Title	A Study on Partial Gathering and Uniform Deployment of Mobile Agents in Distributed Systems
Author(s)	柴田, 将拡
Citation	大阪大学, 2017, 博士論文
Version Type	VoR
URL	https://doi.org/10.18910/61860
rights	
Note	

A Study on
Partial Gathering and Uniform Deployment
of Mobile Agents in Distributed Systems

Submitted to
Graduate School of Information Science and Technology
Osaka University

January 2017

Masahiro SHIBATA

List of Related Publications

Journal Papers

1. Masahiro Shibata, Shinji Kawai, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa, "Partial gathering of mobile agents in asynchronous unidirectional rings", *Theoretical Computer Science*, Elsevier, Vol. 617, pp 1-11, 2016.

Conference Papers

2. Masahiro Shibata, Shinji Kawai, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa, "Algorithms for partial gathering of mobile agents in asynchronous unidirectional rings", *Proceedings of the 16th International Conference on Principles of Distributed Systems*, LNCS 7702, pp. 254-268, Rome Italy, Dec. 2012.

3. Masahiro Shibata, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa, "Move-optimal partial gathering of mobile agents in asynchronous trees", *Proceedings of the 21st International Colloquium on Structural Information and Communication Complexity*, LNCS 8576, pp. 327-342, Gifu Japan, July 2014.

4. Masahiro Shibata, Daisuke Nakamura, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa, "An algorithm for partial gathering of mobile agents in arbitrary networks" *Workshop on Distributed Robotic Swarms*, Tokyo Japan, Oct. 2015.

5. Masahiro Shibata, Toshiya Mega, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa, "Uniform deployment of mobile agents in asynchronous rings", *Proceedings of the 29th ACM Symposium on Principles of Distributed Computing*, pp. 415-424, Chicago America, July, 2016.

Technical Reports

6. Masahiro Shibata, Shinji Kawai, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa, "Algorithms for partial rendezvous of mobile agents in asynchronous rings", *Technical Report of IEICE*, COMP2012-9, Vol. 112, No. 24, pp. 17-24, May 2012.

7. Masahiro Shibata, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa, "An algorithm for uniform deployment of mobile agent in asynchronous rings", *Technical Report of IEICE*, COMP2015-11, Vol. 115, No. 84, pp. 107-114, June 2015.

List of Unrelated Publications

Technical Reports

8. Jun Ri, Masahiro Shibata, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa, "Algorithms for group gossiping of mobile agents", *Technical Report of IEICE*, COMP2014-24, Vol. 114, No. 199, pp. 61-68, Sep. 2014.

9. Tsuyoshi Goto, Masahiro Shibata, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa, "Move-Efficient Fault-Tolerant Simulation of Message-Passing Algorithms by Mobile Agents", *Technical Report of IEICE*, COMP2016-3, Vol. 116, No. 17, April 2016.

Abstract

A *distributed system* consists of autonomous computers (*nodes*) and communication links. In recent years, distributed systems have become large and design of distributed systems has become complicated. As a promising design paradigm of distributed systems, (mobile) agent systems have attracted a lot of attention. Agents can traverse the system, collect information and execute tasks on nodes. Hence, we can encapsulate data and algorithms in agents, which simplifies design of distributed systems. Actually agent systems have many applications such as network exploration, network management, electronic commerce and so on.

The *total gathering problem* (usually it is simply called the gathering problem) is a fundamental and deeply investigated problem for coordination of agents. This problem requires all the agents to meet at a single node. By meeting at a single node, agents can share information or synchronize their behavior. Hence, after the gathering agents can determine their behavior so that they can execute tasks collaboratively and efficiently.

Solving the total gathering problem implies completely symmetry breaking when the initial locations of agents have symmetry. It is known that the symmetry breaking is difficult and sometimes impossible. Due to its difficulty, there are two problems about the total gathering problem. The first is about the total moves, that is, agents require more total moves to solve the total gathering problem, which causes high network loads. The second is about the solvability, that is, if agents do not have distinct IDs, they cannot solve the total gathering problem from several initial configurations.

In this dissertation, we introduce two problems for coordination of agents that require less (or no) symmetry breaking than the total gathering. We investigate the problems

especially in terms of the total moves or solvability and compare them with the total gathering.

First, we introduce a variation of the total gathering problem, called the *g-partial gathering problem*. The g-partial gathering problem is a generalization (or relaxation) of the total gathering problem. This problem requires, for a given positive integer g, that each agent should move to a node so that at least g agents should meet at each of the nodes they terminate at. In the g-partial gathering problem, we investigate the total moves compared with the total gathering problem. While the total gathering problem requires all the agents to meet at the same node, the g-partial gathering problem allows agents to meet at multiple nodes. Hence, the requirement for the g-partial gathering problem is weaker than that for the total gathering problem, that is, the g-partial gathering problem requires less symmetry breaking than the total gathering problem. Thus, agents aim to solve the g-partial gathering problem with fewer total than the total gathering problem.

We consider the g-partial gathering problem in ring networks (Chapter 3) and tree networks (Chapter 4). In ring networks, we assume that each node has a whiteboard where agents can read and write information. Then, if the algorithm is deterministic and assumes unique ID of each agent, or the algorithm is randomized and assumes no IDs of each agent (i.e., anonymous), agents can achieve the g-partial gathering in $O(gn)$ (expected for the randomized algorithm) total moves, where n is the number of nodes. Note that in ring networks, the total gathering problem requires $\Omega(kn)$ total moves, where k is the number of agents. Since $g < k$ holds, we show that agents can achieve the g-partial gathering in fewer total moves than the total gathering problem. Note that agents can attain this improvement of the total moves since the g-partial gathering requires less symmetry breaking than the total gathering problem.

In tree networks, since trees have lower symmetry than rings, we aim to solve the g-partial gathering problem in weaker models than the whiteboard model used in rings. We consider the model such that each agent has one removable token and ability to detect whether there is at least one agent at the current node or not. Note that this model is weaker than the whiteboard model considered in a ring scenario. Then, agents can achieve the g-partial gathering in $O(gn)$ total moves. Note that agents require $\Omega(kn)$

total moves to solve the total gathering problem also in tree networks. Thus, we show that also in tree networks, agents can achieve the g-partial gathering in fewer total moves than the total gathering problem

Second, we introduce the *uniform deployment problem* in ring networks, which requires agents to spread uniformly in the ring network (Chapter 5). In the uniform deployment problem, we investigate the solvability compared with the total gathering problem. Remind that in the total gathering problem, agents need to completely break the symmetry. On the other hand, in the uniform deployment problem agents need to attain the symmetry (i.e., require no symmetry breaking), and attaining symmetry is easier than breaking symmetry. Hence, there is possibility that agents can achieve the uniform deployment in several configurations from which the total gathering can not be achieved. As our result, if agents have knowledge of k and the algorithm requires termination detection, or agent do not have any knowledge and the algorithm does not require termination detection, even for agent with no distinct IDs our proposed algorithms achieve the uniform deployment from any initial configuration, including configurations such that the total gathering cannot be achieved. Note that agents can attain this solvability since the uniform deployment problem requires no symmetry breaking.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

A *distributed system* [1] consists of autonomous computers (*nodes*) and communication links. Nodes execute a *distributed algorithm* [2] to solve a problem and provide a service. To design distributed algorithms, *symmetry breaking* is one of fundamental concepts [3]. This is a technique to select several (possibly one) nodes as special nodes from candidate nodes. When symmetry breaking is achieved, nodes can provide a service based on the selected nodes. There are a lot of researches for symmetry breaking. For example, the *leader election problem* [1] requires to select the exactly one node as a leader node among all nodes. When the leader election is achieved, the selected node can instruct the other node to coordinate. The *maximal independent set problem* [7] requires to select a maximal set of nodes such that there are no link connecting two nodes included in the set. When the maximal independent set is achieved, the selected nodes can behave as local base stations. Symmetry breaking is considered in various networks (e.g., rings [3, 4, 5] and general graphs [6, 7]), and is achieved by using distinct IDs [3, 4], network topology [6] or random numbers [6, 7, 5]. Symmetry breaking has been extensively studied, and it has been known to be difficult, and sometimes impossible from several settings.

In recent years, distributed systems have become large and design of distributed systems (e.g., symmetry breaking) has become complicated. As a promising paradigm of distributed systems, (mobile) agent systems have attracted a lot of attention [8, 9]. Agents can traverse the system, collect information and execute tasks on nodes. Hence,

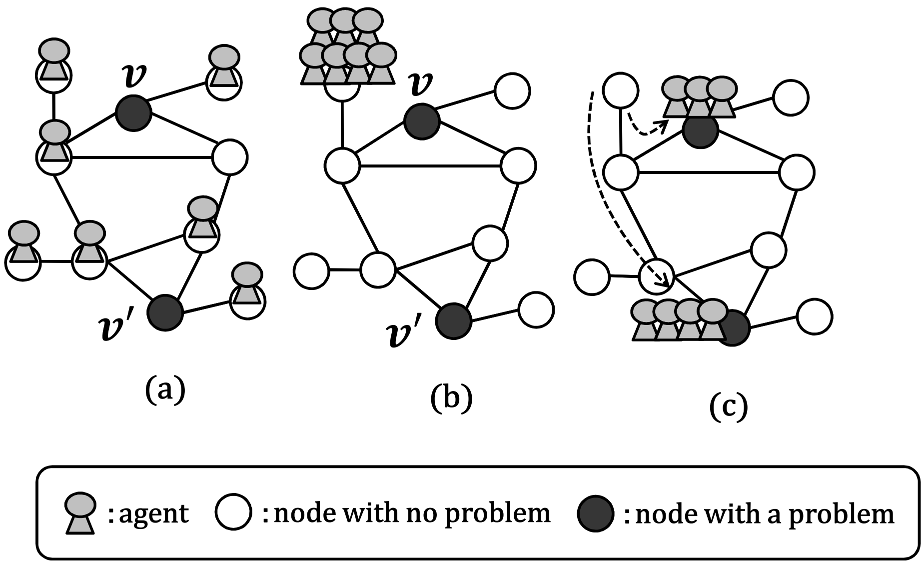

Figure 1.1: An example of the total gathering

we can encapsulate data and algorithms in agents, which simplifies design of distributed systems [10, 11]. Actually agent systems have many applications such as network exploration, network management, electronic commerce and so on.

The *total gathering problem* (or the rendezvous problem) [22] is a fundamental problem for coordination of agents.[1] This problem requires all the agents to meet at a single node. By meeting at a single node, agents can share information or synchronize their behavior. For example in Fig. 1.1 (a), we assume that nodes v and v' have troubles. When agents meet at a single node, agents can share such information (Fig. 1.1 (a) to (b)). Hence, after the gathering agents can determine their behavior so that they can execute tasks collaboratively and efficiently (Fig. 1.1 (b) to (c)).

Even though the achievement of the total gathering can simplify the distributed

[1]Usually it is simply called the gathering problem. In this dissertation, we call it the total gathering problem in contrast to the partial gathering problem we introduce.

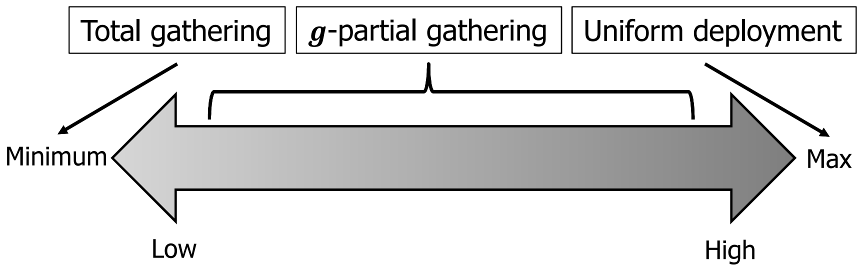

Figure 1.2: The symmetry each problem eventually requires

system, the problem also requires to break (or reduce) the symmetry as mentioned before. Since symmetry breaking is known to be difficult and sometimes impossible, there are two problems about the total gathering problem. The first is about the total moves, that is, agents require more total moves to solve the total gathering problem, which causes high network loads. The second is about the solvability, that is, if agents do not have distinct IDs, they cannot solve the total gathering problem from several initial configurations.

1.1 Overview of This Dissertation

In this dissertation, we introduce two problems for coordination of agents, called the g-partial gathering problem and the uniform deployment problem, which require less or no symmetry breaking than the total gathering problem. For such problems, we investigate the total moves and the solvability compared with the total gathering problem. Fig. 1.2 shows the symmetry each problem eventually requires.

1.1.1 Partial Gathering

First, we introduce the variation of the total gathering problem, called the *g-partial gathering problem.* The g-partial gathering problem is a generalization of the total gathering problem. This problem does not always require all agents to meet at a single node, but requires agents to gather partially at several nodes. More precisely, the g-partial gathering problem requires, for a given positive integer g, that each agent should move to a

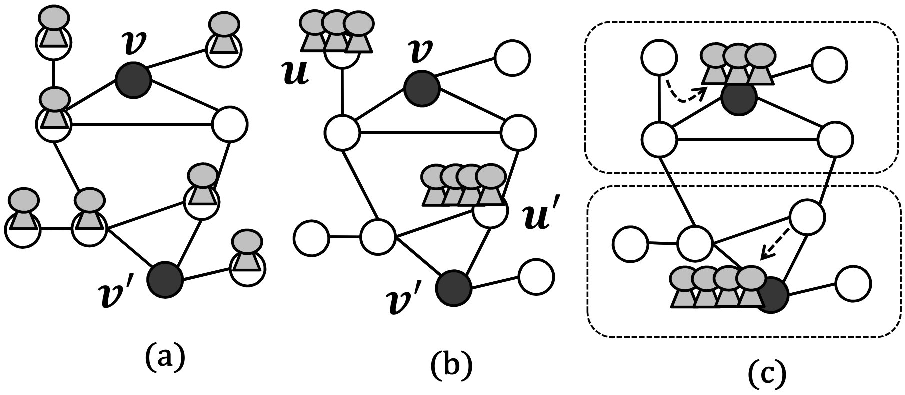

Figure 1.3: An example of the g-partial gathering ($g = 3$)

node so that at least g agents should meet at each of the nodes they terminate at. From a practical point of view, the g-partial gathering problem is still useful especially in large-scale networks. This is because, when agents achieve the g-partial gathering, agents can share information and execute tasks with collaboration among at least g agents (Fig. 1.3 (a) to Fig. 1.3 (b)). In addition, while in the total gathering agents meet at a single node, in the g-partial gathering agents meet at multiple nodes separately. This means that each group with at least g agents can partition the network and own its area that they should monitor efficiently (Fig. 1.3 (b) to Fig. 1.3 (c)).

The g-partial gathering problem is interesting to investigate also from theoretical point of view, and we investigate the problem in terms of the total moves and compare it with the total gathering problem. While the total gathering problem requires all the agents to meet at the same node, the g-partial gathering problem allows agents to meet at multiple nodes. Hence, the g-partial gathering problem has a weaker requirement than the total gathering problem, that is, the g-partial gathering problem requires less symmetry breaking than the total gathering problem. Thus, agents aim to solve the g-partial gathering problem with fewer total moves (i.e. lower costs) than the total gathering problem.

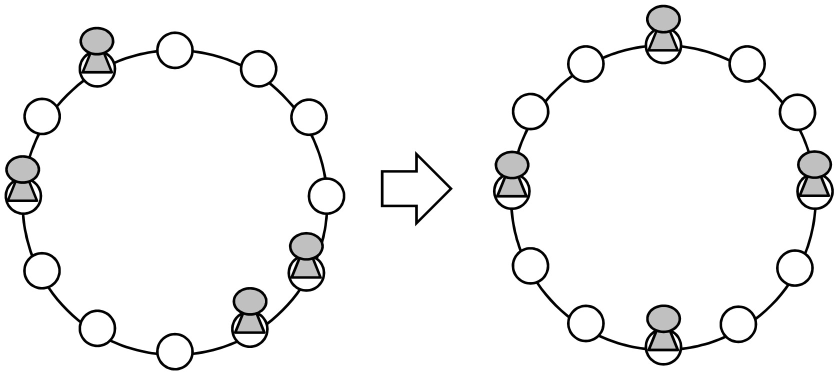

Figure 1.4: An example of the uniform deployment

1.1.2 Uniform Deployment

Second, we introduce the *uniform deployment problem* in ring networks, which requires agents to spread uniformly in the network like Fig. 1.4. From a practical point of view, the uniform deployment is useful for the network management. For instance, if agents with ability to repair faulty nodes are deployed uniformly, such agents can quickly reach and repair faulty nodes after the faults are detected. If agents with database replicas are deployed uniformly, each node can quickly access the database. Hence, we can regard the uniform deployment problem as a kind of the resource allocation problem. The uniform deployment is interesting to investigate also from a theoretical point of view, and we investigate the solvability compared with the total gathering problem. The problem exhibits a striking contrast to the total gathering: the uniform deployment aims to attain the symmetry of agent locations (i.e., requires no symmetry breaking) while the total gathering aims to break the symmetry. Remind that the symmetry breaking is difficult (and sometimes impossible) in distributed systems. Hence, it is interesting to clarify how easily the uniform deployment can be achieved compared with the total gathering.

1.2 Related Works

There exist a lot of researches for coordination of agents. In the following, we explain several problems in each subsection.

1.2.1 Exploration Problem

The *exploration* problem requires that every node is visited at least once by some agent. For a single agent, Sudo at el. [12] considered it under the assumption that each node has a whiteboard, and Dieudonné at el. [13] considered it with *Byzantine tokens*, that is, tokens on nodes continues to appear and disappear. For multiple agents placed at distinct nodes in the initial configuration, Chalopin at el. [14] considered it using tokens in arbitrary networks, and Gasieniec at el. [15] considered the memory requirement in tree networks. For multiple agents placed at the same node in the initial configuration, Dereniowski et al. [16] considered the trade-off between the upper bound of time and the number of agents in tree networks and arbitrary networks, and Yann et al [17] considered the trade-off between the lower bound of time and the number of agents in tree networks.

1.2.2 Leader Agent Election Problem

The *leader agent election problem* is a fundamental problem that requires symmetry breaking. This problem requires agents to select one common agent as a leader among all agents. The leader agent election problem is considered in ring networks for agents using tokens [18], in arbitrary networks under the assumption that each node has a whiteboard [19], in arbitrary networks for agents that cannot mark nodes in any way but can communicate with other agents staying at the same node [20]. The *gossip* problem requires all agents to share information that each agent initially has. Suzuki et al. [21] considered it under the assumption that each node has a whiteboard and agents can communicate with other agents staying at the same node. They showed that if agents solve the leader agent election problem, agents can solve the gossip problem asymptotically optimal in terms of total moves.

1.2.3 Total Gathering Problem

The total gathering problem has been extensively studied. Kranakis at el. [22] considered the total gathering problem for the first time. They considered it for two agents in ring networks, and this work has been extended to consider multiple agents [23, 24]. While [22, 23, 24] assumed that algorithms are deterministic and each agent has a token, Kawai at el. [25] considered a randomized algorithm to solve the total gathering problem under the assumption that each node has a whiteboard. Kranakis at el. [26] considered the total gathering problem in torus networks, and in [27] they conclude the results of [22, 23, 24, 26].

1.2.3.1 Total Gathering for Synchronous Agents or Asynchronous Agents

The total gathering problem for *synchronous agents* is considered in [28, 29, 30]. While [28] considered it for two agents with distinct IDs, [29] considered it for two agents with no distinct IDs but with knowledge where they are located in the network. Dieudidonné and Pelc [30] considered it for multiple agents with ability to communicate with the agents at the same node. The total gathering problem for *asynchronous agents* is considered in [31, 32, 33, 34]. Marco at el. [31] considered it for such agents for the first time. Czyzowicz at el. [33] considered it for two agents with distinct IDs, and Guilbault and Pelc [32] considered it for two agents with no distinct IDs. While in [32, 33] agents require exponential total moves to solve the problem, Dieudonné and Pelc [34] proposed an algorithm to solve the problem in polynomial total moves.

1.2.3.2 Fault Tolerant Gathering

A *fault tolerant gathering problem* is considered in [35, 36, 37, 38, 39]. Flocchini at el. [35] considered the total gathering in ring networks with *faulty tokens*, where tokens may disappear during the execution of the algorithm. In [35], they consider it for synchronous agents, and this work was extended to consider asynchronous agents [36]. Das at el. [37] considered such a problem in arbitrary networks. A *Byzantine gathering problem* is considered in [38, 39], where there exist Byzantine agents that execute any malicious behavior. Dieudonné at el. [38] proposed an algorithm with the minimum number of non-faulty agents under the assumption that Byzantine agents execute any malicious behavior except for changing their IDs. Bouchard at el. [39] proposed an algorithm with the minimum number of non-faulty agents under the assumption that Byzantine agents execute any malicious behavior, including changing their IDs.

1.2.4 Relation Between the Total Gathering Problem and Symmetry

As mentioned before, to solve the total gathering problem, agents need to break (or reduce) the symmetry. It is known that the symmetry breaking is difficult and sometimes impossible. Due to its difficulty, there are two problems about the total gathering problem. The first is about the total moves, that is, agents require more total moves to solve the total gathering problem, which causes high network loads. The second is about the solvability, that is, if agents do not have distinct IDs, they cannot solve the total gathering problem from several initial configurations.

1.3 Organization of This Dissertation

This dissertation consists of six chapters. In Chapter 2, we describe definitions of our system model, agent model, and each problem. In Chapter 3, we propose algorithms to solve the g-partial gathering problem in ring networks. In Chapter 4 we propose algorithms to solve the g-partial gathering problem in tree networks. In Chapter 5 we propose algorithms to solve the uniform deployment problem in ring networks. We conclude this dissertation in Chapter 6.

