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 first 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 configurations 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 configurations. 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 efficient 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 finite 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 define this problem as the *g-partial gathering problem*. Clearly, if $k/2 < g \leq k$ holds, the g-partial gathering problem is equal to the ordinary gathering problem. If $1 \leq g \leq k/2$ 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 configurations

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 configurations 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 configurations. 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 first 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 $g \geq 2$. This means the first 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 defined as follows.

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

We define 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 define 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 $i-1$ 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 define 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 first model, we consider agents that are distinct (i.e., agents have distinct IDs) and execute a deterministic algorithm. We model an agent as a finite automaton $(S, \delta, s_{initial}, s_{final})$. The first element S is the set of the agent a_h's all states, which includes initial state $s_{initial}$ and final state s_{final}. After an agent changes its state to s_{final}, the agent terminates the algorithm. The second element δ is the state transition function. Since we treat deterministic algorithms, δ is described as δ: $S \times W \to S \times W \times M$, 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, δ, $s_{initial}$ 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 first model except for state transition function δ. Since we treat randomized algorithms, δ is described as δ: $S \times W \times R \to S \times W \times M$, 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 first model. We assume that each agent knows the number of agents. Note that all the agents are modeled by the same machine.

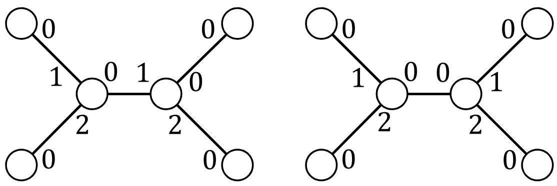
In unidirectional ring network model, we assume that agents move instantaneously, that is, agents always exist at nodes (do not exist at links). Moreover, we assume that each agent cannot detect whether other agents exist at the current node or not.

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 $\{0, 1, \ldots, d_{v_j} - 1\}$. 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 $\{v_i, v_{i+1}\} \in L$ $(0 \le i < k)$ and $v_i \ne v_j$ if $i \ne j$. 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 define 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

Definition 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 finite automaton $(S, \delta, s_{initial}, s_{final})$. The first element S is the set of all states of agents, which includes initial state $s_{initial}$ and final state s_{final}. When an agent changes its state to s_{final}, the agent terminates the algorithm. The second element δ is the state transition function. In the weak multiplicity detection and non-token model, δ is described as $\delta : S \times M_T \times EX_A \to S \times M_T$. In the definition, set $M_T = \{\perp, 0, 1, \ldots, \Delta - 1\}$ represents the agent's movement, where Δ is the maximum degree of the tree. In the left side of δ, 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 first activation). In the right side of δ, 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, δ is described as $\delta : S \times M_T \times N \to S \times M_T$. In the definition, N represents the number of other agents at the current node. In the weak multiplicity detection and removable-token model, δ is described as $\delta : S \times M_T \times EX_A \times EX_T \to S \times EX_T \times M_T$. In the definition, in the left side of δ, $EX_T = \{0, 1\}$ 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 δ, $EX_T = \{0, 1\}$ 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) *configuration* c is defined as a product of states of agents, states of nodes (whiteboards), and locations of agents. In initial configuration 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 configuration. If there exists an agent on node v_j in the initial configuration, 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 configuration c is defined as a product of states of agents and locations of agents. In the removable-token model, configuration c is defined as a product of states of agents, states of nodes (tokens), and locations of agents. Moreover, in the initial configuration 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 configuration c_0.

Let A_i be an arbitrary non-empty set of agents. When configuration c_i changes to c_{i+1} by a step of every agent in A_i, we denote the transition by $c_i \xrightarrow{A_i} c_{i+1}$. 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 configurations $E = c_0, c_1, \ldots$ satisfies $c_i \xrightarrow{A_i} c_{i+1}$ $(i \geq 0)$, E is called an *execution* starting from c_0. Execution E is infinite, or ends in final configuration 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 define the g-partial gathering problem as follows.

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

- *Execution E is finite.*
- *In the final configuration, 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 $g \geq 2$.*

Proof. We consider an initial configuration 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 final configuration, and let $x = |V'|$. Since at least g agents meet at v_j for any $v_j \in V'$, we have $k \geq gx$.

