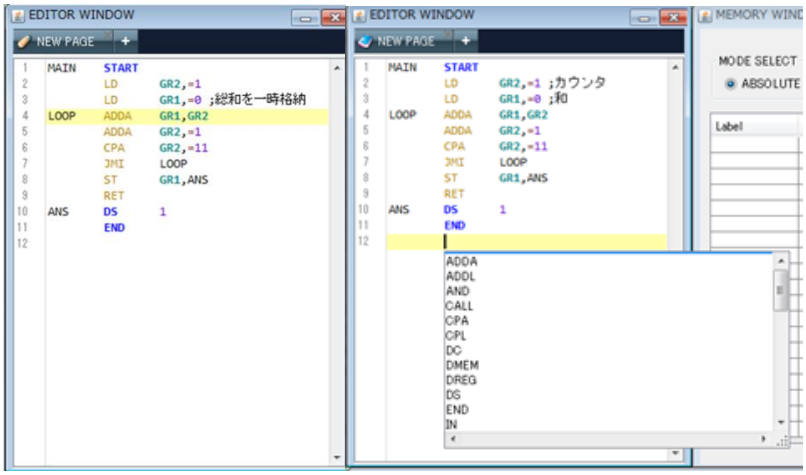


図３　エディタ画面(左はオートコンプリート無し，右はオートコンプリート有り)

４－４．主記憶における不定番地へのプログラムロード機能

COMETⅡ･CASLⅡではプログラムがロードされる主記憶の領域は不定としている.但し，プログラム中のラベルに対応するアドレス値は OS によって実アドレスに補正される．このような番地を本稿では不定番地と呼ぶ．主記憶のプログラムロード及び GUI 画面ではこの要件を実現している．ロードされる主記憶の領域が不定の時，機械語のビットパターンに変化が生じる可能性がある箇所は，オペランドであり，命令コードは変化しない． また，オペランドの中でもビットパターンは変化しない箇所は２つある（**Ⅰ．レジスタ部(第一，第二オペランド)， Ⅱ．実効アドレスの絶対番地指定箇所**）．よって，Ⅰ，Ⅱとそれ以外のオペランドを分けて提示する． はじめに，初期画面として番地の中身を提示する．左部の番号は主記憶にロードされる順番を表している．提示する際は，その内容をⅠⅡのオペランド，ⅠⅡ以外，命令コードの 3 色に分けて提示する．オペランドにおいて，不定となるビットパターンは****と提示する.そして，****には説明を加える.例(＊＊＊＊[LABEL]) 絶対番地指定された箇所に対応するビットパターンは自明なのでそのまま提示する.(例：#1FFF 番地に２を格納 [1FFF]←2)

５．今後の課題

未実装のシミュレーション機能と COMETⅡにおけるプログラム動作を可視化する機能を実装する．そして 提案シミュレータを実際の教育現場で運用し評価を行う．また，他の教育システムへの適用も考えられる．

参考文献

[1]. 西牧悠二，北道敦司，宮崎敏明,“内部動作を視覚化した教育用 MIPS プロセッサシミュレータシステムの開発”電子情報通信学会論文誌 Vol. J96 – D No.10 (2013)

[2]. 今井慈郎，金子敬一，中川正樹,“計算機アーキテクチャ教育支援システムの開発と強調学習への適用”電子情報通信学会論文誌 D Vol. J91- D No.2 pp.188-199(2008)

[3]. 渡辺博芳，荒井正之，武井惠雄 “事例に基づく初等アセンブラプログラミング評価支援システム”情報処理学会論文誌 Vol.42 No.1(2001)

[4]. 小林，上村，和田 アセンブリ言語教育のための CASLⅡシミュレータ調査と比較，第 9 回情報科学ワークショップ(WTCS2013)予稿集, pp.31-34 , (2013 - 09) .

[5]. 小林，林，和田 アセンブリ教育支援システムにおける学習用 CASLⅡシミュレータの提案 - CASLⅡ&COMETⅡシミュレータ仕様書，法政大学理工学部応用情報工学科計算機科学研究室 テクニカルレポート TR14 – 6, 1 – 16, (2014 - 06) .

WebRTC システムに適したテストフレームワークの開発

中川路克之 大下福仁 角川裕次 増澤利光
大阪大学大学院情報科学研究科

概要

ウェブブラウザ（以下，ブラウザ）間リアルタイムコミュニケーションのための標準仕様 **WebRTC**[1] が出現したことでブラウザ間 Peer-to-Peer(P2P) 通信が可能となったが，WebRTC を使ったシステム (以下、WebRTC システム) のテスト環境は未だ整備されておらず，複雑なシステムの構築は困難である．

本稿では，現在開発している「WebRTC システムに適したテストフレームワーク」について紹介する．本研究では P2P システムの各ノードに対応するコンテナ型仮想マシン (Docker) を立ち上げ，VM 内のブラウザをリモート制御し，ノード数の増減を含めたテストシナリオを，P2P システム全体に対して実行するフレームワークを開発している． 本フレームワークにより煩雑だった WebRTC システムのテストを効率的に行えるようになることが期待できる．