Chapter 2

Preliminary

In this chapter, we describe a general definition of a agent model. A network is modeled as a undirected graph $G = (V, L)$, where V is a set of nodes and L is a set of a communication links. We denote by $n \,(= V)$ the number of nodes. We assume that nodes are anonymous (i.e., have no distinct IDs), but each node $v_j \in V$ has a whiteboard that agents on node v_i can read from and write on. We assume that each link l incident to v is uniquely labeled at v with a label chosen from the set $\{0, 1, \ldots, d_v - 1\}$. We call this label *port number*. Since each communication link connects two nodes, it has two port numbers. However, port numbering is *local*, that is, there is no coherence between the two port numbers. 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 $i \ne j$. The *distance* from u to v is the length of the shortest path from u to v.

Let $A = \{a_0, a_1, \ldots, a_{k-1}\}$ be a set of k agents. For simplicity, operations to an index of an agent assume calculation under modulo k. We consider two problem settings: *agents with communication capability* and *agents without communication capability*. Agents with communication capability can send a message of any size to agents at the same node. Agents without communication capability can not communicate with other agents directly, but instead they communicate via whiteboards. An agent is a state machine having an initial state $s_{initial}$. Agents move in the network according to its state transition function. An agent executes the following seven operations in an atomic *step*:

1. The agent reaches some node v (or starts operation at v).
2. For the case of agents with communication capability, the agent receives all the messages (if any).
3. The agent obtains information at v (e.g., the state of the whiteboard and agents staying at v).
4. The agent executes local computation at v.
5. For the case of agents with communication capability, the agent broadcast a message to all the agents staying at v (if any) if it decides to send a message.
6. The agent updates the contents of v's whiteboard.
7. The agent moves to the next node or stays at v.

A (global) *configuration* c is defined as a product of states of agents, states of nodes, messages reached some agent but not consumed yet (for agents with communication capability), and location of agents. We denote by C the set of all possible configurations, In initial configuration $c \in C$, all agents are in the initial state and placed at distinct nodes. In Chapters 3 and 4, we consider the following scheduler and execution (In Chapter 5, we consider another scheduler and execution, and we explain the detail in Chapter 5). When configuration c_i changes to c_{i+1}, a *scheduler* activates a non-empty set of agents, say A_i, and each agent in A_i takes a step as mentioned before. We denote by such a transition $c_i \xrightarrow{A_i} c_{i+1}$. We assume that the scheduler is *fair*, that is, each agent is activated after a finite (unknown) amount of time and infinitely many times. If several agents at the same node are included in A_i, the scheduler activates the agents in an arbitrary exact order. When $A_i = A$ holds for every i, all agents take steps every time. This model is called the *synchronous model*. Otherwise, the model is called the *asynchronous model*. In this dissertation, we consider the asynchronous system. 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. We assume that any execution E is maximal in the sense that E is infinite, or ends in final configuration c_{final} where every agent's state is s_{final}.

Chapter 3

Partial Gathering in Ring Networks

3.1 Introduction

In this chapter, we present algorithms to achieve the g-partial gathering problem in asynchronous unidirectional rings with whiteboards on nodes. The aim in this chapter is to clarify the difference on the move complexity between the total gathering problem and the g-partial gathering problem.

3.1.1 Contribution

The contribution of this paper is summarized in Table 3.1, where k is the number of agents and n is the number of nodes. 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 for which the g-partial gathering

Table 3.1: Results in each model

	Model 1 (Section 3.3)	Model 2 (Section 3.4)	Model 3 (Section 3.5)
Unique agent 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

n: number of nodes, k: number of agents, g: minimum number of agents at each node where agents exist

problem is unsolvable. Next, we propose a deterministic algorithm to solve the g-partial gathering problem for any solvable initial configuration. This algorithm requires $O(kn)$ total moves. Note that the total gathering problem requires $\Omega(kn)$ total moves regardless of deterministic or randomized settings. This is because in the case that all the agents are uniformly deployed, at least half agents require $O(n)$ moves to meet at one node. Hence, the first and second algorithms imply that the g-partial gathering problem can be solved with fewer total moves than the total gathering problem for the both cases. Note that agents can attain this improvement of the total moves since the g-partial gathering requires less symmetry breaking than the total gathering problem. In addition, we show a lower bound $\Omega(gn)$ of the total moves 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.

3.1.2 Related works

The gathering problem for rings has been extensively studied [27, 22, 23, 18, 24, 35, 40, 25] because algorithms for such highly symmetric topologies give techniques to treat the essential difficulty of the gathering problem such as breaking symmetry.

For example, Kranakis et al. [22] considered the gathering problem for two mobile agents in ring networks. This algorithm allows each agent to use a token to select the gathering node based on the token locations. Later this work has been extended to consider any number of agents [23, 24]. Flocchini et al. [23] showed that if one token is available for each agent, the lower bound on the space complexity per agent is $\Omega(\log k + \log\log n)$ bits, where k is the number of agents and n is the number of nodes. Later, Gasieniec et al. [24] proposed the asymptotically space-optimal algorithm for uni-directional ring networks. Barriere et al. [18] considered the relationship between the gathering problem and the leader agent election problem. They showed that the gathering problem and the leader agent election problem are solvable under only the assumption that the ring has sense of direction and the numbers of nodes and agents are relatively prime.

A fault tolerant gathering problem is considered in [35, 41]. Flocchini et al. [35] considered the gathering problem when tokens fail and showed that knowledge of n (number of agents) allows better time complexity than knowledge of k (number of agents). Dobrev et al. [41] considered the gathering problem for the case that there exists a dangerous node, called a black hole. A black hole destroys any agent that visits there. They showed that it is impossible for all agents to gather and they considered how many agents can survive and gather.

A randomized algorithm to solve the gathering problem is shown in [25]. Kawai et al. considered the gathering problem for multiple agents under the assumption that agents know neither k nor n, and proposed a randomized algorithm to solve the gathering problem with high probability in $O(kn)$ total moves.

3.1.3 Organization

This chapter is organized as follows. In Section 3.3 we consider the first model, that is, the algorithm is deterministic and each agent has a distinct ID. In Section 3.4 we consider the second model, that is, the algorithm is randomized and agents are anonymous. In Section 3.5 we consider the third model, that is, the algorithm is deterministic and agents are anonymous. Section 3.6 concludes this chapter.

3.2 Preliminary

3.2.1 System Model

In this chapter, we restrict the network topology only to unidirectional ring networks. Then, ring $R = (V, L)$ is defined as follows:

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

For simplicity, operations to an index of a node assume calculation under modulo n, that is, $v_{(i+1) \bmod n}$ is simply represented by v_{i+1}. We define the direction from v_i to v_{i+1} as the *forward* direction, and the direction from v_{i+1} to v_i as the *backward* direction. Note that the ring is unidirectional, agents staying at some node can move only in the forward direction. In addition, we define the i-th ($i \neq 0$) forward (resp., backward) agent a'_h of agent a_h as the agent such that there are $i-1$ agents between a_h and $a_{h'}$ in the ${a_h}'$s forward (resp., backward) direction. Moreover, we call the a_h's 1-st forward and backward agents *neighboring agents* of a_h respectively.

3.2.2 Agent Model

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 a_h as a finite automaton $(S, \delta, s_{initial}, s_{final})$. The first element S is the set of the a_h's all states, which includes initial state $s_{initial}$ and final state s_{final}. When a_h changes its state to s_{final}, it terminates the algorithm. The second element δ is the state transition function. We denote by W a set of all state (contents) of a whiteboard. Then, since we treat deterministic algorithms, δ is a mapping $S \times W \to S \times W \times M$, where $M = \{1, 0\}$ represents whether the agent makes a movement or not in the step. The value 1 represents movement to the next node and 0 represents stay at the current node. Since rings are unidirectional, each agent moves only to its forward node. Note that if the state of a_h is s_{final} and the state of its current node's whiteboard is w_i, then $\delta\left(s_{final}, w_i\right) = (s_{final}, w_i, 0)$ holds. In addition, we assume that each agent cannot detect whether other agents exist at the current node or not. Moreover, we assume that each agent knows neither the number of nodes n nor agents k. Notice that $S, \delta, s_{initial}$, 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 a mapping $S \times W \times R \to S \times W \times M$, where R represents a set of random values. Note that if the state of some agent is s_{final} and the state of its current node's whiteboard is w_i, then $\delta\left(s_{final}, w_i, R\right) = \left(s_{final}, w_i, 0\right)$ holds. In addition, we assume that each agent cannot detect whether other agents exist at the current node or not, but we assume that each agent knows the number of agents k. Notice that all the agents are modeled by the same state machine since they are anonymous.

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 k. Note that all the agents are modeled by the same state machine. In each model, each agent executes the following three operations in an atomic step: 1) The agent reads the contents of its current node's whiteboard, 2) the agent executes local computation, 3) the agent updates the contents of the node's whiteboard, and 4) moves to the next node or stays at the current node. We assume that agents move instantaneously, that is, agents always exist at nodes (do not exist at links).

3.2.3 System Configuration

In this chapter, a (global) configuration c is defined as a product of states of agents, states of nodes (whiteboards' contents), and locations of agents. In initial configuration $c_0 \in C$, we assume that each node v_j has boolean variable $v_j.initial$ at the whiteboard 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. We consider a fair scheduler defined in Chapter 2, that is, it activates a non-empty set of agents A_i, and each agent in A_i takes a step as mentioned in Section

3.2.3. We also consider execution $E = c_0, c_1, \dots$ defined in Chapter 2.

3.2.4 Problem Definition

The g-partial gathering problem requires, for a given positive integer g, each agent to 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 3.2.1. *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 exists an agent on v_j, there exist at least g agents on v_j.* □

For ring networks, we have the following lower bound on the total number of agent moves. This theorem holds in both deterministic and randomized algorithms.

Theorem 3.2.1. *The total number of agent moves required to solve the g-partial gathering problem is $\Omega(gn)$ if $g \geq 2$.*

Proof. We consider an initial configuration such that all agents are scattered evenly (i.e., all the agents have the same distances to their nearest agents). 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 number of 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 $(i-1)n/k$. Hence, we have

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

Therefore, the total number of 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 number of moves is at least $\Omega(gn)$. □

3.3 The First Model: 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 is that agents elect a leader and then the leader instructs other agents which nodes they meet at. However, since $\Omega(n \log k)$ total moves are required to elect one leader [21], this approach cannot lead to the g-partial gathering in asymptotically optimal total moves (i.e., $O(gn)$). To achieve the partial gathering in $O(gn)$ total moves, we elect multiple agents as leaders by executing the leader agent election partially. By this behavior, the number of moves for the election can be bounded by $O(n \log g)$. In addition, we show that the total number of moves for agents to move to their gathering nodes by leaders' instruction is $O(gn)$. Thus, our algorithm solves the g-partial gathering problem in $O(gn)$ total moves.

The algorithm consists of two parts. In the first part, multiple agents are elected as leader agents. In the second part, the leader agents instruct the other agents which nodes they meet at, and the other agents move to the nodes by the instruction.

3.3.1 The first part: leader election

The aim of the first part is to elect leaders that satisfy the following conditions called *leader election conditions*: 1) At least one agent is elected as a leader, and 2) there exist at least $g-1$ non-leader agents between two leader agents. To attain this goal, we use a traditional leader election algorithm [42]. However, the algorithm in [42] is executed by nodes and the goal is to elect exactly one leader. Hence we modify the algorithm to be executed by agents, and then agents elect multiple 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.

For an intuitive understanding, we first explain the idea of leader election by assuming that the ring is synchronous and bidirectional. Later, the idea is applied to our model, that is, asynchronous unidirectional rings. The algorithm consists of several phases. In each phase, each active agent compares its own ID with IDs of its backward and forward neighboring active agents. More concretely, each active agent a_h writes its own ID id_2 to the whiteboard of its current node, and moves backward and forward. Then, a_h observes ID id_1 of its backward active agent and id_3 of its forward active agent. After this, a_h decides if it remains active or drops out from the candidates of leaders. Concretely, if its own ID id_2 is the smallest among the three IDs, a_h remains active (as a candidate of leaders) in the next phase. Otherwise, a_h drops out from the candidate of leaders and becomes inactive. Note that, in each phase, neighboring active agents never remain as candidates of leaders. Thus, at least half active agents become inactive in each phase. Moreover from [42], after executing j phases, there exist 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, and 2) the number of inactive agents between two active agents is at least $g-1$. Therefore, all remaining active agents become leaders since they satisfy the leader election conditions. Note that, before executing $\lceil \log g \rceil$ phases, the number of active agents may become one. In this case, the active agent immediately becomes a leader.

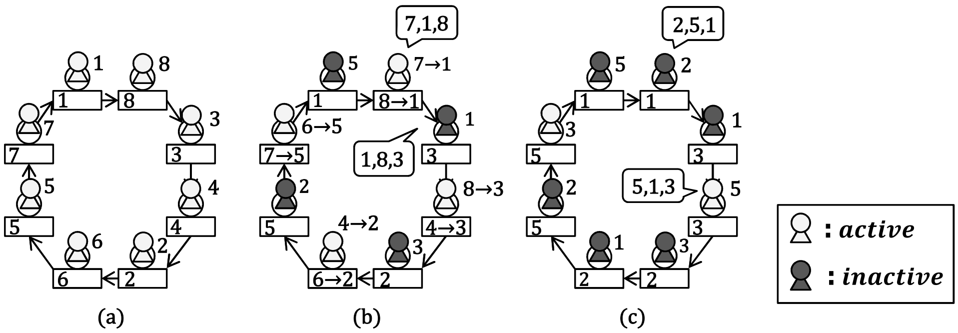

Figure 3.1: An execution example of the leader election part $(k = 8, g = 3)$

In the following, we implement the above algorithm in asynchronous unidirectional rings. First, we implement the above algorithm in a unidirectional ring by applying a traditional technique [42]. Let us consider the behavior of active agent a_h. In unidirectional rings, a_h cannot move backward and 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 if a_h remains active as a candidate, it assigns id_2 to its ID and starts the next phase.[1]

For example, consider the initial configuration in Fig. 3.1 (a). In the figures, the number near each agent is the ID of the agent and the box of each node represents the whiteboard. In the first phase, 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. 3.1 (b). In this configuration, each agent

[1]We imitate the way in [42], but active agent a_h may still use its own ID id_1 in the next phase.

compares three IDs. The agent with ID 1 observes IDs (1, 8, 3), and hence 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 hence it remains active as a candidate because the middle ID 1 is the smallest. Then, it updates its ID to 1 and starts the next phase. The agents with IDs 8, 4, and 6 also remain active as candidates and similarly update their IDs and start the next phase. In the second phase, active agents with updated IDs with 1,2,3, and 5 move until they observe two IDs of active agents respectively, and then the configuration change is changed to the one in Fig. 3.1 (c). In this configuration, the agent with ID 2 observes IDs (2, 5, 1), and it drops out from the candidate because the middle ID is not the smallest. Similarly, the agent with ID 1 also drops out from the candidate. On the other hand, the agent with ID 5 observes IDs (5, 1, 3), and it remain active because the middle ID is the smallest. Similarly, the agent with ID 3 remains active. Since agents with IDs 5 and 3 execute 2 ($= \lceil \log g \rceil$) phases, they become leaders.

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 current ID of a_h. After that, 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 details 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 3.1 and 3.2. All agents start the algorithm with active states, and the behavior of active agent a_h is described in Algorithm 3.1. We describe v_j by the node that a_h currently exists. If 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 3.3.2. Agent a_h and node v_j have the following variables:

- $a_h.id_1, a_h.id_2$, and $a_h.id_3$ are variables for a_h to store IDs of three successive active agents. Agent a_h stores its ID on $a_h.id_1$ and initially assigns its initial ID $a_h.id$ to $a_h.id_1$.

- $a_h.phase$ is a variable for a_h to store its own phase number.

- $v_j.phase$ and $v_j.id$ are variables for an active agent to write its phase number and its ID. For any v_j, initial values of these variables are 0.

- $v_j.inactive$ is a variable to represent whether there exists an inactive agent at v_j or not. That is, agents update the variable to keep the following invariant: If there exists an inactive agent on v_j, $v_j.inactive = true$ holds, and otherwise $v_j.inactive$=*false* holds. Initially $v_j.inactive = false$ holds for any v_j.

In Algorithm 3.1, a_h uses procedure *BasicAction*(), by which agent a_h moves to node $v_{j'}$ satisfying $v_{j'}.phase = a_h.phase$.

The pseudocode of *BasicAction*() is described in Algorithm 3.2. 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). Note that 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 (line 6 of Algorithm 3.1).

In the following, we explain the details of the treatment of asynchronous agents. 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, consider the transition from the configuration of Fig. 3.2 (a) to that of Fig. 3.2 (b) and 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

Algorithm 3.1 The behavior of active agent a_h (v_j is the current node of a_h)

Variables in Agent a_h

int $a_h.phase$;

int $a_h.id_1$,$a_h.id_2$,$a_h.id_3$;

Variables in Node v_j

int $v_j.phase$;

int $v_j.id$;

boolean $v_j.inactive = false$;

Main Routine of Agent a_h

1: $a_h.phase = 1$
2: $a_h.id_1 = a_h.id$
3: $v_j.phase = a_h.phase$
4: $v_j.id = a_h.id$
5: $BasicAction()$
6: **if** $(v_j.phase = a_h.phase) \wedge (v_j.id = a_h.id_1)$ then change its state to a leader state
7: $a_h.id_2 = v_j.id$
8: $BasicAction()$
9: $a_h.id_3 = v_j.id$
10: **if** $a_h.id_2 \geq \min(a_h.id_1, a_h.id_3)$ **then**
11: $\quad v_j.inactive = true$
12: $\quad$ change its state to an inactive state
13: **else**
14: $\quad$ **if** $a_h.phase = \lceil \log g \rceil$ **then**
15: $\qquad$ change its state to a leader state
16: $\quad$ **else**
17: $\qquad a_h.phase = a_h.phase + 1$
18: $\qquad a_h.id_1 = a_h.id_2$
19: $\quad$ **end if**
20: $\quad$ return to step 3
21: **end if**

$a_h.phase > v_c.phase$ holds. 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 $v_c.inactive = true$

Algorithm 3.2 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**

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, in the transition from the configuration of Fig. 3.3 (a) to that of Fig. 3.3 (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, in the transition from the configuration of Fig. 3.4 (a) to that of Fig. 3.4 (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, since $a_b.phase < v_c.phase$ holds, a_b continues to move without waiting for the update of $v_c.phase$. In addition since

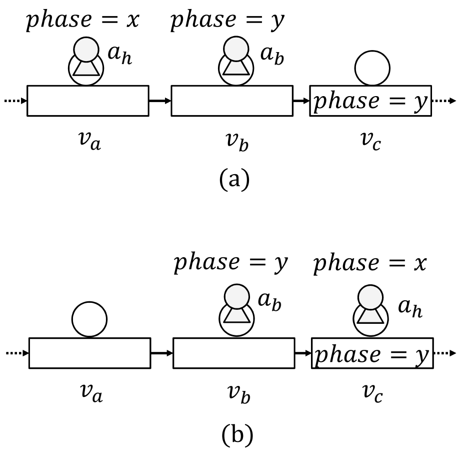

Figure 3.2: The first example of agent a_h that passes other agents (e.g, a_b)

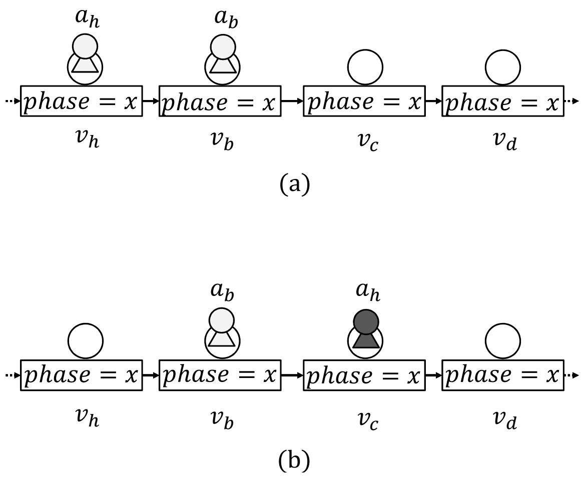

Figure 3.3: The second example of agent a_h that passes other agents (e.g., a_b)

a_h has updated $v_c.id$, a_h sees $v_c.id = a_b.id$. Thus since $a_b.id_1 = a_b.id_2$ holds, a_b becomes inactive when it reaches v_d. After that, a_h resumes the movement.

We have the following lemma about Algorithm 3.1 similarly to [42].

Lemma 3.3.1. *Algorithm 3.1 eventually terminates, and the configuration satisfies the* *following properties.*

- *There exists at least one leader agent.*
- *There exist at least $g-1$ inactive agents between two leader agents.*

Proof. At first, we show that Algorithm 3.1 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. In addition if agent a_h passes another agent $a_{h'}$, a_h waits for $a_{h'}$ at some node v_j until either $v_j.phase$ or $v_j.inactive$ is updated (lines 5 to 8 in Algorithm 3.2). Since the passed agent $a_{h'}$ eventually reaches v_j and updates either $v_j.phase$ or $v_j.inactive$, it does not happen that a_h waits at v_j forever. Moreover, by the time executing $\lceil \log g \rceil$ phases, if there exists exactly one active agent and the other agents are inactive, the active agent changes its state to a leader state. Therefore, Algorithm 3.1 eventually terminates. In the following, we show the above two properties.