For each $v_j \in V'$, we define 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 $(i-1)n/k$. So, we have

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

Therefore, the total moves is at least

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

Since $k \geq gx$ holds, we have

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

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 first 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 first 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 first part: leader election

The aim of the first 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 $g-1$ 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

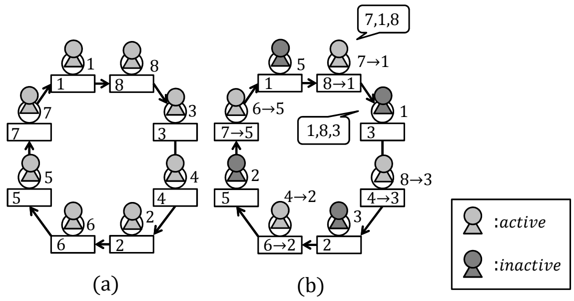


Figure 2: An example for a g-partial gathering problem($k = 9, g = 3$)

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 satisfied: 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 configuration in Fig. 2 (a). In the figures, 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 configuration is changed to the one in Fig. 2 (b). In this configuration, 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
$a_h.phase = 1$ and $a_h.id_1 = a_h.id$
$(v_j.phase, v_j.id) = (a_h.phase, a_h.id_1)$
$BasicAction()$
$a_h.id_2 = v_j.id$
$BasicAction()$
$a_h.id_3 = v_j.id$
if $a_h.id_2 \geq min(a_h.id_1, a_h.id_3)$ **then**
    $v_j.inactive = true$ and become inactive
else
    if $a_h.phase = \lceil \log g \rceil$ **then**
        change its state to a leader state
    else
        $a_h.phase = a_h.phase + 1$
        $a_h.id_1 = a_h.id_2$
    end if
    return to step 2
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 configuration 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 configuration 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 configuration 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,

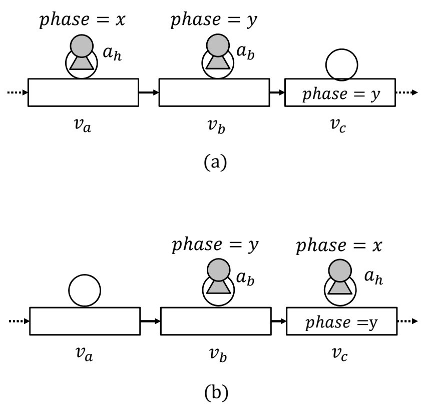


Figure 3: The first example of an agent that passes other agents

---

Algorithm 2 Procedure *BasicAction*() for a_h

1: move to the forward node
2: **while** $(v_j.initial = false) \vee (v_j.inactive = true \wedge a_h.phase \neq v_j.phase)$ **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**

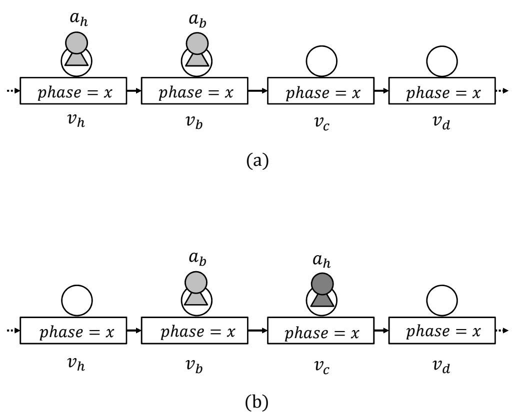


Figure 4: The second example of an agent that passes other agents

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 configuration satisfies the following properties.*

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

Proof. At first, 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

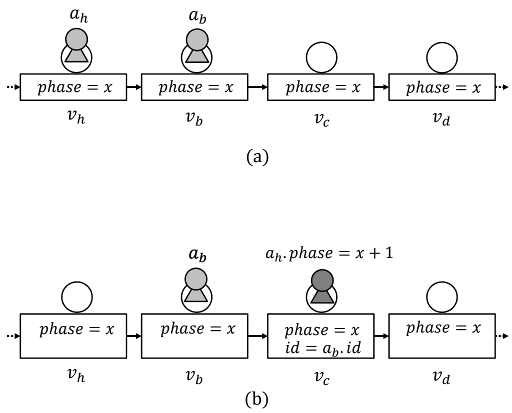


Figure 5: The third example of an agent that passes other agents

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 first, 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 $(2^k - 1) + 1 + (2^k - 1) = 2^{k+1} - 1$. Hence, we can show that 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 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 first part. Let leader nodes (resp., inactive nodes) be the nodes where agents become leaders (resp., inactive agents) in the first 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 configuration in Fig. 6 (a). In this configuration, 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 configuration in Fig. 6 (c).