1 研究背景

1.1 WebRTC

標準化団体 W3C[1]/IETF[2]ではウェブアプリケーションにおけるリアルタイムコミュニケーションを実現するための標準仕様 WebRTC の策定を進めている．WebRTC で可能になるブラウザ間 P2P 通信は，ブラウザ上で動く P2P グリッドコンピューティング [2] など，リアルタイムコミュニケーション以外の用途にも活用することができる．

ただし，現状では WebRTC システム向けのテスト環境が整備されていないため，複雑なシステムの開発は難しい．そこで，本研究では，WebRTC システム向けのテストフレームワークを開発している．

1.2 コンテナ型仮想マシン Docker

Docker は，アプリケーションのデプロイ自動化のためのコンテナ型仮想マシン (VM) である．本フレームワークでは，Docker を用いて WebRTC システムの各ノードに対応する VM を立ち上げる．VM 内にはブラウザ Firefox と Selenium[3]等が格納されている．仮想マシン上のブラウザ (Firefox) を実行環境とするため実運用時と近い環境でテストを行うことができる．コンテナ型仮想マシンは VM の立ち上げが高速で，ノード数が動的に変化する P2P システムのテストに適している．

2 WebRTC テストフレームワーク

WebRTC システムの振る舞いを確認するために，各マシンのブラウザを手動制御して，目視確認を行うのは，マシン数が増えると現実的ではない．しかし，既存のブラウザ向けテストフレームワークはブラウザ単体でのシステムの

[1] W3C - World Wide Web Consortium
[2] IETF - Internet Engineering Task Force
[3] ブラウザのリモート制御ソフト

挙動を対象としており，複数のブラウザが連携して動く WebRTC システムには適さない．

本フレームワークでは，テストクライアントが WebRTC システムを構成する各ノードを集中管理し，各ノードから情報を適時収集する．本フレームワークの特徴は，（P2P システムの特徴的な課題である [3]）ノードの参加・離脱をシナリオファイルとして記述できるところにある．シナリオファイルの入れ替えにより，同じテストを様々なシナリオに対して実行できる．

3　全体の構成

テストサーバと，テストクライアントにより構成される (図 1)．テストクライアントはポートフォワードによってテストサーバ内の各 VM と通信し，Selenium でブラウザを制御する．なお，各 VM にはリモートデスクトップソフト (VNC) をデバッグのために入れてある．

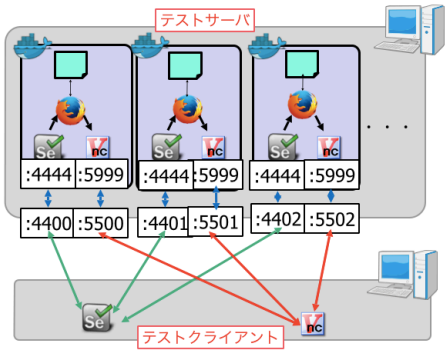


図 1　テストフレームワークの構成

4　完全自動制御の実験

スクリプトによる WebRTC システムの完全自動制御の実験を行った．指定した数 (4) の VM を立ち上げ，各 VM 内のブラウザを制御して WebRTC システムを実行させ，各 VM が連携して動くことをリモートデスクトップソフトにより目視で確認した (図 2).WebRTC システムには Web P2P Grid[2] を用いた．

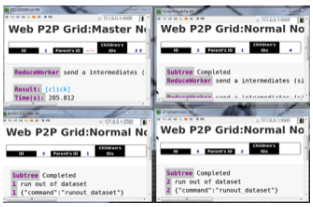


図 2　自動実験の結果

5　今後の展望

本フレームワークは現時点で未完成であり，今後，振る舞いの自動テスト部分を実装し，フレームワークとして実用的なものに仕上げるのが当面の目標となる．その後の展開としては，複数の物理マシンに VM が分散している場合の分散テスト機能を検討している．

6　おわりに

本稿では，現在開発中の WebRTC 向けテストフレームワークについて紹介した．フレームワークの概要について述べた後，現時点の成果である完全自動制御について述べ，最後に今後の展望について述べた．

参考文献

[1] A. Bergkvist et al. , “WebRTC 1.0: Real-time Communication Between Browsers,” W3C, 2012-08-21, `http://www.w3.org/TR/webrtc/` [09 Feb 2013].

[2] 中川路克之，大下福仁，角川裕次，増澤利光, ”ウェブブラウザ上で動作する Peer-to-peer 型 MapReduce フレームワーク”, 電子情報通信学会 2014 年総会 , 2014 年 3 月.

[3] De Almeida et al. , “Testing peer-to-peer systems.” Empirical Software Engineering 15.4 (2010): 346-379.