First, we show that there exists at least one leader agent. From Algorithm 3.1, in each phase if $a_h.id_2$ is smallest of the three IDs, a_h remains active. Otherwise, a_h becomes inactive. Since each agent uses a unique ID, if there exist at least two active agents in some phase i, at least one agent remains active after executing the phase i. Moreover, from line 6 of Algorithm 3.1, 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.

Next, 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 on the phase number and by using the fact that in each phase if an agent a_h remains as a candidate of leaders, then its backward and forward active agents drop out from candidates of leaders. For the case $j=1$, there exists at least $1=2^1-1$ inactive agents between two active agents. For the case $j=l$, we assume that there exist at least 2^l-1 inactive agents between two active agents. Then, after executing $l+1$ phases, since at least one of neighboring active agents becomes inactive, the number of inactive agents between two active agents is at least

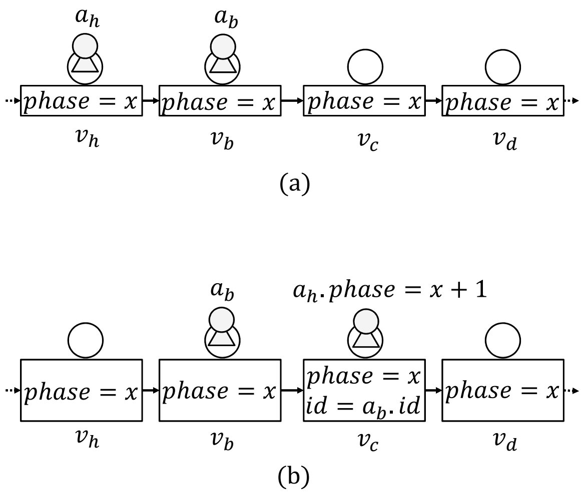

Figure 3.4: The third example of agent a_h that passes other agents (e.g, a_b)

$(2^l - 1) + 1 + (2^l - 1) = 2^{l+1} - 1$. 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 [42].

Lemma 3.3.2. *The total number of agent moves to execute Algorithm 3.1 is* $O(n \log g)$.

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

3.3.2 The second part: movement to gathering nodes

The second part achieves 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. In this part, states of agents are divided into the following three types:

- *leader*: The agent instructs inactive agents where they should move.

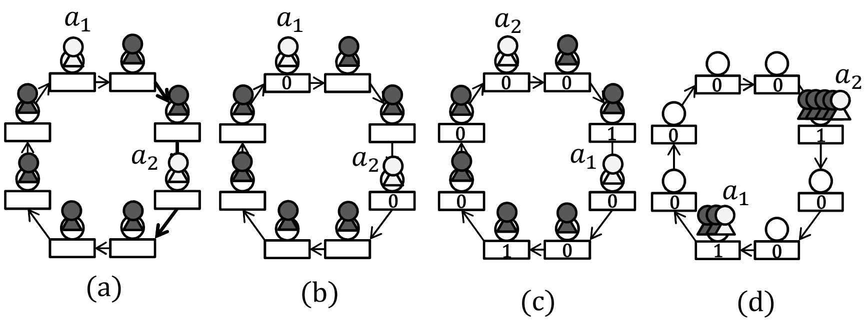

Figure 3.5: The realization of partial gathering ($g = 3$)

- *inactive*: The agent waits for the leader's instruction.
- *moving*: The agent moves to its gathering node.

The idea of the algorithm is to divide agents into groups each of which consists of at least g agents. Concretely, first each leader agent a_h writes 0 to the whiteboard on the current node (i.e., the leader node). Next, a_h moves to the next leader node, that is, the node where 0 is already written to the whiteboard. While moving, whenever 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 writes 0 to the whiteboard. Otherwise, a_h writes 1 to the whiteboard. These numbers are used to indicate whether the node is a gathering node or not. 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. When a_h reaches the next leader node, it changes its own state to a moving state, and we explain the behavior of moving agents later. For example, consider the configuration in Fig. 3.5 (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 (Fig. 3.5 (b)), and then they move and write numbers to whiteboards until they visit the node where 0 is already written to the whiteboard. Then, the system reaches the configuration in Fig. 3.5 (c).

Each non-leader (i.e., inactive agent) a_h waits at the current node until the value of the whiteboard is updated. When the value is updated, a_h changes its own state

Algorithm 3.3 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 $v_j.isGather = \perp$;

to a moving state. Each moving agent moves to the nearest node where 1 is written to the whiteboard. For example, after the configuration in Fig. 3.5 (c), each non-leader agent moves to the node where 1 is written to the whiteboard and the system reaches the configuration in Fig. 3.5 (d). After that, agents can solve the g-partial gathering problem.

Pseudocode. The pseudocode to achieve the partial gathering is described in Algorithm 3.3 to 3.6. 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. In addition, $v_j.inactive = true$ 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 $v_{j'}.initial = false$ holds.

At first, the variables needed to achieve the g-partial gathering are described in Algorithm 3.3. For leader agents instructing inactive agents gathering nodes, agent a_h and node v_j have the following variables:

- $a_h.count$ is a variable for a_h to count the number of inactive nodes a_h visits (The counting is done modulo g). The initial value of $a_h.count$ is 0.
- $v_j.isGather$ is a variable for leader agents to write values to indicate whether node v_j is a gathering node or not. That is, when a leader agent a_h visits an inactive node v_j, a_h writes 1 to $v_j.isGather$ to indicate v_j is a gathering node if $a_h.count = 0$, and a_h writes 0 to $v_j.isGather$ otherwise. The initial value of $v_j.isGather$ is $\perp$.

The pseudocode of leader agents is described in Algorithm 3.4. Since agents move asynchronously, it is possible that there exists active agents executing the first part and leader agents executing the second part at the same time. Hence, it may happen that

Algorithm 3.4 The behavior of leader agent a_h (v_j is the current node of a_h)

$v_j.isGather = 0$
$a_h.count = a_h.count + 1$
move to the forward node
while $v_j.isGather = \perp$ **do**
    while $v_j.initial = false$ **do**
        move to the forward node
    end while
    if $(v_j.inactive = false) \wedge (v_j.isGather = \perp)$ **then**
        wait until $v_j.inactive = true$ or $v_j.isGather \neq \perp$
    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
        $a_h.count = (a_h.count + 1) \bmod g$
        move to the forward node
    end if
end while
change to a moving state

some leader agent a_h may pass some active agent a_i. In this case, a_h waits until a_i catch up with a_h and a_i becomes a leader or inactive. More precisely, when leader agent a_h visits the node v_j such that $v_j.initial = true$ and $v_j.inactive = false$ and $v_j.isGather = \perp$ hold, it detects that it passes some active agent a_i. 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 the agent caches up with it and

Algorithm 3.5 The behavior of inactive agent a_h (v_j is the current node of a_h)

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

Algorithm 3.6 The behavior of moving agent a_h (v_j is the current node of a_h)

```
1: while v_j.isGather ≠ 1 do
2:     move to the forward node
3:     if (v_j.initial = true) ∧ (v_j.isGather = ⊥) then
4:         wait until v_j.isGather ≠ ⊥
5:     end if
6: end while
```

either $v_j.inactive = true$ or $v_j.isGather \neq \perp$ holds (lines 8 to 10). When the leader agent updates $v_j.isGather$, an inactive agent on node v_j changes to a moving state (line 17). After a leader agent reaches the next leader node, it changes its own state to a moving state (line 22). The behavior of inactive agents is described in Algorithm 3.5.

The pseudocode of moving agents is described in Algorithm 3.6. Moving agent a_h moves to the nearest node v_j such that $v_j.isGather = 1$ holds. When all agents complete such moves, the g-partial gathering problem is solved. 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 $v_j.isGather = \perp$ hold, a_h detects that it passed a leader agent. Then, a_h waits there until the leader agent comes and updates $v_j.isGather$ (lines 3 to 5).

We have the following lemma about the algorithm in Section 3.3.2.

Lemma 3.3.3. *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. Let $v^g_0, v^g_1, \ldots, v^g_l$ be nodes such that $v^g_j.isGather = 1$ holds $(0 \leq j \leq l)$ after all leader agents complete their behaviors, and we call these nodes *gathering nodes*. From Algorithm 3.6, each moving agent moves to the nearest gathering node v^g_j. By Lemma 3.3.1, there exist at least

$g-1$ moving agents between v_j^g and v_{j+1}^g Hence, agents can solve the g-partial gathering problem. In the following, we consider the total number of moves required to execute the algorithm.

First, the total number of moves required for each leader agent to move to its next leader node is obviously n. Next, let us consider the total number of moves required for each moving agent to move to nearest gathering node v_j^g (For example, the total moves from Fig 3.5 (c) to Fig 3.5 (d)). Remind that there are at least $g-1$ inactive agents between two leader agents and each leader agent a_h writes 1 to $v_j.isGather$ after writing 0 $g-1$ times. Hence, there are at most $2g-1$ moving agents between v_j^g and v_{j+1}^g. Thus, the total number of these moves is $O(gn)$ because each link is passed by at most $2g$ agents. Therefore, we have the lemma. □

From Lemmas 3.3.2 and 3.3.3, we have the following theorem.

Theorem 3.3.1. *When agents have distinct IDs, our deterministic algorithm solves the g-partial gathering problem in $O(gn)$ total moves.* □

3.4 The Second Model: A Randomized Algorithm for Anonymous Agents

In this section, we propose a randomized algorithm to solve the g-partial gathering problem for anonymous agents under the assumption that each agent knows the total number k of agents. The idea of the algorithm is the same as that in Section 3.3. In the first part, agents execute the leader election partially and elect multiple leader agents. In the second part, the leader agents determine gathering nodes and all agents move to the nearest gathering nodes. In the previous section each agent uses distinct IDs to elect multiple leader agents, but in this section each agent is anonymous and uses random IDs. We also show that the g-partial gathering problem is solved in $O(gn)$ expected total moves.

3.4.1 The first part: leader election

In this subsection, we explain a randomized algorithm to elect multiple leaders by using random IDs. Similarly to Section 3.3.1, the aim in this part is to satisfy the following conditions (leader election conditions): 1) At least one agent is elected as a leader, and 2) there exist at least $g-1$ non-leader agents between two leader agents. The basic idea is the same as Section 3.3.1, that is, each active agent moves in the ring and compares three random IDs. If the ID in the middle is the smallest of the three random IDs, the active agent remains active. Otherwise, the active agent drops out from the candidate of leaders.

Now we explain details of the algorithm. In the beginning of each phase, each active agent selects $3 \log k$ random bits as its own ID. After this, each agent executes in the same way as Section 3.3.1, that is, each active agent moves until it observes two random IDs of active agents and compares three random IDs. If the observed three IDs are distinct, the agent can execute the leader agent election similarly to Section 3.3.1. In addition to the behavior of the leader election in Section 3.1, when an agent becomes a leader at node v_j, the agent sets a *leader-flag* at v_j, and we explain how leader-flags are used later. If no agent observes a same random ID, the total number of moves for the leader agent election is the same as in Section 3.3.1, that is, $O(n \log g)$. In the following, we consider the case that some agent observes a same random ID.

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 3.3.1, that is, if $a_h.id_2 < a_h.id_1 = a_h.id_3$ holds, a_h remains active and a_h becomes inactive otherwise. For example, let us consider a configuration of Fig. 3.6 (a). Each active agent moves until it observes two random IDs (Fig. 3.6 (b)). Then, agent a_1 observes three random IDs (2,1,2) and remains active because $a_1.id_2 < a_1.id_1 = a_1.id_3$ holds. 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 same random IDs and behave similarly to Section 3.3.1, 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

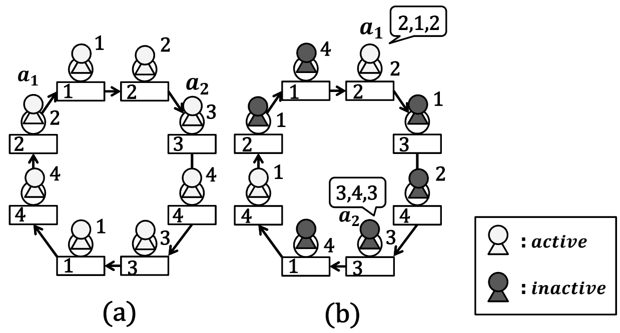

Figure 3.6: An example that some agent observes the same random IDs

inactive.

Next, we consider the case that either $a_h.id_2 = a_h.id_1$ or $a_h.id_2 = a_h.id_3$ holds. In this case, a_h changes its own state to a *semi-leader* state. A semi-leader is an agent that has a possibility to become a leader if there exists no leader agent in the ring. When at least one agent becomes a semi-leader, each active agent becomes inactive. The outline of the behavior of each semi-leader agent is as follows: First each semi-leader travels a round in the ring. After this, if there already exists a leader agent in the ring, each semi-leader becomes inactive. Otherwise, the leader election is executed among all semi-leader agents, and exactly one semi-leader is elected as a leader and the other agents become inactive (including active agents). Note that, we can show that the probability some active agent becomes a semi-leader is sufficiently low and the expected number of semi-leader agents during the leader election is also sufficiently small. Hence even when each semi-leader travels a round in the ring several times, the expected total moves to complete the leader agent election can be bounded by $O(n \log g)$.

Now, we explain the detailed behavior for semi-leader agents. When an active agent a_h becomes a semi-leader, it sets a *semi-leader-flag* on its current whiteboard. In the following, the node where the semi-leader flag is set (resp., not set) is called *a semi-leader node* (resp., *a non-semi-leader node*). After that, semi-leader agent a_h travels a round in the ring. In the travel, when a_h visits a non-semi-leader 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* at v_j. This flag is used so that other agents notice the existence of a semi-leader and become inactive. Moreover when a_h visits a semi-leader node, a_h compares its random ID with the random ID written to the current whiteboard. Then, a_h memorizes whether its random ID is smaller or not and whether another semi-leader has the same random ID as its random ID or not.

After traveling a round in the ring, a_h decides if it becomes a leader or inactive. While traveling in the ring, if a_h observes a leader-flag, it learns that there already exists a leader agent in the ring. In this case, a_h becomes inactive. Otherwise, a_h decides if it becomes a leader or inactive depending on random IDs. Let $a_h.id$ be a_h's random ID and A_{min} be the set of semi-leaders such that each semi-leader $a_h \in A_{min}$ has the smallest random ID id_{min} among all semi-leaders. In this case, each semi-leader $a_h \notin A_{min}$ clears a semi-leaders-flag and becomes inactive. On the other hand, if a_h has the unique minimum random ID (i.e., $|A_{min}| = 1$), a_h becomes a leader. Otherwise, a_h selects a random ID again, writes the ID to the current whiteboard, travels a round in the ring. Then, a_h obtains new random IDs of semi-leaders. Each semi-leader a_h repeats such a behavior until $|A_{min}| = 1$ holds.

Pseudocode. The pseudocode to elect leader agents is given in Algorithm 3.7 to 3.11. Algorithm 3.7 represents variables required for the behavior of active agents, and Algorithm 3.8 represents the behavior of active agents. Agent a_h and node v_j have the following variables:

- $a_h.id_1, a_h.id_2$, and $a_h.id_3$ are variables for a_h to store random IDs of three successive active agents. Note that a_h stores its own random ID on $a_h.id_1$.
- $a_h.phase$ is a variable for a_h to store its phase number.
- $v_j.phase$ and $v_j.id$ are variables for an active agent to write its phase number and its random ID. For every v_j, initial values of these variables are 0.
- $v_j.tour\text{-}flag$ and $v_j.leader\text{-}flag$ are variables to represent whether there exists an semi-leader agent and a leader agent or not respectively. The initial values of these variables are $false$.

Algorithm 3.7 Values required for the behavior of active agent a_h (v_j is the current node of a_h)

Variables for 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 for Node v_j

int $v_j.phase$;

int $v_j.id$;

boolean $v_j.inactive = false$;

boolean $v_j.tour\text{-}flag = false$;

boolean $v_j.leader\text{-}flag = false$;

- $a_h.semiObserve$ is a variable for a_h to decide whether it observes a tour-flag or not. The initial value of $a_h.semiObserve$ is $false$.

In addition to these variables, agents a_h uses the procedure $random(l)$ to get its own random ID. This procedure returns l random bits.

In each phase, each active agent selects its own random ID of $3 \log k$ bits length through $random(3 \log k)$, and moves until it observes two random IDs by $BasicAction()$ in Algorithm 3.2. If each active agent a_h neither observes a tour-flag nor observes phase numbers and random IDs such that $(a_h.phase = v_j.phase) \wedge (a_h.id_2 = a_h.id_1 \vee a_h.id_2 = a_h.id_3)$ holds, this pseudocode works similarly to Algorithm 3.3.1. In this case when an agent becomes a leader, the agent sets a leader-flag at v_j (lines 20 to 23). 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 (lines 11 to 14). Remind that $v_j.inactive$ is a variable to represent whether there exists an inactive agent or not. If an active agent a_h observes three random IDs such that $(a_h.phase = v_j.phase) \wedge (a_h.id_2 = a_h.id_1 \vee a_h.id_2 = a_h.id_3)$ holds, then a_h changes its own state to a semi-leader state (line 15).

Algorithm 3.9 represents variables required for the behavior of semi-leader agents, and Algorithm 3.10 and Algorithm 3.11 represent the behavior of semi-leader agents.

Algorithm 3.8 The behavior of active agent a_h (v_j is the current node of a_h)

1: $a_h.phase = 1$
2: $a_h.id_1 = random(3 \log k)$
3: $v_j.phase = a_h.phase$
4: $v_j.id = a_h.id_1$
5: $BasicAction()$
6: **if** $v_j.tour = true$ **then** $a_h.semiObserve = true$
7: $a_h.id_2 = v_j.id$
8: $BasicAction()$
9: **if** $v_j.tour = true$ **then** $a_h.semiObserve = true$
10: $a_h.id_3 = v_j.id$
11: **if** $a_h.semiObserve = true$ **then**
12: $\quad v_j.inactive = true$
13: $\quad$ change its state to an inactive state
14: **end if**
15: **if** $(a_h.phase = v_j.phase) \wedge (a_h.id_1 = a_h.id_2 \vee a_h.id_2 = a_h.id_3)$ **then** change its state to a semi-leader state
16: **if** $a_h.id_2 \geq \min(a_h.id_1, a_h.id_3)$ **then**
17: $\quad v_j.inactive = true$
18: $\quad$ change its state to an inactive state
19: **else**
20: $\quad$ **if** $a_h.phase = \lceil \log g \rceil$ **then**
21: $\qquad v_j.leader\text{-}flag = true$
22: $\qquad$ change its state to a leader state
23: $\quad$ **else**
24: $\qquad a_h.phase = a_h.phase + 1$
25: $\quad$ **end if**
26: $\quad$ return to step 2
27: **end if**

Algorithm 3.9 Values required for the behavior of semi-leader agent a_h (v_j is the current node of a_h)

Variables for Agent a_h

int $a_h.semiPhase$;

int $a_h.semiID$;

int $a_h.agentCount$;

boolean $a_h.isMin = true$

boolean $a_h.isUnique = true$

boolean $a_h.leaderObserve = false$

Variables for Node v_j

int $v_j, semiPhase$;

int $v_j.id$;

boolean $v_j.leader\text{-}flag$;

boolean $v_j.semi\text{-}leader\text{-}flag$;

boolean $v_j.tour\text{-}flag$;

Semi-leader-agent a_h and node v_j have the following variables:

- $a_h.semiID$ is a variable for a_h to store its random ID.
- $a_h.agentCount$ is a variable for a_h to detect the completion of one round of the ring travel.
- $a_h.isMin$ is a variable for a_h to detect whether its random ID is the smallest or not. The initial value of $a_h.isMin$ is $true$.
- $a_h.isUnique$ is a variable for a_h to detect whether another semi-leader has the same random ID as its random ID. The initial value of a_h. $isUnique$ is $true$.
- $a_h.leaderObserve$ is a variable for a_h to detect whether there exists a leader agent in the ring or not. The initial value of $a_h.leaderObserve$ is false.
- $a_h.semiPhase$ is a variable for a_h to store its phase number in the semi-leader state.

Algorithm 3.10 The first half behavior of semi-leader agent a_h (v_j is the current node of a_h)

if $v_j.\mathit{tour\text{-}flag} = \mathit{true}$ **then**
  $v_j.\mathit{inactive} = \mathit{true}$
  change its state to an inactive state
end if
$v_j.\mathit{semi\text{-}leader\text{-}flag} = \mathit{true}$
$a_h.\mathit{semiPhase} = 1$
$v_j.\mathit{semiPhase} = a_h.\mathit{semiPhase}$
$v_j.\mathit{id} = \mathit{random}(3 \log k)$
$a_h.\mathit{semiID} = v_j.\mathit{id}$
while $a_h.\mathit{agentCount} \neq k$ **do**
  move to the forward node
  while $v_j.\mathit{initial} = \mathit{false}$ **do** move to the forward node
  $a_h.\mathit{agentCount} = a_h.\mathit{agentCount} + 1$
  if $v_j.\mathit{leader\text{-}flag} = \mathit{true}$ **then** $a_h.\mathit{leaderObserve} = \mathit{true}$
  if $v_j.\mathit{semi\text{-}leader\text{-}flag} = \mathit{true}$ **then**
    if $a_h.\mathit{semiPhase} \neq v_j.\mathit{semiPhase}$ **then** wait until $a_h.\mathit{semiPhase} = v_j.\mathit{semiPhase}$
    if $v_j.\mathit{id} < a_h.\mathit{semiID}$ **then** $a_h.\mathit{isMin} = \mathit{false}$
    if $v_j.\mathit{id} = a_h.\mathit{semiID}$ **then** $a_h.\mathit{isUnique} = \mathit{false}$
  else
    $v_j.\mathit{tour\text{-}flag} = \mathit{true}$
  end if
end while

- $v_j.\mathit{semiPhase}$ is a variable for a semi-leader agent to write its phase number in the semi-leader state.