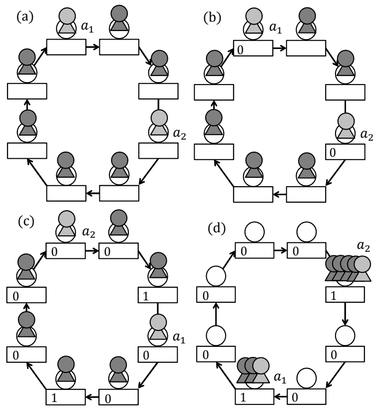


Figure 6: The realization of partial gathering($g = 3$)

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 first node where 1 is written to the whiteboard. For example, after the configuration in Fig. 6 (c), each non-leader agent moves to the node where 1 is written to the whiteboard and the system reaches the configuration 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 first, 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.)

```
v_j.isGather = 0 and a_h.count = a_h.count + 1
move to the forward node
while v_j.isGather = ⊥ do
    while v_j.initial = false do
        move to the forward node
    end while
    if (v_j.inactive = false) ∧ (v_j.isGather = ⊥) then
        wait until v_j.inactive = true or v_j.isGather ≠ ⊥
    end if
    if v_j.inactive = true then
        if a_h.count = 0 then
            v_j.isGather = 1
        else
            v_j.isGather = 0
        end if
        // an inactive agent at v_j changes to a moving state
        a_h.count = (a_h.count + 1) mod g
        move to the forward node
    end if
end while
change to a moving state
```

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

```
wait until v_j.isGather ≠ ⊥
change to a moving state
```