Variables $a_h.\mathit{semiPhase}$ and $v_j.\mathit{semiPhase}$ are used for the case that there exist several semi-leaders having the same smallest random IDs. In addition to these variables, each node v_j has variables $v_j.\mathit{id}$, $v_j.\mathit{leader\text{-}flag}$, $v_j.\mathit{semi\text{-}leader\text{-}flag}$, and $v_j.\mathit{tour\text{-}flag}$ as defined

Algorithm 3.11 The latter half behavior of semi-leader agent a_h (v_j is the current node of a_h)

```
1: if a_h.leaderObserve = true then
2:    v_j.inactive = true
3:    change its state to an inactive state
4: end if
5: if a_h.isMin = false then
6:    v_j.semi-leader-flag = false
7:    v_j.inactive = true
8:    change its state to an inactive state
9: end if
10: if a_h.isUnique = true then
11:    change its state to a leader state
12: else
13:    a_h.semiPhase = a_h.semiPhase + 1
14:    a_h.agentCount = 0
15:    return to step 7 of Algorithm 3.10
16: end if
```

in Algorithm 3.7.

Before semi-leader a_h begins moving in the ring (from v_j), if it detects tour-flag at v_j, another semi-leader $a_{h'}$ has already visited v_j. Then a_h becomes inactive and does not start the travel in the ring (lines 1 to 4 of Algorithm 3.10). This is because, otherwise, each semi-leader cannot share the same random IDs. After each semi-leader travels a round in the ring, if there exists exactly one semi-leader whose random ID is the smallest, the semi-leader becomes a leader and the other semi-leaders become inactive. Otherwise, each semi-leader a_h whose random ID is the smallest updates its phase and random ID again, and travels a round in the ring (lines 12 to 15 of Algorithm 3.11). Then, a_h obtains new value of random IDs. Each semi-leader a_h repeats such a behavior until exactly one semi-leader has the smallest random ID.

We have the following lemmas similarly to Section 3.3.1.

Lemma 3.4.1. *Algorithm 3.8 eventually terminates, and the configuration satisfies the following properties.*

- *There exists at least one leader agent.*
- *There exist at least $g-1$ inactive agents between two leader agents.*

Proof. The above properties are the same as Lemma 3.1. Thus, if no agent becomes a semi-leader during the algorithm, each agent behaves similarly to Section 3.3.1 and the above properties are satisfied. Moreover if at least one agent becomes a semi-leader, exactly one semi-leader is elected as a leader and the other agents become inactive. Then, the above properties are clearly satisfied.

Therefore, we have the lemma. □

Lemma 3.4.2. *The expected total number of agent moves to elect multiple leader agents is $O(n \log g)$.*

Proof. If there exist no neighboring active agents having the same random IDs, Algorithms 3.8 works similarly to Section 3.3.1, and the total number of moves is $O(n \log g)$. In the following, we consider the case that some 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 active agents in the i-th phase, the probability that there exist neighboring active agents having the same random IDs is at most $k_i \times (\frac{1}{2})^l$. Since at least half active agents drop out from candidates in each phase, the probability that neighboring active agents have the same random IDs until the end of the $\lceil \log g \rceil$ phases 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 $\frac{2}{k^2} < \frac{1}{k}$. We assume that k active agents become semi-leaders and circulate around the ring because this case requires the most total moves. Then, each semi-leader a_h compares its random ID with random IDs of each semi-leader. Let A_{min} be the set of semi-leader agents whose random IDs are the smallest. If $|A_{min}| = 1$ holds, agents finish the leader agent election and the total number of moves is at most $O(kn)$. Otherwise, at least two semi-leaders have the same smallest random IDs. This probability is at most $k \times (\frac{1}{2})^l$. In this case, each semi-leader a_h updates its phase and random ID again, travels a round in the ring, and obtains new random IDs of each semi-leader. Each semi-leader a_h repeats such a behavior until $|A_{min}| = 1$ holds. We assume that $t = k \times (\frac{1}{2})^l$ and semi-leaders complete the leader agent election after they circulate around the ring s times. In this case, before they circulate around the ring $s-1$ times, $|A_{min}| \neq 1$ holds every time they circulate around the ring. In addition when they circulate around the ring s times, $|A_{min}| = 1$ holds, and the probability such that $|A_{min}| = 1$ holds is clearly less than 1. Hence, the probability such that agents complete the leader election after they circulate around the ring s times is at most $t^{s-1} \times 1 = t^{s-1}$, and the total number of moves is at most skn. Since the probability that at least one agent becomes a semi-leader is at most $\frac{1}{k}$, the expected total number of moves for the case that some agents become semi-leaders and complete the leader agent election is at most $O(n \log g) + \frac{1}{k} \times \sum_{s=1}^{\infty} t^{s-1} \times skn = n \sum_{s=1}^{\infty} st^{s-1}$. Let S_n be $1 \times 1 + 2 \times t + \cdots + nt^{n-1}$. Then, we have $S_n = (nt^{n+1} - (n-1)t^n + 1)/(1-t)^2$. When $n = \infty$, we have $S_n = 1/(1-t)^2$. Moreover since $t = k \times (\frac{1}{2})^l$ and $l = 3 \log k$ hold, we have $t < \frac{1}{2}$ and $S_n < 4$. Furthermore, the expected total number of moves is at most $O(n)$. Since the total moves to elect multiple leaders for the case that no agent becomes a semi-leader is $O(n \log g)$, the expected total moves for the leader election is $O(n \log g)$.

Therefore, we have the lemma. □

3.4.2 The second part: movement to gathering nodes

After executing the leader agent election in Section 3.4.1, the conditions shown by Lemma 3.4.1 is satisfied, that is, 1) At least one agent is elected as a leader, and 2) there exist at least $g-1$ inactive agents between two leader agents. Thus, we can execute the algorithms in Section 3.3.2 after the algorithms in Section 3.4.1. Therefore, agents can solve the g-partial gathering problem.

From Lemmas 3.3.3, 3.4.1, and 3.4.2, we have the following theorem.

Theorem 3.4.1. *When agents have no IDs, our randomized algorithm solves the g-partial gathering problem in expected $O(gn)$ total moves.* □

3.5 The Third Model: A 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.

3.5.1 Existence of Unsolvable Initial Configurations

To explain unsolvable initial configurations, we define the *distance sequence* of the initial configuration. For initial configuration c_0, we define the distance sequence of agent a_h as $D_h(c_0) = (d_0^h(c_0), \dots, d_{k-1}^h(c_0))$, where $d_i^h(c_0)$ is the distance between the i-th forward agent of a_h and the $(i+1)$-th forward agent of a_h in c_0. Then, we define the distance sequence of configuration c_0 as the lexicographically minimum sequence among $\{D_h(c_0) | a_h \in A\}$, and we denote it by $D(c_0)$. In addition, we define several functions and variables for sequence $D = (d_0, d_1, \dots, d_{k-1})$. Let $\mathit{shift}(D, x) = (d_x, d_{x+1}, \dots, d_{k-1}, d_0, d_1, \dots, d_{x-1})$ and when $D = \mathit{shift}(D, x)$ holds for some x such that $0 < x < k$ holds, we say D or the ring is *periodic* (Otherwise, we say D or the ring is *aperiodic*). Moreover, we define *period* of D as the minimum (positive) value such that $\mathit{shift}(D, \mathit{period}) = D$ holds.

Then, we have the following theorem.

Theorem 3.5.1. *Let c_0 be an initial configuration. If $D(c_0)$ is periodic and period is less than g, the g-partial gathering problem is not solvable.*

Proof. Let $m = k/\mathit{period}$. Let A_j $(0 \le j \le \mathit{period} - 1)$ be a set of agents a_h such that $D_h(c_0) = \mathit{shift}(D(c_0), j)$ 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. □

3.5.2 Proposed Algorithm

In this section, we propose a deterministic algorithm to solve the g-partial gathering problem in $O(kn)$ total moves for solvable initial configurations. Let $D = D(c_0)$ be the distance sequence of initial configuration c_0. From Theorem 3.5.1, 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 $period \geq g$ holds. In this section, we assume that each agent knows the number k of agents.

The idea of the algorithm is as follows: First each agent a_h travels a round in 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 $period\,(\geq g)$ agents meet at the node, and the algorithm solves the g-partial gathering problem.

We have the following theorem about Algorithm 3.12.

Theorem 3.5.2. *When agents have no IDs, our deterministic algorithm solves the g-partial gathering problem in $O(kn)$ total moves if the initial configuration is solvable.*

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 3.5.1 and a_h terminates the algorithm. In the following, we consider the case that $period \geq g$ holds. From line 20 in Algorithm 3.12, 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

Algorithm 3.12 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 of int $a_h.D[\,]$;

array of int $D_{min}[\,]$;

Main Routine of Agent a_h

$a_h.total = 0$
$a_h.dis = 0$
while $a_h.total \neq k$ **do**
  move to the forward node
  while $v_j.initial = false$ **do**
    move to the forward node
    $a_h.dis = a_h.dis + 1$
  end while
  $a_h.D[a_h.total] = a_h.dis$
  $a_h.total = a_h.total + 1$
  $a_h.dis = 0$
end while
let D_{min} be a lexicographically minimum sequence among $\{shift(a_h.D, x) | 0 \leq x \leq k-1\}$.
$period = \min\{x > 0 | shift(D_{min}, x) = D_{min}\}$
if $(g > period)$ **then**
  terminate the algorithm
  // the g-partial gathering problem is not solvable
end if
$a_h.x = \min\{x \leq 0 | shift(a_h.D, x) = D_{min}\}$
move to the forward node $\sum_{i=0}^{a_h.x-1} a_h.D[i]$ times

g-partial gathering problem.

Next, we analyze the total moves required to solve the g-partial gathering problem. In Algorithm 3.12, all agents circulate 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. □

3.6 Concluding Remarks

In this chapter, we proposed three algorithms to solve the g-partial gathering problem in asynchronous unidirectional rings. The first algorithm is deterministic and works for distinct agents. The second algorithm is randomized and works for anonymous agents under the assumption that each agent knows the total number of agents. The third algorithm is deterministic and works for anonymous agents under the assumption that each agent knows the total number of agents. In the first and second algorithms, several agents are elected as leaders by executing the leader agent election partially. The first algorithm uses agents' distinct IDs and the second algorithm uses random IDs. In the both algorithms, after the leader election, leader agents instruct the other agents where they meet. On the other hand, in the third algorithm, each agent moves around the ring and moves to a node and terminates so that at least g agents should meet at the same node. We have showed that the first and second algorithms requires $O(gn)$ total moves, which is asymptotically optimal.

Chapter 4

Partial Gathering in Tree Networks

4.1 Introduction

In this chapter, we present algorithms to achieve the g-partial gathering in asynchronous tree network. In Chapter 3, agents achieve the g-partial gathering in asynchronous rings under the assumption that each node has a whiteboard. In this chapter, since trees have lower symmetry than rings, we aim to solve the g-partial gathering problem in models weaker than the whiteboard model considered in Chapter 3's ring scenario.

4.1.1 Contribution

The contribution of this paper is summarized in Table 4.1. We consider two multiplicity detection models and two token models. Note that any combination of these multiplicity detection models and token models is weaker than the whiteboard model. First, we consider the non-token model. In this case, we show that agents require $\Omega(kn)$ total moves to solve the g-partial gathering problem even for the strong multiplicity detection model. We omit this result in Table 4.1. Next, we consider the case of the weak multiplicity detection and non-token model, where the weak multiplicity detection model assumes that each agent can detect whether another agent exists at the current node or not but

Table 4.1: Results in each model

	Model 1 (Section 4.4)		Model 2 (Section 4.5)	Model 3 (Section 4.6)
Token model	Non-token		Non-token	Removable-token
Multiplicity detection	Weak		Strong	Weak
Tree topology	Asymmetric	Symmetric	Arbitrary	Arbitrary
Solvability	Solvable	Insolvable ($g \geq 5$)	Solvable	Solvable
The total moves	$\Theta(kn)$ [46]	-	$\Theta(kn)$	$\Theta(gn)$

cannot count the exact number of the agents. In this case, for asymmetric trees, from [46] agents can achieve the g-partial gathering problem in $O(kn)$ total moves. From the lower bound of the total moves for non-token model, this algorithm is asymptotically optimal in terms of total moves. In addition, for that case that the tree is symmetric and $g \geq 5$ holds, we show that there exist no algorithms to solve the g-partial gathering problem. Hence, we need to relax the restriction of either the multiplicity detection or the token model. Next, we consider the case that the restriction of the multiplicity detection is relaxed: the strong multiplicity detection and non-token model, where the strong multiplicity detection model allows each agent to count the number of agents at the current node. In this case, we propose a deterministic algorithm to solve the g-partial gathering problem in $O(kn)$ total moves. From the lower bound of the total moves for the non-token model, this algorithm is also asymptotically optimal in terms of the total moves. Finally, we consider the case that the restriction of the token model is relaxed: 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. Note that in this model, agents require $\Omega(gn)$ total moves to solve the g-partial gathering problem. Hence, this algorithm is also asymptotically optimal in terms of the total moves.

4.1.2 Related works

Recently, the total gathering problem for trees has been extensively studied because tree networks are utilized in a lot of applications. For example, Fraigniaud and Pelc [43] considered the gathering problem in tree networks for the first time. This algorithm achieves the gathering for two synchronous agents with an arbitrary delay in starting time. The space complexity for each agent is $O(\log n)$ bits, which is asymptotically optimal [44]. Later, they considered the space complexity for the case that two synchronous agents start the algorithm at the same time [44]. In this case, they proposed an algorithm to achieve the gathering for $O(\log l + \log \log n)$ memory per agent, where l is the number of leaves.

The time complexity required for two agents' gathering in tree networks is considered in [45, 46]. Czyzowicz et al. [45] considered the trade-off between time and space complexities for two synchronous agents' gathering for the case that each agent has $k \geq c \log n$ memory bits (c is some constant). In this case, they proposed an algorithm to solve the gathering problem in $O(n + n^2/k)$ time, which is asymptotically optimal. Elouasbi and Pelc [46] considered the time complexity trade-off between determinism and randomization. They proposed a deterministic algorithm for two synchronous agents' gathering in $O(n)$ time. On the other hand, when agents know the maximum degree of the tree and the upper bound of the initial distance between two agents, they proposed a randomized algorithm to achieve the two synchronous agents' gathering with high probability in $O(\log n)$ time.

Asynchronous gathering for two or more agents is considered in [47]. Baba et al. showed a lower bound of space complexity for time-optimal algorithms, that is, they showed that each agent requires $\Omega(n)$ memory bits to solve the gathering problem in $O(n)$ time. In addition, they proposed a space-optimal algorithm to solve the gathering problem on the condition that the time complexity is asymptotically optimal, that is, both the time complexity and the space complexity are $O(n)$.

4.1.3 Organization

This chapter is organized as follows. In Section 4.3 we show the lower bound of total moves for the non-token model. In Section 4.4 we consider the first model, that is, the weak multiplicity detection and non-token model. In Section 4.5 we consider the second model, that is, the strong multiplicity detection and non-token model. In Section 4.6 we consider the third model, that is, the weak multiplicity detection and removable-token model. Section 4.7 concludes this chapter.

4.2 Preliminary

4.2.1 System Model

In this chapter, we restrict the network topology only to a tree network $T = (V, L)$. We describe several definition about T. First, we explain about center nodes. Let us consider the following sequence of trees constructed recursively as follows: $T_0 = T$ and T_{i+1} is obtained from T_i by removing all its leaves. Let j be the minimum value such that T_j has at most two nodes. Then, we call such nodes *center nodes*. We use the following theorem about center nodes later.

Theorem 4.2.1. [48] *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 Chapter 4.4.

Definition 4.2.1. *A tree T is symmetric iff there exists a function $\lambda : V \to V$ such that all the following conditions hold (See Fig. 4.1):*

- *For any $v \in V$, $v \neq \lambda(v)$ holds.*
- *For any $u, v \in V$, u is adjacent to v iff $\lambda(u)$ is adjacent to $\lambda(v)$.*
- *For any link $\{u, v\} \in L$, the port number assigned to $\{u, v\}$ at u is equal to the port number assigned to link $\{\lambda(u), \lambda(v)\}$ at $\lambda(u)$.*

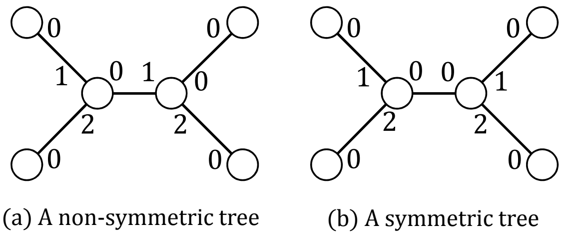

Figure 4.1: Asymmetric and symmetric trees

When tree T is symmetric, we say nodes u and v in T are symmetric if $u = \lambda(v)$ holds. When tree T is not symmetric, we say tree T is asymmetric. $\square$

4.2.2 Agent Model

We assume that agents know neither n nor k. We consider the *strong multiplicity detection model* and the *weak multiplicity detection model*. 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 at its current node. In both models, each agent cannot read the state of any other agent. In this chapter, we assume that each whiteboard has only 0 or 1 bit memory, that is, 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 leaves a token on its initial node at the beginning of the algorithm, and agents can remove any owner's token during the execution of the algorithm.

We assume that agents are anonymous (i.e., agents have no IDs) and execute a deterministic algorithm. Similarly to Section 3.2.2, We model an agent as a finite state machine $(S, \delta, s_{initial}, s_{final})$. In the weak multiplicity detection and non-token model, δ is described as $\delta : S \times M_T \times R_A \to S \times M_T$. 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 assigned at the current node to the link the agent used 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, $R_A = \{0, 1\}$ 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 $\delta : S \times M_T \times \{0, 1, \ldots, k-1\} \rightarrow S \times M_T$. In the definition, $\{0, 1, \ldots, k-1\}$ represents the number of other agents at the current node. In the weak multiplicity detection and removable-token model, δ is described as $\delta : S \times M_T \times R_A \times R_T \rightarrow S \times R_T \times M_T$. In the definition, in the left side of δ, $R_T = \{0, 1\}$ represents whether a token exists at the current node or not. The value 0 of R_T represents that there does not exist a token at the current node, and the value 1 of R_T represents that there exists a token at the current node. In the right side of δ, $R_T = \{0, 1\}$ represents whether the agent removes a token at the current node or not. If the value of R_T in the left side is 1 and the value of R_T in the right side is 0, it means that the 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 both models, we assume that each agent is not imposed any restriction on the memory.

During the execution of the algorithm, agents are located either on nodes or links. Each agent a_h executes the following three operations in an atomic step: 1) Agent a_h reaches some node v, 2) agent a_h executes local computation at v, and 3) agent a_h leaves v or stays there. In the local computation, agent a_h executes the following operations: 1) Agent a_h obtains information about its local configuration (i.e., the states of all agents at the current node v and the token state at v for the removable-token model) 2) agent a_h executes some computation at v, 3) agent a_h decides whether a_h removes the token or not for the case of the removable-token model, 4) agent a_h decides whether a_h moves to the next node or not, and 5) agent a_h decides the port number to leave from (in the case that it decides to move). We assume that a_h completes possible local computation at each step, that is, at the end of a step, a_h either leaves v or decides to stay at v.

If a_h decides to stay at v, after the decision a_h does nothing (i.e., does not change its state, does not remove the token at v, and does not leave v) unless other agents change a_h's local configuration. Note that the above atomic actions can be easily implemented if each node has a *buffer* that stores agents visiting the node and makes them execute processes in a FIFO order, and this assumption is very natural in a distributed system. In addition we assume that agents move in the tree network in a FIFO manner, that is, when agent a_h leaves some node v_j before another agent a_i leaves v_j through the same communication link as a_h, then a_h reaches v_j's neighboring node v'_j before a_i. Note that such FIFO assumptions are natural because 1) agents are implemented as messages in practice, and 2) FIFO assumptions of messages are natural and can be easily realized in distributed systems.

4.2.3 System Configuration

In the non-token model, a global configuration c is defined as a product of states of agents, states of links, and locations of agents. Here, the state of link (v_j, v'_j) is a sequence of agents that are in transit from v_j to v'_j in this order. In the removable-token model, configuration c is defined as a product of states of agents, states of nodes (tokens), states of links, and locations of agents. Note that in both models, the locations of agents are either on nodes or links. In addition, in the initial configuration c_0, we assume that 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 agent at v_j.

We consider a fair scheduler defined in Chapter 2, that is, it activates a non-empty set of agents A_i, and each agent in A_i takes a step as mentioned in Section 4.2.3. We assume that if the scheduler activates some agent a_j that is 1) in a sequence of agents that are in transit in some link (v_l,v'_l), but 2) not in the head of the sequence, then a_j does not take a step (i.e., does not reach v'_l). We also consider execution $E = c_0, c_1, \ldots$ defined in Chapter 2.

4.2.4 Problem Definition

In Definition 3.2.1, we defined the g-partial gathering problem in ring networks. We can use this definition also in tree networks. In addition, in Theorem 3.2.1 we showed that agent require $\Omega(gn)$ total moves to solve the g-partial gathering problem in ring networks. We can show that this lower bound holds also in tree networks by considering a line network such that $\lfloor g/2 \rfloor$ agents are placed at consecutive nodes starting from one endpoint and the other $k - \lfloor g/2 \rfloor$ agents are placed at consecutive nodes starting from the other endpoint. Then, clearly at least $\lfloor g/2 \rfloor$ agents need to move to the center node. This requires $\lfloor g/2 \rfloor \times \lfloor n/2 \rfloor = \lfloor gn/4 \rfloor$ moves.

4.3 Lower Bound of the Total Moves for the Non-Token Model

For the non-token model, we have the following lower bound of the total moves. This results holds even for the strong-multiplicity detection model.

Theorem 4.3.1. *In the 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. Let *a local configuration* of agent a staying at node v be a boolean value indicating whether another agent stays at v or not. Then, we define *a waiting state* of agents as follows: an agent a is in the waiting state at node v if a never leaves v before the local configuration of a changes. Concretely, there are two cases. The first case is that, when a visits node v and enters a waiting state at v, there exist no other agents at v. In this case, a neither changes its waiting state nor leaves v until another agent visits v. When the scheduler activates a and a observes such an agent, a can break its waiting state and leave v. The second case is that, when a visits v and enters a waiting state at v, there exists another agent at v. In this case, a neither changes its waiting state nor leaves v until there are no other

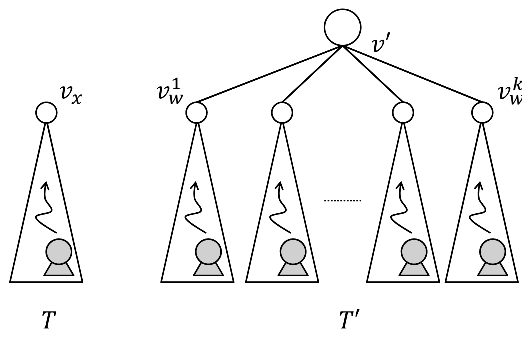

Figure 4.2: Figures of T and T'

agents at v. When the scheduler activates a and a detects such a situation, a can break its waiting state and can leave v.[1]

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 every agent not in a waiting state: 1) makes a movement, and 2) visits a node. Let a_i be the first agent that enters a waiting state in this execution. Clearly, a_i does not meet other agents unless it enters a waiting state. If a_i makes $\Omega(n)$ moves before it enters a waiting state, each of the other agents makes $\Omega(n)$ moves. This implies the total number of 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 which a_i does not visit before it enters a waiting state. Let v_w be the node where a_i is placed 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. 4.2). 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

[1] The final state of an agent after gathering is a waiting state. Hence, the final state is a kind of the waiting state.

knowledge of n, each agent performs 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 its execution. Therefore, algorithm A cannot solve the g-partial gathering problem in T'. This is a contradiction. □

4.4 Weak Multiplicity Detection and Non-Token Model

In this section, we consider the g-partial gathering problem for Model 1 in Table 4.1, that is, the weak multiplicity detection and non-token model. First, we consider the case for asymmetric trees, and agents can achieve the g-partial gathering problem in $O(kn)$ total moves from the past result. Next, we consider the case that the tree symmetric and agents are placed symmetrically in the initial configuration. In this case, we show that there exist no algorithms to solve the g-partial gathering problem if $g \geq 5$ holds.

4.4.1 Proposed algorithm for asymmetric trees

From [46], for asymmetric tree agents can achieve the total gathering in $O(kn)$ total moves, and this result can be clearly applied to the g-partial gathering. Hence, we have the following theorem.

Theorem 4.4.1. *In the weak multiplicity detection and non-token model, agents solve the g-partial gathering problem in $O(kn)$ total moves for asymmetric trees.* □

4.4.2 Impossibility result for symmetric trees

In this section, we show that there exist no algorithms to solve the g-partial gathering problem for symmetric trees. We consider the case such that in the initial configuration even agents are placed symmetrically in a symmetric tree, that is, if there exists an agent at node v, there also exists an agent at node v', where v and v' are symmetric. Then, we have the following theorem.

Theorem 4.4.2. *Let us consider the initial configuration such that agents are placed symmetrically in a symmetric tree. Then, in the weak multiplicity detection and non-token model, there exist no algorithms to solve the g-partial gathering problem if $g \geq 5$ holds.*

Proof. For contradiction, we assume that the g-partial gathering problem can be solved. We prove the theorem for the case that g is an odd number (we can also prove the theorem similarly for the case that g is an even number). We assume that the tree network is symmetric, and for any node v, we denote by v' the node symmetric to v. We consider the initial configuration c_0 such that $3g-1$ agents are placed symmetrically in the symmetric tree, that is, if there exists an agent at v, there also exists an agent at v'. For any agent a located at a node v in c_0, let a' denote the agent that is located at v' in c_0. Note that since $2g < k = 3g-1 < 3g$ holds, agents are allowed to meet at one or two nodes. Then, we have the following lemma [43].

Lemma 4.4.1. *Assume that each pair of nodes v_1 and v_1', v_2 and v_2', $\ldots$ v_m and v_m' is symmetric in tree T. If agents a_i and a_i' ($1 \leq i \leq m$) start an algorithm from v_i and v_i', respectively, there exists an execution in which each pair acts in a symmetric manner even in an asynchronous model.* □

We consider a waiting state defined in Section 4.3. Then, the definition means that even when the local configuration of some waiting agent changes, the agent does not change its state unless the scheduler activates the agent. Note that, if an agent is staying at some node, then it is either in an initial state or a waiting state. Then, we have the following lemma about a waiting state.

Lemma 4.4.2. *At any node v_j where at least three waiting agents exist, at least two of the 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 local configuration of a_1^j changes, and a_1^j can leave v_j. Since we consider the weak multiplicity detection model, even if a_1^j leaves v_j, a_2^j and a_3^j cannot detect the fact and local configurations of a_2^j and a_3^j do not change. Thus, agents a_2^j and a_3^j never leave v_j. □

Let us consider a configuration such that there exist at least three nodes where there exist at least three waiting agents, respectively. We call such a configuration a *three-node three-waiting-agent configuration*. Then in three-node three-waiting-agent configurations, by Lemma 4.4.2 there exist at least three nodes where agents exist at the end of the algorithm execution. In addition since agents are allowed to meet at one or two nodes because of $k < 3g$, agents cannot solve the g-partial gathering problem when the system reaches a three-node three-waiting-agent configuration. This is the key idea of the proof. We consider an adversarial scheduler such that once some agent enters a waiting state, the scheduler never activates the agent until all agent enter waiting states. When all agents are in waiting state, we denote by such a configuration c_t. Note that c_t is the configuration such that all agents' states are waiting states and each agent enters a waiting state exactly once. Then, the outline of the proof is described as follows. At first, we construct configuration c_t by considering the adversarial scheduler. Then, we consider the placement of waiting agents in c_t and show the unsolvability in any placement. If c_t is a three-node three-waiting-agent configuration or a configuration such that there exists at most one waiting agent at each node, we can clearly show that agents cannot solve the g-partial gathering problem. Otherwise, we show that, in any placement of waiting agents in c_t, there exists an execution by an adversarial scheduler such that the system reaches either 1) a three-node three-waiting-agent configuration, 2) a configuration such that there exists at most one waiting agent at each node, or 3) a configuration such that there exist two nodes with agents but there exist at most $g-1$ waiting agents at one of them.

At first, we consider the execution until the system reaches the first configuration c_t such that all agents are in waiting states. We consider an execution E_t under the following fair scheduler α_t that makes agents' movements as follows: 1) When α_t activates some agent a whose initial node is v, α_t also activates the agent a' whose initial node v' at the same time, and 2) if an agent a enters a waiting state at node v, α_t never activates a and

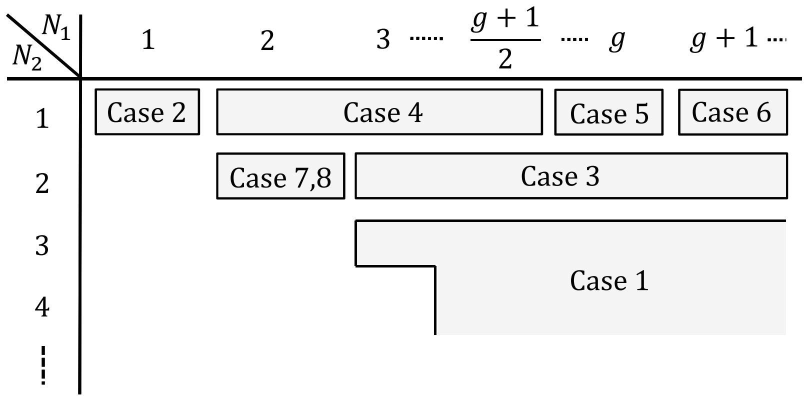

Figure 4.3: Classification depending on values of N_1 and N_2 ($N_1 \geq N_2$)

a never leaves v until all agents enter waiting states.[2]

Note that, in any algorithm, each agent necessarily enters a waiting state (otherwise, if an agent never enters a waiting state, the agent moves in the tree network forever). Agents execute such behaviors until they reach c_t. Then, since agents are initially placed symmetrically and move symmetrically, it follows that if there exist l waiting agents at a node v in c_t, there also exist l waiting agents at node v'. Thus we can denote the nodes where agents exist in c_t by $v_1, \ldots, v_s, v'_1, \ldots, v'_s$. In addition, let N_l (resp., N'_l) be the number of waiting agents at v_l (resp., v'_l) in c_t. Clearly, $N_l = N'_l \, (1 \leq l \leq s)$ and $N_1 + N_2 + \cdots + N_s = k/2$ hold. Without loss of generality, we assume that $N_1 \geq N_2 \geq \cdots \geq N_s$ holds. Moreover, we assume that agents $a^j_1, a^j_2, \ldots, a^j_{N_j}$ (resp., $a^{j'}_1, a^{j'}_2, \ldots, a^{j'}_{N'_j}$) enter waiting states at v_j (resp., v'_j) in this order.We consider the following eight cases depending on values of $N_1, N_2, \ldots, N_s$ ($N'_1, N'_2, \ldots, N'_s$), and show that agents cannot solve the g-partial gathering problem in any case (contradiction). Fig. 4.3 represents the classification depending on values of N_1 and N_2. In addition, Case 7 considers $N_1 = N_2 = 2$ and $N_3 = 1$, and Case 8 considers $N_1 = N_2 = N_3 = 2$.

⟨Case 1: $N_2 \geq 3$ holds.⟩

[2] Scheduler α_t is fair because the system reaches configuration c_t in finite number of agents' steps.

In this case, there exist at least three waiting agents at each of v_1, v_2, v_1' and v_2' (three-node three-waiting-agent configuration). Hence from Lemma 4.4.2, there exist at least four nodes where agents exist at the end of algorithm execution. However, since $k = 3g-1$ holds, agents are allowed to meet at one or two nodes. This contradicts the assumption that agents can solve the g-partial gathering problem.

⟨Case 2: $N_1 = N_2 = \cdots = N_s = 1$ holds.⟩
In this case, there exist no nodes where more than one agent exists in c_t. From the definition of a waiting state, the local configuration of each agent does not change and each agent never leaves the current node. This contradicts the assumption.

Before considering Case 3, we introduce the notion of elimination. Let us select a set of agents A_{elimi} such that both $|A_{elimi}| \leq g-1$ and $A_{elimi} \subseteq \{a_i^j | 1 \leq j \leq s, 2 \leq i \leq N_j\} \cup \{a_i^{j'} | 1 \leq j' \leq s, 2 \leq i \leq N_j'\}$ hold. In addition, let c_0^{elimi} be the configuration obtained from c_0 by eliminating all agents in A_{elimi} in c_0. Moreover we define an execution E_t^{elimi} as follows: When in E_t the scheduler activates sets of agents $A_0, A_1, \ldots, A_{t-1}$ in this order and the system reaches c_t, then in E_t^{elimi} the scheduler activates sets of agents $A_0 - A_{elimi}, A_1 - A_{elimi}, \ldots, A_{t-1} - A_{elimi}$ in this order and the system reaches c_t^{elimi}. Then, we have the following lemma.

Lemma 4.4.3. *The locations and states of agents in $A - A_{elimi}$ in c_t^{elimi} are the same as those in c_t.*

Proof. We prove the lemma for the case of $|A_{elimi}| = 1$. Then, we can similarly prove the lemma for the case $|A_{elimi}| \geq 2$ by applying the following argument to each of A_{elimi} one by one. Let a_i^j $(2 \leq i \leq N_j)$ be the unique agent in A_{elimi}. In this case, we show that the locations and states of agents in $A - A_{elimi}$ in c_l^{elimi} $(0 \leq l \leq t)$ are equal to those in c_l. At first, we denote by c_p the configuration in E_t immediately after a_i^j enters a waiting state at v_j. Note that a_i^j enters a waiting state without being observed by any other agents. This is because until c_p, a_i^j reaches some node v, executes local computation, and leaves the current node in an atomic step, that is, a_i^j never waits at any node before c_p. In addition, in c_p there already exist waiting agents $a^j_1, \dots, a^j_{i-1}$. Moreover, we denote by c_q $(p < q)$ the configuration in which some agent a visits v_j for the first time after c_p.

Now let us consider E^{elimi}_t. First we can show that, except for a^j_i, the locations and states of agents in each of $c^{elimi}_0, c^{elimi}_1, \dots c^{elimi}_p$ in E^{elimi}_t are the same as those in each of $c_0, c_1, \dots, c_p$ in E_t. This is because in E_t, a^j_i moves without being observed by any other agents. Similarly, we can show that the locations and states of agents in each of $c^{elimi}_{p+1}, \dots, c^{elimi}_{q-1}$ are the same as those except for a^j_i in each of $c_{p+1}, \dots, c_{q-1}$. Next, we consider the locations and states of agents in c^{elimi}_q. In c^{elimi}_q, some agent a visits v_j and then there exist $i-1$ waiting agents $a^j_1, \dots, a^j_{i-1}$ at v_j. On the other hand in c_q, there exist waiting agents $a^j_1, \dots, a^j_i$ at v_j. Then, agent a cannot distinguish the difference between c_q and c'_q because $i \geq 2$ holds and we consider the weak multiplicity detection model. Thus, agent a behaves in the same way as in E_t and the locations and states of agents in c^{elimi}_q are the same as those in c_q, except for a^j_i.

In the following, we show by induction that the locations and states of agents in each of $c^{elimi}_{q+1}, \dots, c^{elimi}_t$ are the same as those except for a^j_i in each of $c_{q+1}, \dots, c_t$. We assume that the locations and sates of agents in each of c^{elimi}_r $(q+1 \leq r \leq t-1)$ are the same as those except for a^j_i in each of c_r. Then, in c^{elimi}_t if there exists no agent that visits v_j, the locations and states of agents in c^{elimi}_{r+1} in E^{elimi}_{r+1} are the same as those in each of c_{r+1} in E_t. This is because between c_r and c_{r+1} in E_t, a^j_i stays at v_j and it is never observed by agents except for agents already staying at v_j. In c^{elimi}_{r+1} if there exists some agent a that visits v_j, there exist i' $(i' \geq i)$ waiting agents at v_j. Then, agent a cannot distinguish the difference between c_{r+1} and c^{elimi}_{r+1} because $i' \geq 2$ holds and we consider the weak multiplicity detection model. Hence, agent a behaves in the same way as in E_t and the locations and states of agents in c^{elimi}_{r+1} are the same as those in c_{r+1}, except for a^j_i. Thus, we can show that the locations and states of agents in each of $c^{elimi}_{q+1}, \dots, c^{elimi}_t$ are also the same as those except for a^j_i in each of $c_{q+1}, \dots, c_t$. Therefore, the locations and states of agents in $A - A_{elimi}$ in c^{elimi}_t are equal to those in c_t, and we have the lemma.

□

By Lemma 4.4.3 and the fact that in c_t all agents are in waiting states, we can clearly show that in c_t^{elimi} all agents are in waiting states. We use this lemma to show the contradiction in the remaining cases.[3]

$\langle$Case 3: $N_1 \geq 3$ and $N_2 = 2$ hold.$\rangle$
In this case, there exist three waiting agents a_1^1, a_2^1, and a_3^1 ($a_1^{1'}, a_2^{1'}$, and $a_3^{1'}$, respectively) at v_1 (v_1'), and agents a_2^1 and a_3^1 ($a_2^{1'}$ and $a_3^{1'}$, respectively) never leave v_1 (v_1') by Lemma 4.4.2. Since $k = 3g - 1$ holds and agents are allowed to meet at one or two nodes, all agents must meet at v_1 or v_1'.

Now let us consider the initial configuration c_0^{elimi} obtained from c_0 by eliminating agents a_2^2 and $a_2^{2'}$. Then from Lemma 4.4.3, there exists an execution E_t^{elimi} from c_0^{elimi} to c_t^{elimi}, where there exists exactly one waiting agent a_1^2 ($a_1^{2'}$) at v_2 (v_2') in c_t^{elimi}. In this configuration, agents a_1^2 and $a_1^{2'}$ need to meet at v_1 or v_1'. To do this, it is necessary that some agent enters a waiting state at v_2 and v_2' in order to make a_1^2 and $a_1^{2'}$ observe changes of local configurations and leave there. We consider an execution E_x^{elimi} under the scheduler α_x^{elimi} deciding agents and their behavior as follows. Let $b_1, \ldots, b_h$ ($b_1', \ldots, b_h'$) be the sequence of agents such that 1) b_1 (b_1') is an agent that can leave the current node in c_t^{elimi}, 2) b_i (b_i') ($2 \leq i \leq h-1$) is an agent in the waiting state at some node v_{bi} (v_{bi}') where no other agents exist (note that b_i can leave v_{bi} when b_{i-1} arrives at v_{bi} and enters a waiting state), and 3) b_h (b_h') is an agent in the waiting state at v_2 (v_2'), that is, $b_h = a_1^2$ ($b_h' = a_1^{2'}$). Then in α_x^{elimi}, agents b_j and b_j' ($1 \leq j \leq h-1$) are activated at the same time, and behave symmetrically. Finally, agents b_{h-1} and b_{h-1}' enter waiting states at v_2 and v_2', respectively, and we call such a configuration c_x^{elimi}. An example is shown in Fig. 4.4. In the figure, we assume that agents a_2^2 and $a_2^{2'}$ of the dotted lines are eliminated. In addition, the black agents $a_2^1, a_3^1, a_2^{1'}$, and $a_3^{1'}$ never leave the current nodes by the end of the algorithm. In Fig. 4.4, agents a_1^1 and $a_1^{1'}$ move symmetrically and enter waiting states at v_3 and v_3', respectively (Fig. 4.4 (b)), and after this, agents a_1^3 and $a_1^{3'}$ move symmetrically and enter waiting states at v_2 and v_2', respectively (Fig. 4.4 (c) to

[3]From Case 6 to Case 8, we consider a configuration obtained from c_0 by eliminating at least four agents, and we cannot apply this way for the case of $2 \leq g \leq 4$.

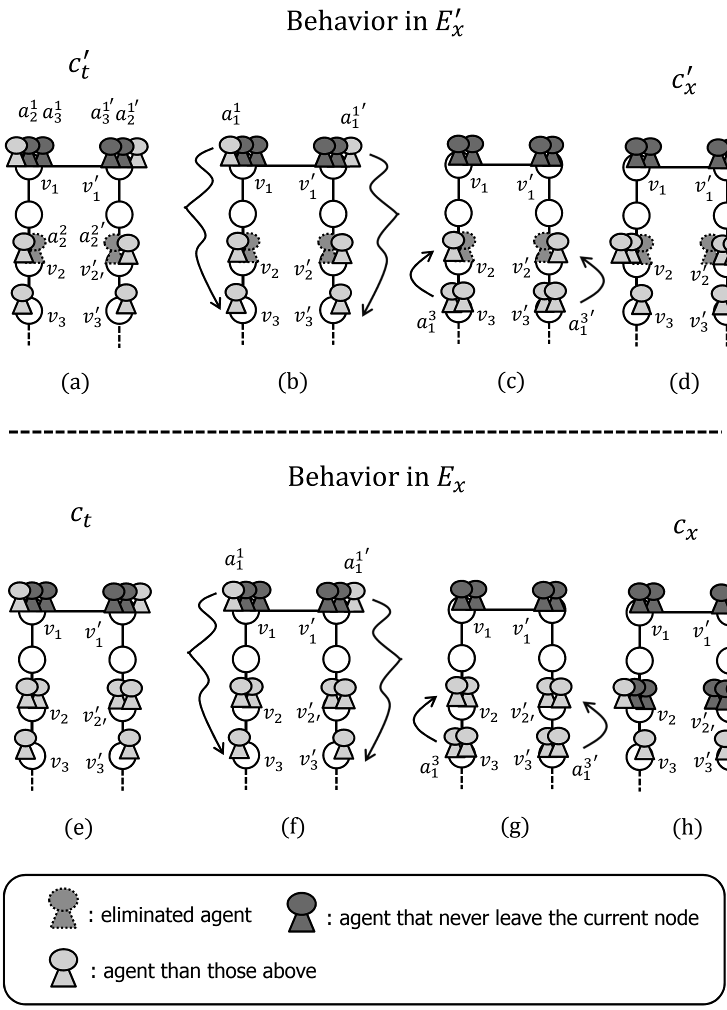

Figure 4.4: An example of Case 3

Fig. 4.4 (d)).