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

```
while v_j.isGather ≠ 1 do
    move to the forward node
    if (v_j.initial = true) ∧ (v_j.isGather = ⊥) then
        wait until v_j.isGather ≠ ⊥
    end if
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 first, 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.

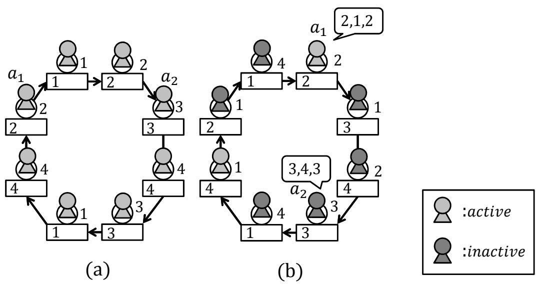


Figure 7: An example that some agent observes the same random IDs

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 first 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 first 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 configuration 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 $a_1.id_2 < a_1.id_1 = a_1.id_3$ satisfies. 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 configuration. 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 define *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 configuration, 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

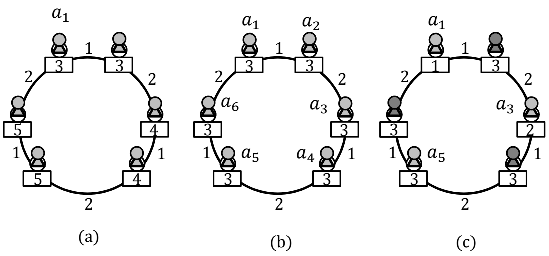


Figure 8: The behavior of semi-leaders

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-leadersInfo*. This pair is used to decide if a semi-leader a_h becomes a leader or inactive after moving around the ring. We define $pair_i^h$ 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-leadersInfo*. Let a_h.*semi-leadersInfo* $= (pair_1^h, pair_2^h, \ldots, pair_t^h)$, where t implies the number of semi-leaders. Then, we define $info_{min}$ as the lexicographically minimum array among $\{a_h.\textit{semi-leadersInfo} | a_h \text{ is a semi-leader }\}$. For array $info = (pair_1, pair, \ldots, pair_t)$, we define $shift(info, x) = (pair_x, pair_{1+x}, \ldots,\ pair_t,\ pair_1\ , \ldots, pair_{x-1})$. If $info = shift(info, x)$ holds for some x such that $0 < x < t$, we say $info$ is periodic. If $info$ is periodic, we define the period of info as $period = \min\{x > 0 | info = shift(info, x)\}$. If a_h.*semi-leadersInfo* is not periodic, there exists exactly one semi-leader $a_{h'}$ that $a_{h'}$.*semi-leadersInfo* $= info_{min}$. Then, a_h becomes a leader and the other semi-leaders become inactive. For example, consider the configuration 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-leadersInfo* $= ((3,1),(3,2),(4,1),(4,2),(5,1),(5,2))$. Since a_1.*semi-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-leadersInfo* $\neq info_{min}$ holds.

If a_h.*semi-leadersInfo* is periodic, there exist several semi-leaders a_h that a_h.*semi-leadersInfo* $= info_{min}$ holds, and we define A_{semi} as the set of such agents. In this case, each semi-leader a_i that *semi-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 $|A_{semi}| \leq \lfloor k/g \rfloor$ 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-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-leadersInfo* $= info_{min}$ holds. For example, let us consider the condiguration like Fig. 8(b). In this figure, $k = 15$ holds. Agents a_1, a_3, and a_5 obtain *semi-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-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 $|A_{semi}| \leq \lfloor k/g \rfloor = 3$ 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 configuration 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-flag* and *leader-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

$a_h.phase = 1$
$a_h.id_1 = random(3 \log k)$
$v_j.phase = a_h.phase$
$v_j.id = a_h.id_1$
$BasicAction()$
if $v_j.tour = true$ **then**
    $a_h.semiObserve = true$
end if
$a_h.id_2 = v_j.id$
$BasicAction()$
if $v_j.tour = true$ **then**
    $a_h.semiObserve = true$
end if
$a_h.id_3 = v_j.id$
if $a_h.semiObserve = true$ **then**
    change its state to an inactive state
end if
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**
    change its state to a semi-leader
end if
if $a_h.id_2 \geq min(a_h.id_1, a_h.id_3)$ **then**
    $v_j.inactive = true$ and become inactive
else
    if $a_h.phase = \lceil \log g \rceil$ **then**
        $leader\text{-}flag = true$
        change its state to a leader state
    else
        $a_h.phase = a_h.phase + 1$
    end if
    return to step 2
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 $g-1$ 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 first half behavior of semi-leader agent a_h (v_j is the current node of a_h)

if $tour\text{-}frag = true$ **then**
  change its state to an inactive state
end if
$semi\text{-}leader\text{-}flag = true$
$a_h.semiPhase = 1$
$v_j.semiPhase = a_h.semiPhase$
$a_h.x = 0$
while $a_h.agentCount \neq k$ **do**
  move to the forward node
  while $v_j.initial = false$ **do**
    move to the forward node
  end while
  $a_h.agentCount = a_h.agentCount + 1$