Now, let us consider c_t. In c_t, there exist two waiting agents a_1^2 and a_2^2 ($a_1^{2'}$ and $a_2^{2'}$, respectively) 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 we consider the execution E_x similarly to E_x^{elimi}, that is, agents b_1 and b'_1, b_2 and b'_2, $\ldots$, b_{h-1} and b'_{h-1} are activated and behave symmetrically in this order, while agents a_1^2 and $a_1^{2'}$ are not activated. Finally, agents b_{h-1}

and b'_{h-1} enter waiting states at v_2 and v'_2, respectively. We call such a configuration c_x.[4] Then in c_x, there exist three waiting agents a^2_1, a^2_2, and b_{h-1} ($a^{2'}_1, a^{2'}_2$, and b'_{h-1}, respectively) at v_2 (v'_2), and agents a^2_2 and b_{h-1} ($a^{2'}_2$ and b'_{h-1}, respectively) never leave the current node by Lemma 4.4.2. For example in Fig. 4.4, agents a^1_1 and $a^{1'}_1$ move symmetrically and enter waiting states at v_3 and v'_3, respectively (Fig. 4.4 (e) to Fig. 4.4 (f)), and after this, agents a^3_1 and $a^{3'}_1$ move symmetrically and enter waiting states at v_2 and v'_2, respectively (Fig. 4.4 (g) to Fig. 4.4 (h)). Then there exist three waiting agents a^2_1, a^2_2, and a^3_1 ($a^{2'}_1, a^{2'}_2$, and $a^{3'}_1$, respectively) at v_2 (v'_2), and agents a^2_2 and a^3_1 ($a^{2'}_2$ and $a^{3'}_1$, respectively) never leave the current node by Lemma 4.4.2. Note that, agents $a^1_2, a^1_3, a^{1'}_2$ and $a^{1'}_3$ also never leave the current node. Thus in c_x, there exist four nodes where agents exist and never leave the current nodes (three-node three-waiting-agent configuration), which is a contradiction.

From Case 4 to Case 6, we consider cases that there exist at least two waiting agents a^1_1 and a^1_2 ($a^{1'}_1$ and $a^{1'}_2$, respectively) at v_1 (v'_1), and there exists at most one waiting agent at the other nodes.

⟨Case 4: $2 \leq N_1 \leq (g+1)/2$ and $N_2 = 1$ hold.⟩

In this case, we consider the initial configuration c^{elimi}_0 obtained from c_0 by eliminating agents $a^1_2, \ldots, a^1_{N_1}, a^{1'}_2, \ldots, a^{1'}_{N'_1}$. Note that, the number of eliminated agents $a^1_2, \ldots, a^1_{N_1}$, $a^{1'}_2, \ldots, a^{1'}_{N'_1}$ is $2N_1 - 2 \leq g-1$ since $N_1 \leq (g+1)/2$ holds. Then from Lemma 4.4.3, there exists an execution E^{elimi}_t from c^{elimi}_0 to c^{elimi}_t, where there exists at most one waiting agent at each node in c^{elimi}_t. This configuration is the same as the Case 2 and agents cannot solve the g-partial gathering problem.

⟨Case 5: $(g+3)/2 \leq N_1 \leq g$ and $N_2 = 1$ hold.⟩

In this case, we consider the initial configuration c^{elimi}_0 obtained from c_0 by eliminating agents $a^1_2, \ldots, a^1_{N_1}$. Note that, the number of eliminated agents $a^1_2 \ldots, a^1_{N_1}$ is $N_1 - 1 \leq g-1$

[4] Execution E_x is fair because the system reaches configuration c_x in finite number of agents' steps. Similarly, we can show that schedulers or executions we consider in the rest of this section are fair.

since $N_1 \leq g$ holds. Then from Lemma 4.4.3, there exists an execution E_t^{elimi} from c_0^{elimi} to c_t^{elimi}, where there exist N'_1 waiting agents at v'_1 and at most one waiting agent at the other nodes in c_t^{elimi}. Since agents are allowed to meet at one or two nodes and only $a_1^{1'}$ can leave the current node in this configuration, it is necessary that agent $a_1^{1'}$ firstly leaves v'_1 and enters a waiting state at some node where a waiting agent exists to make the waiting agent leave there. Without loss of generality, we assume that $a_1^{1'}$ enters a waiting state at v'_j where waiting agent $a_1^{j'}$ exists. We call such a configuration c_x^{elimi} and define E_x^{elimi} as an execution from c_t^{elimi} to c_x^{elimi}. Moreover after this, agents need to make the configuration such that some agent a' enters a waiting state at v_j in order to meet there or make agent a_1^j leave there. We call such a configuration c_y^{elimi} and define E_y^{elimi} as an execution from c_x^{elimi} to c_y^{elimi}. For example in Fig. 4.5, agent $a_1^{1'}$ moves and enters a waiting state at $v_{3'}$ (Fig. 4.5 (a) to Fig. 4.5 (b)), and after this, agent $a_1^{3'}$ moves and enters a waiting state at v_3 (Fig. 4.5 (c)).

Now let us consider c_t. In c_t, agents a_1^1 and $a_1^{1'}$ can leave the current nodes and the other agents cannot leave the current nodes. Then we consider an execution E_x under the fair scheduler α_x, where a_1^1 and $a_1^{1'}$ are activated at the same time, behave symmetrically and enter waiting states at v_j and v'_j, respectively. We call such a configuration c_x. Then, the local configurations of a_1^j and $a_1^{j'}$ change and they can leave v_j and v'_j, respectively. However, we consider the execution E_y similarly to E_y^{elimi}, that is, agent $a_1^{j'}$ leaves v'_j and some agent a' enters a waiting state at v_j, while a_1^j is not activated. Then there exist three waiting agents a_1^j, a_1^1, and a' at v_j, and agents a_1^1 and a' never leave v_j by Lemma 4.4.2. For example in Fig. 4.5, agents a_1^1 and $a_1^{1'}$ move and enter waiting states at v_3 and v'_3, respectively (Fig. 4.5 (d) to Fig. 4.5 (e)), and after this, agent $a_1^{3'}$ leaves v'_3 and enters a waiting state at v_3 (Fig. 4.5 (f)). Then there exist three waiting agents a_1^3, a_1^1, and $a_1^{3'}$ at v_3, and agents a_1^1 and $a_1^{3'}$ never leave v_3. Note that, agents $a_2^1, a_3^1, a_2^{1'}$ and $a_3^{1'}$ also never leave the current node. Thus in c_y, there exist three nodes where agents exist at the end of algorithm execution (three-node three-waiting-agent configuration), which is a contradiction.

$\langle$Case 6: $N_1 \geq g+1$ and $N_2 = 1$ hold.$\rangle$

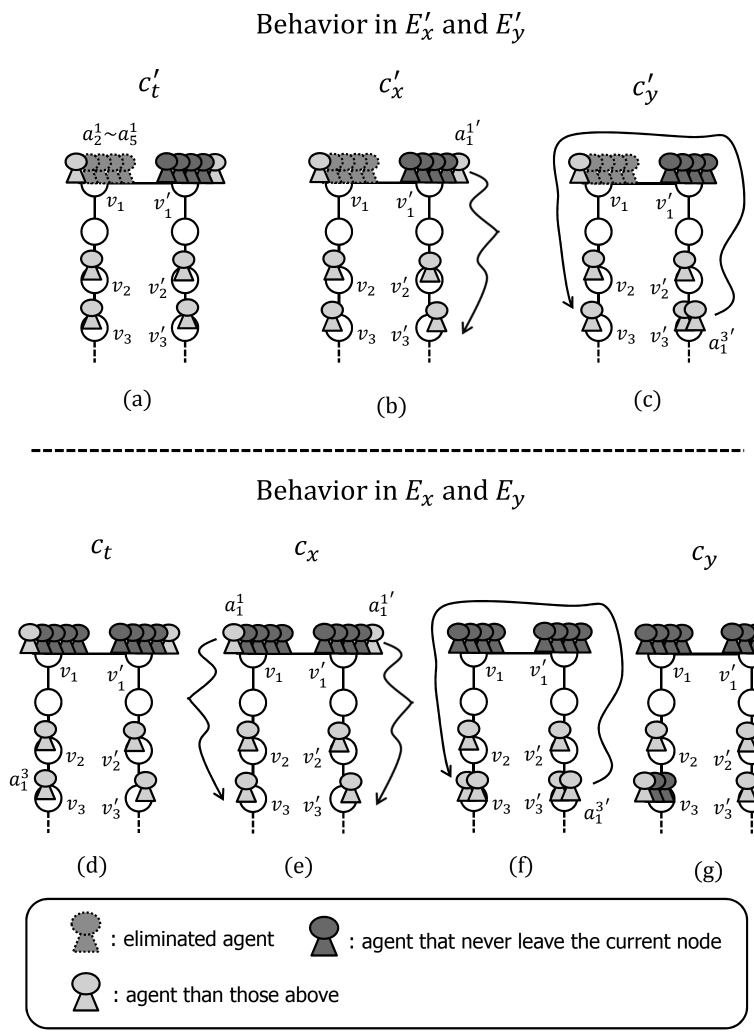

Figure 4.5: An example of Case 5

In this case, agents are allowed to meet at v_1 or v'_1. As a one way to satisfy this, we consider an execution E_x from c_t to c_x, where each agent moves symmetrically until they enter waiting states at v_1 or v'_1 in c_x. Then, there exist $(3g-1)/2$ agents at v_1 and v'_1, respectively.

Now let us consider the initial configuration c_0^{elimi} obtained from c_0 by eliminating agents $a_4^1, \dots, a^1_{4+(g+1)/2-1}$. Then from Lemma 4.4.3, there exists an execution E_t^{elimi}

from c_0^{elimi} to c_t^{elimi}, where there exist $N_1 - (g+1)/2$ waiting agents at v_1, N_1' $(= N_1)$ waiting agents at v_1', and at most one waiting agent at the other nodes in c_t^{elimi}. Moreover we consider the execution E_x^{elimi} similarly to E_x, and we define c_x^{elimi} as the configuration that all agents meet at v_1 or v_1'. Then since $(g+1)/2$ agents $a_4^1, \ldots, a_{4+(g-1)/2-1}^1$ are eliminated, the number of agents that meet at v_1 is $(3g-1)/2 - (g+1)/2 = g-1$. This contradicts that agents can solve the g-partial gathering problem.

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

$\langle$Case 7: $N_1 = N_2 = 2$ and $N_3 = 1$ hold.$\rangle$
In this case, there are two waiting agents at v_1, v_2, v_1', and v_2', and at most one waiting agent at the other nodes in c_t. Now we consider the initial configuration c_0^{elimi} obtained from c_0 by eliminating agents $a_2^1, a_2^2, a_2^{1'}$, and $a_2^{2'}$. Then from Lemma 4.4.3, there exists an execution E_t^{elimi} from c_0^{elimi} to c_t^{elimi}, where there exists at most one waiting agent at each node in c_t^{elimi}. This configuration is the same as the Case 2 and agents cannot solve the g-partial gathering problem.

$\langle$Case 8: $N_1 = N_2 = N_3 = 2$ holds.$\rangle$
In this case, there are two waiting agents at $v_1, v_2, v_3, v_1', v_2'$, and v_3' in c_t. Now we consider the initial configuration c_0^{elimi} obtained from c_0 by eliminating agents $a_2^2, a_2^3, a_2^{2'}$, and $a_2^{3'}$. Then from Lemma 4.4.3, there exists an execution E_t^{elimi} from c_0^{elimi} to c_t^{elimi}, where there exist two waiting agents a_1^1 and a_2^1 ($a_1^{1'}$ and $a_2^{1'}$, respectively) at v_1 (v_1') and one waiting agent at v_2, v_3, v_2', and v_3', respectively. In this configuration, it is necessary that some agent enters a waiting state at v_2, v_3, v_2' and v_3' in order to meet there or to make the waiting agents leave the current nodes. Without loss of generality, we assume that at first some agents enter waiting states at v_2 and v_2', respectively, and after this, some agents enter waiting states at v_3 and v_3', respectively. To do this, we consider an execution E_x^{elimi} under the scheduler α_x^{elimi} similarly to Case 3. That is, there exist the sequence of agents $b_1, \ldots, b_h$ ($b_1', \ldots, b_h'$) such that agent b_h (b_h') is in the waiting state at v_2 (v_2').

Then in α_x^{elimi}, agents b_j and b'_j $(1 \leq j \leq h-1)$ are activated at the same time, behave symmetrically, and enter waiting states at $v_{b(j+1)}$ and $v'_{b(j+1)}$, respectively. Remind that at node $v_{b(j+1)}$, there exists a waiting agent $b_(j+1)$. Then, local configurations of agents b_{j+1} and b'_{j+1} change. Finally, agents b_{h-1} and b'_{h-1} enter waiting states at v_2 and v'_2, respectively, and we call such a configuration c_x^{elimi}. Then, local configurations of a_1^2 and $a_1^{2'}$ change and they can leave the current nodes. For example in Fig. 4.6, agent a_1^1 $(a_1^{1'})$ leaves at v_1 (v'_1) and directly enters a waiting state at v_2 (v'_2) (Fig. 4.6 (a) to Fig. 4.6 (b)). Moreover after c_x^{elimi}, we consider an execution E_y^{elimi} under the scheduler α_y^{elimi} similarly to α_x^{elimi}, that is, there exists the sequence of agents $d_1, \ldots, d_i$ $(d'_1, \ldots, d'_i)$ such that agent d_i (d'_i) is in the waiting state at v_3 (v'_3). Then in α_y^{elimi}, agents d_j and d'_j $(1 \leq j \leq i-1)$ are activated at the same time, behave symmetrically, and enter waiting states at $v_{d(j+1)}$ and $v'_{d(j+1)}$, respectively. Note that at node $v_{d(j+1)}$, we assume that there exists a waiting agent d_{j+1}. Then, local configurations of agents d_{j+1} and d'_{j+1} change. Finally, agents d_{i-1} and d'_{i-1} enter waiting states at v_3 and v'_3, respectively, and we call such a configuration c_y^{elimi}. For example in Fig. 4.6, agent a_1^2 $(a_1^{2'})$ leaves v_2 (v'_2) and directly enters a waiting state at v_3 (v'_3) (Fig. 4.6 (b) to Fig. 4.6 (c)).

Now let us consider c_t. In c_t, agents $a_1^1, a_1^2, a_1^3, a_1^{1'}, a_1^{2'}$ and $a_1^{3'}$ can leave the current nodes. However we consider the execution E_x similarly to E_x^{elimi}, that is, agents b_1 and b'_1, b_2 and b'_2, ... , b_{h-1} and b'_{h-1} are activated and behave symmetrically in this order, while agents a_1^2 and $a_1^{2'}$ are not activated. Finally, agents b_{h-1} and b'_{h-1} enter waiting states at v_2 and v'_2, respectively. We call such a configuration c_x. Then there exist three waiting agents a_1^2, a_2^2, and b_{h-1} $(a_1^{2'}, a_2^{2'}$, and b'_{h-1}, respectively) at v_2 (v'_2), and a_2^2 and b_{h-1} $(a_2^{2'}$ and b'_{h-1}, respectively) never leave the current node. For example in Fig. 4.6, agent a_1^1 $(a_1^{1'})$ leaves v_1 (v'_1) and directly enters a waiting state at v_2 (v'_2) (Fig. 4.6 (d) to Fig. 4.6 (e)). Then there exist three waiting agents a_1^2, a_2^2, and a_1^1 $(a_1^{2'}, a_2^{2'}$, and $a_1^{1'}$, respectively) at v_2 (v'_2), and a_2^2 and a_1^1 $(a_2^{2'}$ and $a_1^{1'}$, respectively) never leave the current node. Moreover after this, we consider the execution E_y similarly to E_y^{elimi}, that is, agents d_1 and d'_1, d_2 and d'_2, ... , d_{i-1} and d'_{i-1} are activated and behave symmetrically in this order, while agents a_1^3 and $a_1^{3'}$ are not activated. Finally, agents b_{i-1} and b'_{i-1} enter waiting states at v_3 and v'_3, respectively. We call such a configuration c_y. Then

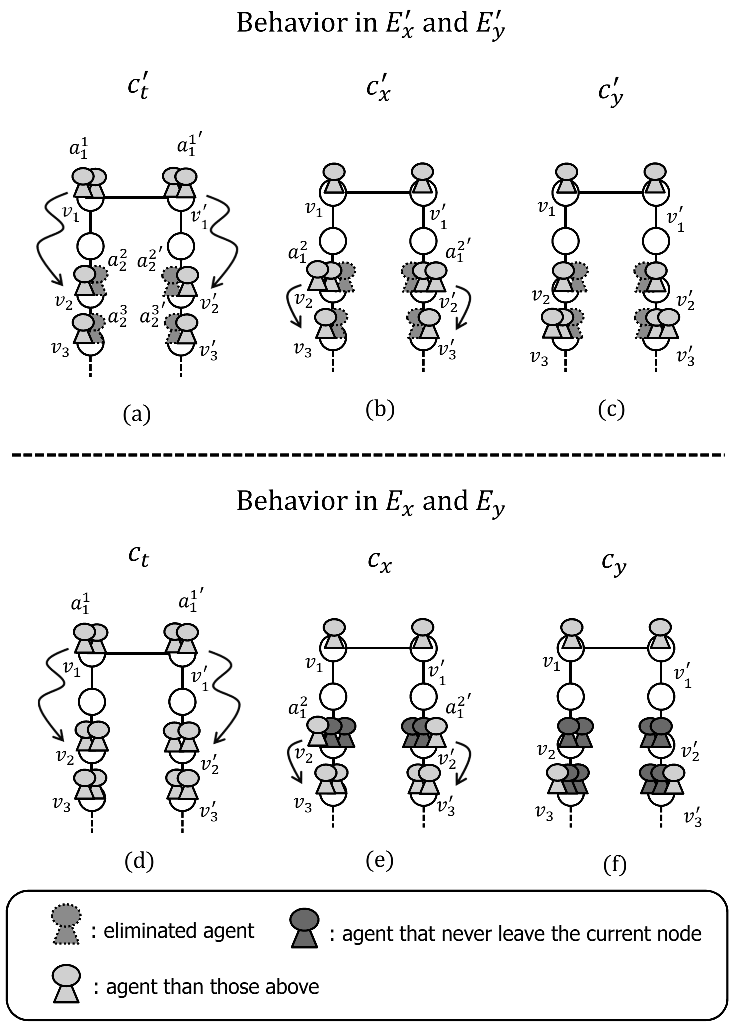

Figure 4.6: An example of Case 8

there exist three waiting agents a_1^3, a_2^3, and d_{i-1} ($a_1^{3'}, a_2^{3'}$, and d'_{i-1}, respectively) at v_3 (v'_3), and a_2^3 and d_{i-1} ($a_2^{3'}$ and d'_{i-1}, respectively) never leave the current node. For example in Fig. 4.6, agent a_1^2 ($a_1^{2'}$) leaves v_2 (v'_2) and directly enters a waiting state at v_3 (v'_3) (Fig. 4.6 (e) to Fig. 4.6 (f)). Then there exist three waiting agents a_1^3, a_2^3, and a_1^2 ($a_1^{3'}, a_2^{3'}$, and $a_1^{2'}$, respectively) at v_3 (v'_3), and a_2^3 and a_1^2 ($a_2^{3'}$ and $a_1^{2'}$, respectively) never leave the current node. Thus in c_y there exist four nodes where agents exist at the end of algorithm execution (three-node three-waiting-agent configuration). This contradicts that agents can solve the g-partial gathering problem.

Therefore, we have the theorem. □

4.5 Strong Multiplicity Detection and Non-Token Model

In this section, we consider a deterministic algorithm to solve the g-partial gathering problem for Model 2 in Table 4.1, that is, the strong multiplicity detection and non-token model. We propose a deterministic algorithm to solve the g-partial gathering problem in $O(kn)$ total moves. Recall that, in the strong multiplicity detection model, each agent can count the number of agents at the current node.

At the beginning, each agent performs *a basic walk* [46]. 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 of v_j, a_h leaves v_j through the port $(p+1) \mod d_{v_j}$. The basic walk allows each agent to traverse the tree in the DFS-traversal. Hence, when each agent visits nodes $2(n-1)$ times, it visits all the nodes and returns to the initial node. Remind that nodes are anonymous and agents do not know the number n of nodes. However, if an agent records the topology of the tree it ever visits, it can detect that it visits all the nodes and returns to the initial node. Concretely, in the DFS-traversal, if agent a_h visits some node far (resp., closer) from its initial node, it memorizes "+" (resp., "−"). When the number of "+" and "−" that a_h ever memorized are the same, it can recognize that it returns to its initial node. Moreover, if there exists no port p incident to its initial node such that a_h does not leave its initial node through p, it can detect that it observed all the nodes in the tree.

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.

Algorithm 4.1 The behavior of active agent a_h (v_j is the current node of a_h.)

Main Routine of Agent a_h

```
perform 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 − 1 agents except for a_h then
        terminate the algorithm
    else
        move to the other center node
        terminate the algorithm
    end if
end if
```

If there exist at most $g-1$ agents except for 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 4.1. We have the following theorem.

Theorem 4.5.1. *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 4.1, 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 the center nodes. If there already exist g or more agents at the center node, a_h moves to the other center node. Since $k \geq 2g$ holds, agents can solve the g-partial gathering problem.