  if $leader\text{-}frag = true$ **then**
    $a_h.leaderObserve = true$
  end if
  if $semi\text{-}leader\text{-}flag = true$ **then**
    if $a_h.semiPhase \neq v_j.semiPhase$ **then**
      wait until $a_h.semiPhase = v_j.semiPhase$
    end if
    $a_h.semi\text{-}leadersInfo[a_h.x] = (v_j.id, a_h.N_{tour})$
    $a_h.N_{tour} = 0$
    $a_h.x = a_h.x + 1$
  end if
  if $v_j.tour = false$ **then**
    $v_j.tour = ture$
  end if
  $a_h.N_{tour} = a_h.N_{tour} + 1$
end while

lemma 3.4 *Algorithm 7 eventually terminates, and the configuration satisfies the following properties.*

- *There exists at least one leader agent.*
- *There exist at most $\lfloor k/g \rfloor$ leader agents.*
- *There exist at least $g-1$ 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 $\{shift(a_h.semi\text{-}leaderInfo[\,],x) | 0 \le x \le a_h.x - 1\}$.
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** $|A_{semi}| \le \lfloor k/g \rfloor$ **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 satisfied. 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 $|A_{semi}| \le \lfloor k/g \rfloor$ 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 $g - 1$ 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 $(k - \lfloor k/g \rfloor) \div (\lfloor k/g \rfloor) \ge g - 1$. Thus, there exist at least $g - 1$ 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 finish 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 finish 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 satisfied, 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 $g-1$ 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 first, we show that there exist unsolvable initial configurations 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 configuration.

3.3.1 Existence of Unsolvable Initial Configurations

To explain unsolvable initial configurations, we define distance sequence of a configuration. For configuration c, we define 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 define distance sequence of configuration c as the lexicographically minimum sequence among $\{D_h(c)|a_h \in A\}$. We denote distance sequence of configuration c by $D(c)$. For sequence $D = (d_0, d_1, \ldots, d_{k-1})$, we define $shift(D, x) = (d_x, d_{1+x}, \ldots, d_{k-1}, d_0, d_1, \ldots, d_{x-1})$. If $D = shift(D, x)$ holds for some x such that $0 < x < k$, we say D is periodic. If D is periodic, we define the period of D as $period = \min\{x > 0|D = shift(D, x)\}$.

Theorem 3.3 *Let c_0 be an initial configuration. 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 $(0 \le j \le period - 1)$ 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 configuration. 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 final configuration. 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 configurations, 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 configuration 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 $period \ge g$ 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 identifies 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 $period(\ge g)$ 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 configuration 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 $\{shift(a_h.D, x) | 0 \leq x \leq k-1\}$.
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: $a_h.x = \min\{x \leq 0 | shift(a_h.D, x) = D_{min}\}$
18: move to the forward node $\sum_{i=0}^{a_h.x'-1} a_h.D[i]$ times

Proof. At first, 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 $period \geq g$ 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 $period(\geq g)$ 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 first 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 $g \geq 5$ 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 $3g-1$ agents are placed symmetrically in the initial configuration 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 $2g \leq k \leq 3g-1$ 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 configuration for a changes. Concretely, there are two cases. The first 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 configuration 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 define c_t as the configuration 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 $N_1 \geq N_2 \geq \cdots \geq N_t$ 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 configuration 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 first agent that enters a waiting state at v_j, when a_2^j enters a waiting state at v_j, the configuration 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 configurations 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 $N_2 \geq 3$ 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 $k = 3g - 1$ 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 configuration 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 configuration 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 $k = 3g - 1$ and $2g \leq k$ holds, at most $g - 1$ 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 define c'_t as the configuration 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 first 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.

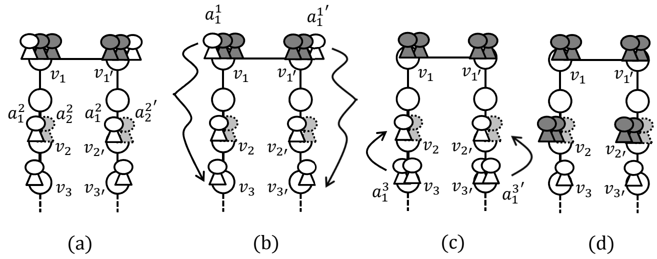


Figure 9: An example of the pattern 3

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 configuration for a_k^j at v_j with agents a_1^j and a_i^j is equal to the configuration 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 $N_1 \geq 3$ 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 configuration 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 figure, 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 configuration 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 figure, 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 configuration 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 first 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 $3 \leq N_1 \leq (g+1)/2$, 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 configuration 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 $2N_1 - 2 \leq g - 1$ since $N_1 \leq (g+1)/2$ 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 configuration is similarly to the pattern 2 and agents cannot solve the g-partial gathering problem.

⟨pattern 5: for the case that $(g+3)/2 \leq N_1 \leq g$, and $N_2 = 1$ hold⟩
In this case, let us the consider the initial configuration 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 $N_1 - 1 \leq g - 1$ since $N_1 \leq g$ 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 configuration, firstly 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 configuration c_u' that some agent a' enters a waiting state at v_j, and we define E_u' as an execution from c_t' to c_u'. In the figure, 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)).

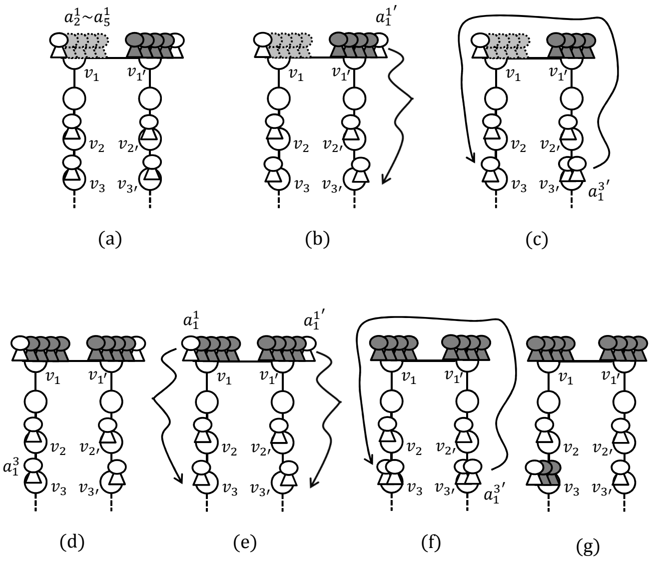


Figure 10: An example of the pattern 5

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 configurations 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 figure, 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 $3 \leq N_1 \leq g-1$, 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 configuration 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 configuration 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 configuration 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 configuration 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 configuration 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 figure, 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 finite 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,

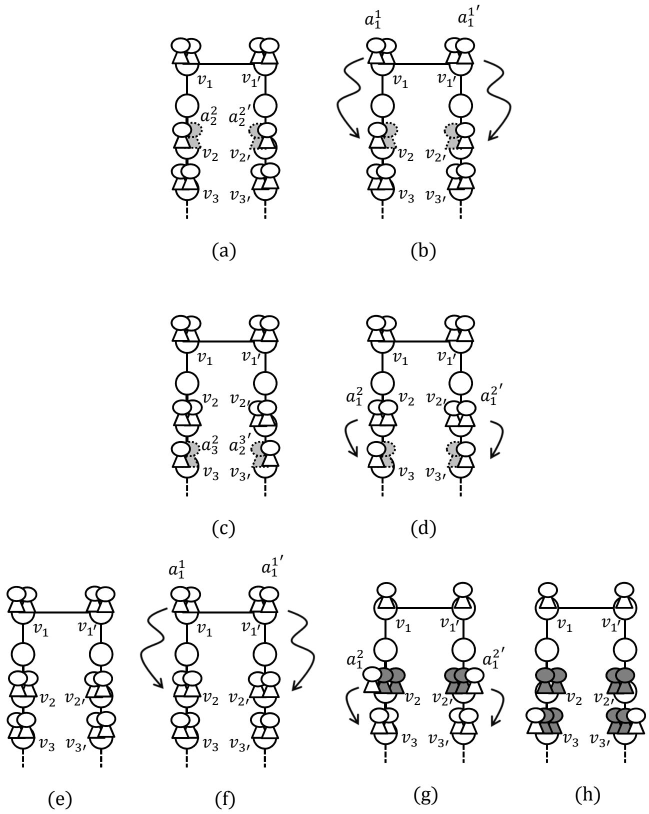


Figure 11: An example of the pattern 8

agents a_2^2 and a ($a_2^{2'}$ and a') never leave v_2 ($v_{2'}$), and we call this configuration c_u. In the figure, 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 configuration c_0'' that agents a_2^3 and $a_2^{3'}$ are eliminated. Then from Lemma 4.3, there exists an configuration 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 configuration 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 figure, 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 finite 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 figure, 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.

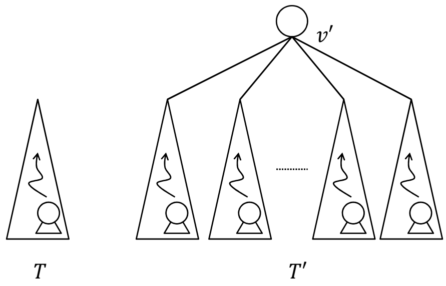


Figure 12: Figures of T and T'

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 configuration 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 firstly 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 configuration 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 $(1 \leq j \leq k)$ 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 $(1 \leq j \leq k)$ be the node in T^j corresponding to v_w in T. Consider the configuration 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 $(1 \leq j \leq k)$ 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 $g \leq k/2$. 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 $2(n-1)$ 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 first 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 first, 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 first 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 $k \geq 2g$ holds, agents can solve the g-partial gathering problem.