Next, we analyze the total number of moves. 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 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. Therefore, agents require $O(kn)$ total moves. □

4.6 Weak Multiplicity Detection and Removable-Token Model

In this section, we consider the g-partial gathering problem for Model 3 in Table 4.1, that is, the weak multiplicity detection and removable-token model. We show that agents can achieve the g-partial gathering in asymptotically optimal total moves (i.e., $O(gn)$) by using only one removable token of each agent. Recall 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 Chapter 3.3, which considers the g-partial gathering problem for distinct agents (i.e. having IDs) in unidirectional ring networks with whiteboards. In Chapter 3.3, agents execute the leader agent election algorithm partially, and then leader agents instruct non-leader agents which node they should meet at. When applying the above idea in Chapter 3.3 to the model in this section, there exist two problems. The first is the difference of network topology, that is, Chapter 3.3 considers unidirectional ring networks but in this paper we consider tree networks. The second is the difference of agents' and nodes' ability, that is, in Chapter 3.3 agents have distinct IDs and each node has a whiteboard but in this paper agents have no IDs and each node is allowed to only have at most one removable token. The first problem is solved by embedding the unidirectional ring in the tree network, and we explain this in the next paragraph. The second problem is solved by the combination of port numbers and removable-tokens, and we explain this in Section 4.6.1 and 4.6.2.

Now, we explain the way to embed the ring from the tree network. Agents perform the basic walk and embed a unidirectional ring network in the tree network by the Euler tour technique. Concretely, letting $v_1, v_2, \ldots, v_{2(n-1)}$ be the node sequence such that agent a_h visits the nodes in this order in the basic walk starting at v_0, we can regard 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. For simplicity in the virtual ring, operations to an index of a node assume calculation under modulo $2(n-1)$, that is, $v_{(i+1) \bmod 2(n-1)}$ is simply represented by v_{i+1}. In addition in the virtual ring, we define *the neighboring agent* of a_h as the first agent in a_h's forward (backward) direction, i.e., there exist no agents between them. Moreover, when a_h visits a node v_j through a port p of v_j from a node v_{j-1} in the virtual ring, agents also use p as the port number of (v_{j-1}, v_j) at v_j. For example, let us consider a tree in Fig. 4.7 (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. 4.7 (a) is shown in Fig. 4.7 (b). Each number in Fig. 4.7 (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. At the beginning of the algorithm, each agent a_h leaves its token node through the port 0 in the basic walk. Thus, when a_h visits some token node in the tree such that a_h leaves there through the port 0 in the next movement, that is, when a_h visit some token node v_j through the port $(d_{v_j} - 1)$, a_h regards the node as the token node in the virtual ring. In Fig. 4.7 (a), if nodes a and b are token nodes, then in Fig. 4.7 (b), nodes a and b'' are token nodes. By this definition, a token node in the tree network is mapped to exactly one token node in the virtual ring. Thus, by performing the basic walk, we can regard that all agents move in the same virtual ring although agents start the algorithm at different nodes. This is because the virtual ring starting at some node in the tree is actually represented by a port sequence P, and the virtual ring starting at other nodes in the same tree can be represented by the lexicographically transformation of P. In Fig. 4.7, the virtual ring starting at a_h's initial node is represented by 001020. On the other hand, the virtual ring starting at another token node is represented by 000102, and this sequence can be also represented by the lexicographically transformation of 001020. 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, a_h arrives at v_{j+1} before a_i.

In the following section, we explain the algorithm on the virtual ring. Note that we can show the asymptotically equivalence in terms of total moves between a tree and a virtual ring, because a tree with n nodes is regarded as a virtual ring with $2n-1$ nodes. The algorithm consists of two parts. In the first part, agents elect some leader agents by

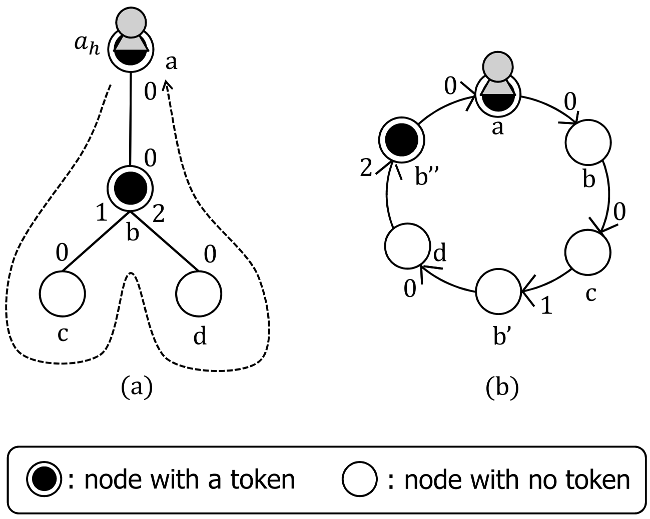

Figure 4.7: An example of the basic walk

partially executing the leader agent election algorithm. In the second part, the leader agents instruct the other agents which node they should meet at, and the other agents move to the node.

4.6.1 The first part: leader election

In this section, we explain how to elect multiple leader agents. Note that, in this part no token is removed. In the leader agent election, each agent takes a state from the following three states:

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

The aim of the first part is similar to Chapter 3.3, that is, to elect some leaders and satisfy the following two properties: 1) At least one agent is elected as a leader, and 2) in the virtual ring, there exist at least $g - 1$ inactive agents between two leader agents.

In the following, we explain the way to apply the idea of the leader election using distinct IDs of agents and whiteboards of nodes in Chapter 3.3 to anonymous agents in the weak multiplicity detection and removable-token model. 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 at v_{j+m} through which a_h visits m-th node v_{j+m} 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. 4.7 (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 of whiteboards by using removable tokens. Fortunately, we can easily overcome this problem if agents can detect active nodes. 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 for a unidirectional ring, that is, a_h behaves as if it would be an active agent with ID id_2 in a bidirectional ring. 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 set of leader candidates 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 where no agents exist, a_h knows that a_h visits an active node and the other nodes are not active 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 for leaders) in the next phase or not. Different from Chapter 3.3, 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 for leaders. Otherwise, a_h becomes inactive and drops out from the set of leader candidates. For example, let us consider the initial configuration like Fig. 4.8 (a). In the figure, black nodes are token nodes and the numbers near communication links are port numbers. The virtual ring of Fig. 4.8 (a) is shown in Fig. 4.8 (b). For simplicity, we omit non-token nodes in Fig. 4.8 (b). The numbers in Fig. 4.8 (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 for leaders, and a_1, a_2, and a_3 drop out from the set of leader candidates. Note that, like Fig. 4.8, if an agent observes the same virtual IDs three times, it drops out from the set of leader candidates. 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. Thus in each phase, at least the half of active agents become inactive, and we show this later (Lemma 4.6.2). 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 4.2. 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 inactive or a leader, a_h immediately moves to the next part and executes the algorithm for an inactive state or a leader state in Section 4.6.2. In Algorithm 4.2, a_h uses the following variables:

- id_1, id_2, and id_3 are variables for storing three virtual IDs.
- *phase* is a variable for storing its own phase number.

In Algorithm 4.2, each active agent a_h moves until a_h observes three virtual IDs and decides whether a_h remains active as a candidate for leaders or not on the basis of virtual IDs. Note that, since each agent moves in a FIFO manner, it does not happen

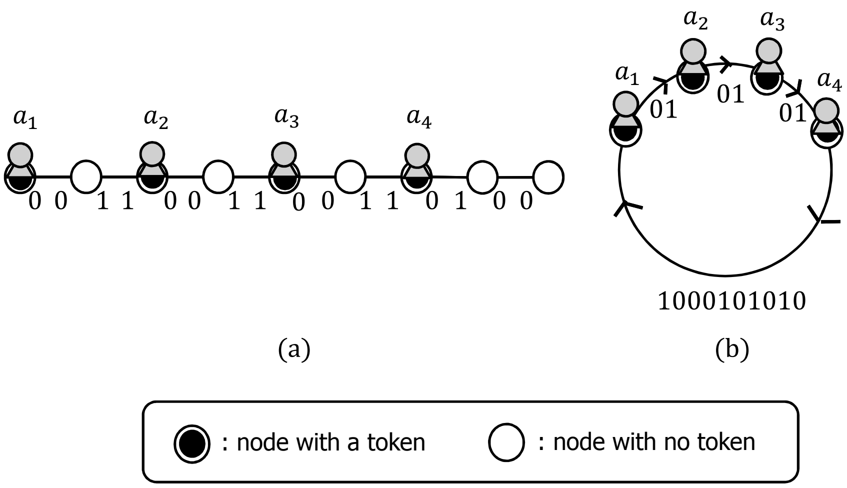

Figure 4.8: An example that agents observe the same port sequence

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 4.2, 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 Procedure 4.1. In *NextActive*, a_h uses the following variables:

- *port* is an array for storing a virtual ID.
- *move* is a variable for storing the number of nodes it visits.

During the basic walk, each active agent visits active node v_j through the port $(d_{v_j} - 1)$. Thus, when agent a_h leaves active node v_j, it always uses the port 0 and leaves there (line 2 in Procedure 4.1).

Note that, if there exist only one or two active agents in some phase, then the agent travels once 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 (lines 5 to 8 in Algorithm 4.2). To do this, agents record the topology every time they visit nodes, but we omit the description of this behavior in Algorithm

Algorithm 4.2 The behavior of active agent a_h (v_j is the current node of a_h.)

Variables for Agent a_h

int $phase = 0$;

int id_1, id_2, id_3;

Main Routine of Agent a_h

```
phase = phase + 1
id_1 = NextActive()
id_2 = NextActive()
id_3 = NextActive()
if the number of active agent in the tree is two or less then
    change its state to a leader state
    break Algorithm 4.2
end if
if (id_2 < min(id_1, id_3)) ∨ (id_2 = id_3 < id_1) then
    if (phase = ⌈log g⌉) then
        change its state to a leader state
        break Algorithm 4.2
    else
        go to line 1
    end if
else
    change its state to an inactive state
end if
```

4.2 and Procedure 4.1.

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

Lemma 4.6.1. *When there exist three or more active agents, there exist two active agents having different virtual IDs.*

Proof. To show the lemma, we use the theorem from [15]. Let $t[1..q]$ be a port sequence

Procedure 4.1 int *NextActive*() (v_j is the current node of a_h.)

Variables for Agent a_h

array *port*[ ];

int *move*;

Behavior 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*[ ]

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 gets in traverse from an active node to the next active node. Moreover, $(t[1..q])^k$ denotes the concatenation of k copies of $t[1..q]$. If $t[1..q] = (t[1..q'])^k$ holds some positive integers q' and k ($q' < q$), we call $t[1..q]$ is periodic. Otherwise, we call $t[1..q]$ is not periodic. 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 4.6.1. [15] *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.*

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. Then, $t[1..|x|] = x$ holds. In addition, when each active agent moves in the tree and observes one virtual ID x, each link in the virtual link is passed by exactly once. Hence, $t[(\ell|x|+1)..(\ell+1)|x|] = x$ holds $(0 \leq \ell \leq k'-1)$ and $t[1..k'|x|] = (t[1..|x|])^{k'}$ holds. Moreover, in this case the total number of their moves (i.e., $k'|x|$) is equal to the length of the tree. If x is not periodic, the length of the tree is $k'|x|$. However from Theorem 4.6.1, the length of the tree is never $k'|x|$, which is a contradiction. If x is periodic, $t[1..|x|] = (t[1..|x'|])^s$ holds for some x' and s (x' is not periodic). Then, $t[1..k'|x|] = t([1..|x'|])^{k's}$ holds and the length of the tree is $k's|x'|(= k'|x|)$. However, from Theorem 4.6.1, the length of the tree is never $k's|x'|$, which is also a contradiction. □

Next, we have the following lemmas about Algorithm 4.2.

Lemma 4.6.2. *Algorithm 4.2 eventually terminates, and satisfies the following two properties.*

- *There exists at least one leader agent.*

- *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 4.2 eventually terminates. In the following, we show the above two properties.

At first, we show that there exists at least one leader agent. From lines 5 to 7 of Algorithm 4.2, when there exist only one or two active agents in some phase, the agents become leaders. We assume that in some phase, active agent a_h observes three IDs $a_h.id_1, a_h.id_2$, and $a_h.id_3$ in this order. When there are three or more 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 for leaders, and otherwise a_h drops out from the set of leader candidates. Thus, unless all agents observe the same virtual IDs, at least one agent remains active as a candidate for leaders. From Lemma 4.6.1, 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 least $g-1$ inactive agents between two leader agents in the virtual ring. At first, we show that in each phase, at least half of active agents become inactive. 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 for 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 set of leader candidates. In the following, we 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 set of leader candidates. 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 for leaders and drops out from the candidate. Thus in each phase, at least half of active agents drop out from the set of leader candidates and become inactive. Now, we show that there exist at least $g-1$ inactive agents between two leader agents. We firstly 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 of $j=1$, there exists at least $2^1-1=1$ inactive agent between two active agents as mentioned above. For the case of $j=k$, 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 4.6.3. *Algorithm 4.2 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. □

4.6.2 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 the elected agents. 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, each agent takes one of the following three states:

- *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 Chapter 3.3, that is, to divide agents into groups each of which consists of at least g agents. While in Chapter 3.3, each node has a whiteboard, in this section each node is allowed to only have a removable token. Each leader agent a_h moves to the next leader node, and during the movement 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. The behavior guarantees that at least $g-1$ agents exist between any two token nodes when all the leaders complete the behavior. After that, agents move to the nearest token nodes, which guarantees that at least g agents meet at each token node.

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 (including the current node) 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 $v_{j'}$ (Later, explain how a_h detects whether it visits the next leader node $v_{j'}$ or not). After that, a_h removes a token at $v_{j'}$. When all the leaders complete this behavior, there exist at least $g-1$ inactive agents between two token nodes. Hence, agents solve the g-partial gathering problem by moving to the nearest token node (This is done by changing their states to moving states). For example, let us consider the configuration like Fig. 4.9 (a) ($g = 3$). We assume that a_1 and a_2 are leader agents and the other agents are inactive agents. In Fig. 4.9 (b), a_1 visits node v_2 and a_2 visits node v_4, respectively. The number near each node represents the number (modulo g) of inactive nodes that a_1 or a_2 has ever visited. Then, 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. 4.9 (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. 4.9 (c) to Fig. 4.9 (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 4.3. Variable $tCount$ is used to count the number of inactive nodes a_h has ever visited. When a_h visits a token node v_j where another agent exists, 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 $tCount \neq g-1$ (does not remove a token otherwise) and continues to move (lines 5 to 9). This guarantees that, if a token at inactive node v_j is not removed,

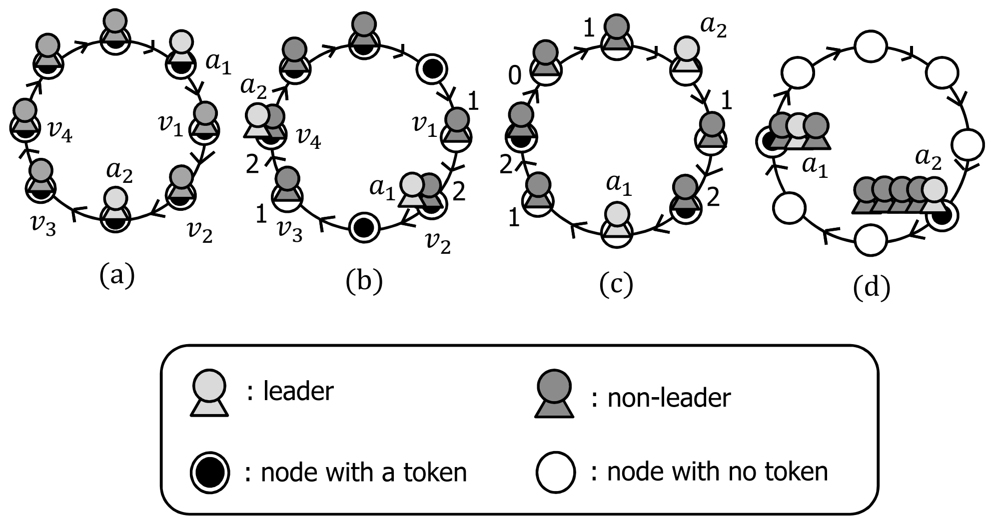

Figure 4.9: Partial gathering in the removable-token model for the case of $g = 3$ (a_1 and a_2 are leaders, and black nodes are token nodes)

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 (line 7). When a_h visits a token node $v_{j'}$ where no agents exist, $v_{j'}$ is the next leader node. This is because token nodes are leader nodes or inactive nodes, and from an atomicity of the execution there exist no agents at each leader node. Note that also from an atomicity of the execution, it does not happen that some leader agent visits a leader node v such that another agent becomes a leader at v but still stays at v. 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 4.3, a_h uses the procedure *NextToken*() to move to the next token node. The pseudocode of *NextToken*() is described in Procedure 4.2. In Procedure 4.2, a_h performs the basic walk until a_h visits a token node v_j through the port $(d_{v_j} - 1)$.

The pseudocode of inactive agents is described in Algorithm 4.4. 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 (lines 4 to 6). If a_h observes another agent, the agent is a moving agent and terminates the algorithm at v_j (lines 7

Algorithm 4.3 The behavior of leader agent a_h (v_j is the current node of a_h)

Variable in Agent a_h

int $tCount = 0$;

Main Routine of Agent a_h

```
1:  NextToken()
2:  while there exists another agent at v_j do
3:      //this is an inactive node
4:      tCount = (tCount + 1) mod g
5:      if tCount ≠ g − 1 then
6:          remove a token at v_j
7:          //an inactive agent at v_j changes its state to a moving state
8:      end if
9:      NextToken()
10: end while
11: remove a token at v_j
12: change its state to a moving state
```

to 9). 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 4.5. In the virtual ring, each moving agent a_h moves to the nearest token node by using *NextToken*().

We have the following lemma about the algorithms.

Lemma 4.6.4. *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 $v_0^g, v_1^g, \ldots, v_l^g$ be inactive nodes that still have tokens after all leader agents complete their behaviors, and we call these nodes *gathering nodes*. From Algorithm 4.3, each leader agent a_h removes the tokens at the consecutive $g-1$ inactive nodes and does not remove the token at the next inactive node. By this behavior and Lemma 4.6.2, there exist at least $g-1$ moving agents between v_i^g and v_{i+1}^g. Moreover,

Procedure 4.2 void *NextToken*() (v_j is the current node of a_h.)

leave v_j through the port 0
let p be the port number through which a_h visits v_j
while (there does 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 4.4 The behavior of inactive agent a_h (v_j is the current node of a_h)

Main Routine of Agent a_h

while (there does 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 4.5 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

these moving agents move to the nearest gathering node v^g_{i+1}. Therefore, agents solve the g-partial gathering problem.

In the following, we evaluate the total number of moves required for the algorithms. At first, let us consider the total number of moves required for leader agents to move to the next leader nodes. This requires $2(n-1)$ total moves since all leader agents travel once around the virtual ring. Next, let us consider the total number of moves required for moving (inactive) agents to move to the nearest token nodes (For example, the total number of moves form Fig. 4.9 (c) to Fig. 4.9 (d)). From Algorithm 4.5, each moving agent moves to the nearest gathering node. In the following, we show that the number of moving agents between some gathering node v_i^g and its forward gathering node v_{i+1}^g is $O(g)$. From Algorithm 4.3, the moving agents between v_i^g and v_{i+1}^g consist of inactive agents and leader agents between v_i^g and v_{i+1}^g. Since there exists at least one gathering node between two leader nodes, there exists at most one leader node between v_i^g and v_{i+1}^g. If there exist no leader node between v_i^g and v_{i+1}^g, then clearly there exist $g-1$ inactive nodes between v_i^g and v_{i+1}^g. If there exists one leader node v_l between v_i^g and v_{i+1}^g, there exist at most $g-1$ inactive nodes between v_i^g and v_l, and at most $g-1$ inactive nodes between v_l and v_{i+1}^g, respectively. Thus, there exist at most $O(g)$ moving agents between gathering nodes v_i^g and v_{i+1}^g, and the total number of moves required for moving (inactive) agents to move to the nearest gathering nodes is at most $O(gn)$ since each communication link is passed by at most $O(g)$ times.

Therefore, we have the lemma. □

From Lemma 4.6.3 and Lemma 4.6.4, we have the following theorem.

Theorem 4.6.2. *In the weak multiplicity detection and the removable-token model, our algorithm solves the g-partial gathering problem in $O(gn)$ total moves.*

4.7 Concluding Remarks

In this chapter, we considered the g-partial gathering problem in asynchronous tree networks. At first, in the non-token model we showed that agents require $\Omega(kn)$ total moves to solve the g-partial gathering problem. After this, we considered three model variants. First, in the weak multiplicity detection and non-token model, for asymmetric trees agents can solve g-partial gathering problem in $O(kn)$ total moves from the past result, and we showed that there exist no algorithms to solve the g-partial gathering problem for symmetric trees. Second, in the strong multiplicity detection and non-token model, we proposed a deterministic algorithm to solve the g-partial gathering problem in $O(kn)$ total moves. Finally, in the weak multiplicity detection and removable-token model, we proposed a deterministic algorithm to solve the g-partial gathering problem in $O(gn)$ total moves.