Next, we analyze the total moves to solve the g-partial gathering problem. At first, agents perform the basic walk and record the topology of the tree. This requires at most $2(n-1)$ 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 configuration, each agent leaves a token at the initial node. We define *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 $2(n-1)$ times, then each communication link is passed twice and a_h returns to v_0. Thus, when a_h visits nodes $v_1, v_2, \ldots, v_{2(n-1)}$ in this order, then we consider that a_h moves in the unidirectional ring network with $2(n-1)$ nodes. Later, we call this ring *the virtual ring*. In the virtual ring, we define 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-1} in the virtual ring, agents also use p as the port number at (v_{j-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 define 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 define that a_h visits a token node in the virtual ring if it visits v_j through the port $(d_{v_j}-1)$. 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 definition, 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 first 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 first 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.

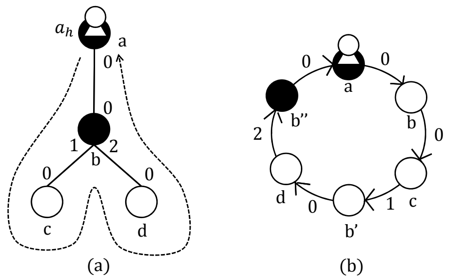


Figure 13: An example of the basic walk

The aim of the first 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 $p_1, p_2, \dots p_l$, 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 $p_1, p_2, \dots p_l$ 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 configuration like Fig. 14(a). In the figure, 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,

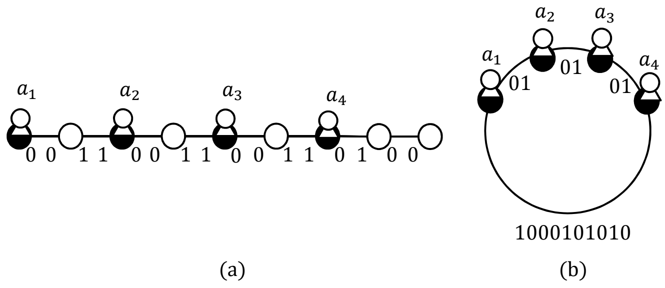


Figure 14: An example that agents observe the same port sequence

---

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 define $(t[1..q])^k$ 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, $2(n-1)$. Then, we use the following theorem.

Theorem 4.4 *Let T be a tree of length at least $q \geq 1$. Assume that $t[1..q]$ is not periodic and $t[1..kq] = (t[1..q])^k$ for some $k \geq 3$. 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
$move = 0$
leave v_j through the port 0
// arrive at the forward node
let p be the port number through which a_h visits v_j
$port[move] = p$
$move = move + 1$
while (there does not exist a token) $\vee$
($p \neq d_{v_j} - 1$) $\vee$ (there exists another agent) **do**
  leave v_j through the port $(p + 1) \mod d_{v_j}$
  // arrive at the forward node
  let p be the port number through which a_h visits v_j
  $port[move] = p$
  $move = move + 1$
end while
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 $k' \geq 3$ 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 satisfies 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 first, 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 satisfies $a_h.id_2 < \min(a_h.id_1, a_h.id_3)$, 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 satisfies $a_h.id_2 = a_h.id_3 < a_h.id_1$. 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 first, 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 configuration 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 configuration 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

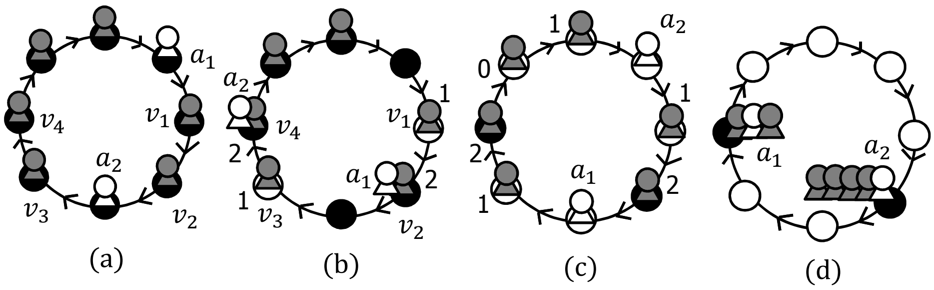


Figure 15: Partial gathering for removable-token model for the case $g = 3$ (a_1 and a_2 are leaders, and black nodes are token nodes)

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
NextToken()
while there exists another agent at v_j **do**
  //this is an inactive node
  $tCount = (tCount + 1) \mod g$
  if $tCount \neq g - 1$ **then**
    remove a token at v_j
    //an inactive agent at v_j changes its state to a moving state
  end if
  NextToken()
end while
remove a token at v_j
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 first, 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 *final 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 final node. Therefore, agents solve the g-partial gathering problem.

In the following, we analyze the total moves required for the algorithms. At first, 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 final node. We assume that some moving agent a_h goes to final 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 final node v_i^f and its forward final 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 final 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 final node v_i^f and v_{i+1}^f, and the total moves required for each moving (inactive) agent to move to the nearest final 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 configurations 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 infinite 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 finding 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.