Chapter 5

Uniform Deployment in Ring Networks

5.1 Introduction

In this chapter, we present algorithms to achieve the uniform deployment in asynchronous unidirectional rings. In [49, 50, 51], the uniform deployment problem is considered under the assumption that agents are oblivious (or memoryless) but can observe multiple node within its visibility range. This assumption is often called a *Look-Compute-Move* model. In this chapter, we assume agents that have memory but cannot observe nodes except for their currently visiting nodes. To our best knowledge, this is the first research considering the uniform deployment for such agents.

5.1.1 Contribution

Contributions of this paper are summarized in Table 5.1. We assume that each agent initially has a token and can release it on a visited node. After a token is released at some node, agents cannot remove the token. In addition, we assume that agents can send a message of any size to agents at the same node. We consider two problem settings. First, we consider agents with knowledge of k, where k is the number of agents. In this case, we propose two algorithms. The first algorithm solves the uniform deployment with

Table 5.1: Results in each model

	Result 1 (Section 5.3.1)	Result 2 (Section 5.3.2)	Result 3 (Section 5.4.1)	Result 4 (Section 5.4.2)
Knowledge of k	Available	Available	Not Available	Not Available
Termination detection	Required	Required	Required	Not Required
Solvable / Unsolvable	Solvable	Solvable	Not Solvable	Solvable
Agent memory	$O(k \log n)$	$O(\log n)$	-	$O((k/l) \log(n/l))$
Time complexity	$O(n)$	$O(n \log k)$	-	$O(n/l)$
Total agent moves	$O(kn)$	$O(kn)$	-	$O(kn/l)$

n: number of nodes, k: number of agents, l: symmetry degree of the initial configuration

termination detection. This algorithm requires $O(k \log n)$ memory space per agent, $O(n)$ time, and $O(kn)$ total moves, where n is the number of nodes. The second algorithm also solves the uniform deployment problem with termination detection. This algorithm reduces the memory space per agent to $O(\log n)$, but allows $O(n \log k)$ time, and requires $O(kn)$ total moves. Note that agents require $\Omega(kn)$ total moves to solve the problem. Hence, we can show that the both proposed algorithms are asymptotically optimal in terms of total moves.

Next, we consider agents with no knowledge of k or n. In this case, we show that, when termination detection is required, there exists no algorithm to solve the uniform deployment problem. Intuitively, it is due to impossibility of finding k or n when some part of the initial configuration has symmetry: when an agent misestimates these at smaller numbers than actual ones, it prematurely terminates and the uniform deployment cannot be achieved. For this reason, we consider the relaxed uniform deployment problem that does not require termination detection, and we propose an algorithm to solve the relaxed uniform deployment problem. In this algorithm, each agent estimates k and n (possibly at smaller values than actual ones) and behaves based on the estimation. Thus, the efficiency of the algorithm depends on the estimation. To evaluate the efficiency, we introduce the following parameter l to denote the symmetry degree of an initial configuration: we say that an initial configuration has symmetry degree l

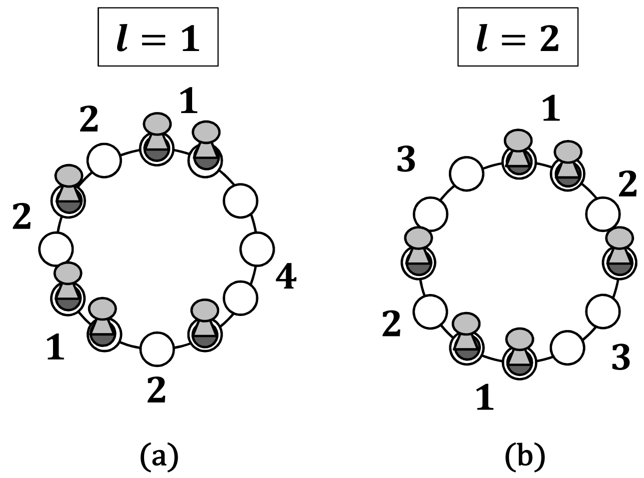

Figure 5.1: An example of the symmetry degree

when its distance sequence can be represented as l-times repetition of some aperiodic sequence. For example, the initial configuration in Fig. 5.1 (a) has symmetry degree 1 since its whole distance sequence (1,4,2,1,2,2) is aperiodic, and the initial configuration in Fig. 5.1 (b) has symmetry degree 2 since its whole distance sequence (1,2,3,1,2,3) is represented as 2-times repetition of aperiodic sequence (1,2,3). Hence, the symmetry degree becomes larger for a higher symmetric initial configuration. Note that agents cannot know l but the efficiency depends on it. Using the symmetry degree parameter l, the efficiency of the algorithm is denoted as follows: this algorithm requires $O((k/l)\log(n/l))$ memory space per agent, $O(n/l)$ time, and $O(kn/l)$ total moves. At fist glance, the upper bound $O(kn/l)$ of the total moves may seem to violate the lower bound $\Omega(kn)$ of the total moves. However, for some initial configuration with $l \geq 2$, the location is closer to the uniform deployment configuration and agents require less than $\Omega(kn)$ total moves to solve the problem. Hence, from such initial configurations agents can make adaptive movement and can solve the problem in less than $\Omega(kn)$ total moves. Thus, the algorithm achieves the uniform deployment more efficiently when the initial configuration has higher symmetry degree. This is a natural but interesting property. For example, for an asymmetric initial configuration this algorithm requires $O(k\log n)$ memory space per agent, $O(n)$ time, and $O(kn)$ total moves. However, when l is $\omega(1)$, this algorithm requires $o(k \log n)$ memory space per agent, $o(n)$ time, and $o(kn)$ total moves. When l is $\Omega(n)$, this algorithm requires $O(1)$ memory space per agent, $O(1)$ time, and $O(k)$ total moves.

Note that, for any initial configuration such that all agents are in the initial states and placed at the distinct nodes, all proposed algorithms achieve the uniform deployment, which is a striking difference from the total gathering problem because the total gathering problem is not solvable from some initial configurations. Note that agents can attain this solvability since the uniform deployment problem requires no symmetry breaking.

5.1.2 Related works

There are several researches considering the uniform deployment problem in a Look-Compute-Move model. Flocchini et al. [49] considered it in a cycle environment of length m (m is a real number). They considered the two types of uniform deployment: *exact* and *ϵ-approximate*. In the exact uniform deployment, agents move in the ring so that the distance between any two consecutive agents is the same, say d. In the ϵ-approximate uniform deployment, agents move in the ring so that the distance is between $d - \epsilon$ and $d + \epsilon$. They showed that if agents do not have common sense of direction, agents cannot solve the exact uniform deployment problem even if agents have unlimited memory and visibility range. If agents have common sense of direction, they proposed an algorithm to solve the exact uniform deployment problem for agents with knowledge of d. In addition, for any $\epsilon > 0$ they proposed an algorithm to solve the ϵ-approximate uniform deployment problem for agents without knowledge of d. Elor et al. [50] considered the uniform deployment also in the ring networks. They considered agents without knowledge k or n, but with visibility range VR. They considered a semi-synchronous model, that is, a subset of all agents execute a behavior in each round. They showed that, if $VR < \lfloor n/k \rfloor$ holds, agents cannot solve the uniform deployment problem. If $VR \geq \lfloor n/k \rfloor$ holds, they proposed an algorithm to solve the balanced uniform deployment problem without quiescence. That is, agents eventually satisfy the condition of the uniform deployment and continue to move in the ring satisfying the condition. In addition, they proposed an algorithm to solve the semi-balanced uniform deployment problem with quiescence. That is, agents eventually terminate the algorithm satisfying the condition such that the distance between any two adjacent agents is between $n/k - k/2$ and $n/k + k/2$. On the other hand, Barriere et al. [51] considered the uniform deployment in the grid networks and proposed an algorithm to achieve the uniform deployment in $O(n/d)$ time, where d is the interval of the uniform deployment.

5.1.3 Organization

The Chapter is organized as follows. In Section 5.3 we consider agents with knowledge of k. In Section 5.4 we consider agents with no knowledge of k or n. Section 5.5 concludes this chapter.

5.2 Preliminary

5.2.1 System Model

In this chapter, we restrict the network topology only to ring networks. We use the same definition of a ring $R = (V, L)$ as in Section 3.2.1. In this chapter, we assume that whiteboards are allowed to have only tokens. We define T as a set of all states (i.e., number of tokens) of a node.

5.2.2 Agent Model

We consider two problem settings: agents with knowledge of k and agents with no knowledge of k or n. We assume that each agent initially has a *token* and can release it on a node that it is visiting. The token on an agent or a node can be realized in one bit that denotes existence of the token, and thus, the token cannot carry any additional information. Note that if agents are not allowed to have tokens, they cannot mark nodes in any way and this means that the uniform deployment problem cannot be solved. This is because if all agents move in a synchronous manner, they cannot get any information of other agents. After a token is released at some node, agents cannot remove the token.

Note that since agents are anonymous, they cannot recognize the owner of each token. In addition, we assume that agents can send a message of any size to agents at the same node. Similarly to Section 4.2.2, we assume that agents move through a link in a FIFO manner. Each agent a_h executes the following five operations in an atomic step: 1) The agent reaches a node v (when a_h is in transit toward v), or it starts operations at v (when a_h is at v), 2) the agent receives all the messages (if any), 3) the agent executes local computation, 4) the agent broadcasts a message to all the agents staying at the same node v (if any) if it decides to send a message, and 5) the agent leaves v if it decides to move. After taking an atomic step, a_h has no message.

5.2.3 System Configuration

In this chapter, a (global) configuration c is defined as a 5-tuple $c = (S, T, M, P, Q)$ and the correspondence table is given in Table 5.2. The first element S is a k-tuple $S = (s_0, s_1, \dots, s_{k-1})$, where s_i is the state (including the state to denote whether it holds a token or not) of agent a_i $(0 \leq i \leq k-1)$. The second element T is an n-tuple $T = (t_0, t_1, \dots, t_{n-1})$, where t_i is the state (i.e., the number of tokens) of node v_i $(0 \leq i \leq n-1)$. The third element M is a k-tuple $M = (m_0, m_1, \dots m_{k-1})$, where m_i is a sequence of messages reached a_i but not consumed yet by a_i. The remaining elements P and Q represent the positions of agents. The element P is an n-tuple $P = (p_0, p_1, \dots, p_{n-1})$, where p_i is a sequence of agents staying at node v_i $(0 \leq i \leq n-1)$. The element Q is an n-tuple $Q = (q_0, q_1, \dots, q_{n-1})$, where q_i is a sequence of agents residing in the FIFO queue corresponding to link (v_{i-1}, v_i) $(0 \leq i \leq n-1)$. Hence, agents in q_i are those in transit from v_{i-1} to v_i.

In initial configuration $c_0 \in C$, we assume that no node has any token. In addition, in c_0 the node where agent a stays is called the *home node* of a and denoted by $v_{HOME}(a)$. We assume that in c_0 agent a is stored at a buffer of its home node $v_{HOME}(a)$. This assures that agent a starts the algorithm at $v_{HOME}(a)$ before any other agent visits $v_{HOME}(a)$, that is, a is the first agent that takes an action at $v_{HOME}(a)$. Next, we define symmetry degree l more precisely. For periodic rings, that is, for rings such that $\mathit{shift}(D_0, x) = D_0$ holds for some x $(0 < x < k)$, we define $l = n/k$. For aperiodic rings, we define $l = 1$.

Table 5.2: Meaning of each element in configuration $c = (S, T, M, P, Q)$

Element	Meaning and example
$S = (s_0, s_1, \ldots, s_{k-1})$	Set of agent states (s_i: the state of agent a_i)
$T = (t_0, t_1, \ldots, t_{n-1})$	Set of node states (t_i: the state of node v_i)
$M = (m_0, m_1, \ldots, m_{k-1})$	Set of message sequences (m_i: a sequence of massages sent to a_i and not received by a_i)
$P = (p_0, p_1, \ldots, p_{n-1})$	Set of agents staying at nodes (p_i: a sequence of agents staying at node v_i)
$Q = (q_0, q_1, \ldots, q_{n-1})$	Set of agents residing on links (q_i: a sequence of agents in transit from v_{i-1} to v_i)

A *schedule* is an infinite sequence of agents. A schedule $X = \rho_1, \rho_2, \ldots$ is fair if every agent appears in X infinitely often. An infinite sequence of configurations $E = c_0, c_1, \ldots$ is called an *execution* from c_0 if there exists a fair schedule $X = \rho_1, \rho_2, \ldots$ that satisfies the following conditions for each h ($h > 0$):

- If $\rho_{h-1} \in p_i$ holds for some i in a configuration c_h, the states of ρ_{h-1} and v_i in c_{h-1} are changed to those in c_h by a local computation of ρ_{h-1}. Let $a_j = \rho_{h-1}$. If $m_j \neq \emptyset$, all messages in m_j are delivered to a_j and consumed, that is, m_j becomes $\emptyset$. In addition, if ρ_{h-1} sends a message, the message is appended to each tail of m_l such that agent a_l is at v_i. Moreover if ρ_{h-1} releases its token at v_i, the value of t_i increases by one. After this if ρ_{h-1} decides to move to v_{i+1}, ρ_{h-1} is removed from p_i and is appended to the tail of sequence q_{i+1}. If ρ_{h-1} decides to stay, ρ_{h-1} is still in p_i. The other elements in c_{h-1} are the same as those in c_h.
- If ρ_{h-1} is at the head of q_i for some i in a configuration c_h, ρ_{h-1} moves to v_i, that is, ρ_{h-1} is removed from q_i. Then, the states of ρ_{h-1} and v_i in c_{h-1} are changed to those in c_h by a local computation of ρ_{h-1}. If ρ_{h-1} sends a message, the message is appended to each tail of m_l such that agent a_l is at v_i. In addition, if ρ_{h-1} releases its token at v_i, the value of t_i increases by one. After this if ρ_{h-1} decides to move to v_{i+1}, ρ_{h-1} is appended to the tail of sequence q_{i+1}. If ρ_{h-1} decides to stay, ρ_{h-1} is inserted in p_i. The other elements in c_{h-1} are the same as those in c_h.

We consider an asynchronous system, that is, the time for each agent to transit to the next node and to wait until execution of the next operation (when staying at a node) is finite but unbounded.

5.2.4 Problem Definition

The uniform deployment problem in a ring network requires $k\,(\geq 2)$ agents to spread uniformly in the ring, that is, the distance between any two *adjacent agents* should become identical. Here, we say two agents are adjacent when there exists no agent between them. However, we should consider the case that n is not a multiple of k. In this case, we aim to distribute the agents so that the distance d of any two adjacent agents should be $\lfloor n/k \rfloor$ or $\lceil n/k \rceil$.

We consider *the uniform deployment problem with termination detection* and *the uniform deployment problem without termination detection*. At first, we define the uniform deployment problem with termination detection. In this case, *a halt state* is defined as follows: when agent a_h enters a halt state, it terminates the algorithm, that is, a_h neither changes its state nor leaves the current node even if another agent sends a message to a_h. Hence if an agent enters a halt state, it can detect its termination. Now, we define the uniform deployment problem with termination detection as follows.

Definition 5.2.1. *An algorithm solves the uniform deployment problem* with *termination detection if any execution satisfies the following conditions.*

- *All agents change their states to the halt states in finite time.*
- *When all agents are in the halt states, $q_i = \emptyset$ holds for any $q_i \in Q$ and each distance d of two adjacent agents is $\lfloor n/k \rfloor$ or $\lceil n/k \rceil$.* □

Next, we define the uniform deployment problem without termination detection. In this case, *a suspended state* is defined as follows: when agent a_h enters a suspended state, it neither changes its state nor leaves the current node unless another agent sends a message to a_h. If a_h receives a message, it can resume its behavior and leave the current node. The uniform deployment problem without termination detection allows all agents to stop in suspended states, which is also known as communication deadlock.

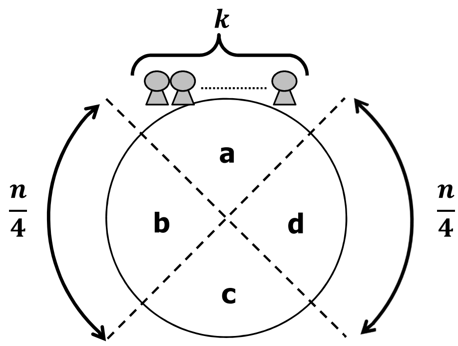

Figure 5.2: The initial configuration to derive a lower bound $\Omega(kn)$ of the total moves

Definition 5.2.2. *An algorithm solves the uniform deployment problem* without *termination detection if any execution satisfies the following conditions.*

- *All agents change their states to the suspended states in finite time.*
- *When all agents are in the suspended states, $q_i = \emptyset$ holds for any $q_i \in Q$ and each distance d of two adjacent agents satisfies $\lfloor n/k \rfloor$ or $\lceil n/k \rceil$.* □

For the uniform deployment problem, we have the following lower bound of total moves. this lower bound holds even if agents have knowledge of k.

Theorem 5.2.1. *When $k \leq pn$ holds for some constant $p\,(p < 1)$, a lower bound of the total moves to solve the uniform deployment problem (with or without termination detection) is $\Omega(kn)$ even if agents have knowledge of k.*

Proof. We assume for simplicity that $k \leq n/4$ holds and consider the initial configuration such that all agents stay in a quarter part of the ring like Fig. 5.2. In such an initial configuration, the ring is aperiodic and $l = 1$ holds. Then, the ring is divided into four quarter parts, and in the initial configuration, all agents are in the part a. To achieve the uniform deployment, $k/4$ agents need to move to the part c, the opposite part of a, and each of them must move at least $n/4$ times. Thus the total number of moves is at least $(k/4) \times (n/4) = kn/16$. This argument can be easily extended to any constant $p(p < 1)$ satisfying $k \leq pn$. □

Next, we evaluate the *time complexity* as the time required to achieve the uniform deployment. Since there is no on time in asynchronous systems, it is impossible to measure the exact time. Instead we consider the *ideal time complexity*, which is defined as the execution time under the following assumptions: 1) The time required for an agent to move from a node to its neighboring node or to wait until execution of the next action is at most one, and 2) the time required for local computation is ignored (i.e., zero)[1]. Note that these assumptions are introduced only to evaluate the time complexity, that is, algorithms are required to work correctly in asynchronous systems. In the following, we simply use terms "time complexity" and "time" instead of "ideal time complexity". Then, we can show the following theorem similarly to Theorem 5.2.1.

Theorem 5.2.2. *A lower bound of the time complexity to solve the uniform deployment problem (with or without termination detection) is* $\Omega(n)$. □

5.3 Agents with knowledge of k

In this section, we consider the uniform deployment problem for agents with knowledge of k.[2] We propose two algorithms to solve the uniform deployment problem with termination detection. The first algorithm is trivial one and requires $O(k \log n)$ memory space per agent, $O(n)$ time, and $O(kn)$ total moves. The second algorithm reduces the memory space per agent to $O(\log n)$, but allows $O(n \log k)$ time, and requires $O(kn)$ total moves.

5.3.1 A trivial algorithm with $O(k \log n)$ agent memory

In this section, we propose an algorithm to solve the uniform deployment problem with termination detection which requires $O(k \log n)$ memory space per agent, $O(n)$ time, $O(kn)$ total moves. For simplicity, we assume $n = ck$ for some positive integer c, and we can remove this assumption in Section 5.3.1. The algorithm consists of the following two phases: the selection phase and the deployment phase. In the selection phase, each agent

[1] This definition is based on the ideal time complexity for asynchronous message-passing systems [52].

[2] We assume agents with knowledge of k, but agents with knowledge of n can similarly solve the problem.

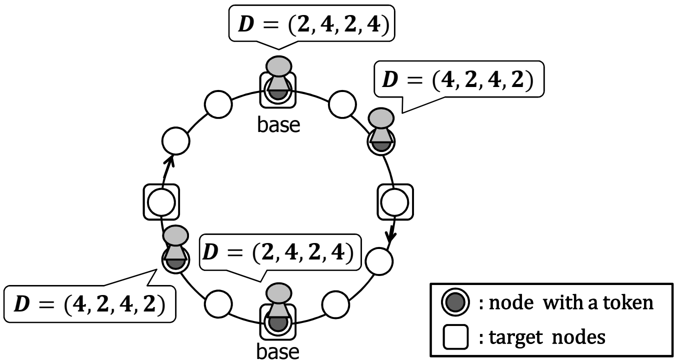

Figure 5.3: The base nodes and the target nodes

travels once around the ring and selects a *base node* as a reference node of the uniform deployment. In the deployment phase, based on the base node, each agent determines a *target node* where it should stay and moves there.

In the selection phase, each agent a_h firstly releases its token at its home node $v_{HOME}(a_h)$, and after this travels once around the ring. Note that since agents have knowledge of k, they can detect they travelled once around the ring or not. During the traversal, a_h memorizes the distance *dis* between two adjacent token nodes, and stores *dis* to an array D for memorizing the distance sequence. When finishing travelling the ring, a_h gets the value of n and the distance sequence $D = (d_0, d_1, \ldots, d_{k-1})$, where d_j is is the distance from the j-th token node it found to the $(j+1)$-th token node. Note that a_h's home node $v_{HOME}(a_h)$ is considered as the 0-th token node. Let x be the minimum number such that $\mathit{shift}(D, x) = D_{min}$ holds, where D_{min} is the lexicographically minimum distance sequence among $\{\mathit{shift}(D, x) | 0 \le x \le k-1\}$. Then, a_h selects base node v_{base} where the agent whose distance sequence is D_{min} initially stays. If D is aperiodic, all the agents select the same node as a base node. If D is periodic, multiple nodes are selected as base nodes (Fig. 5.3). However in this case, each agent can determine its base node and target node uniquely, and we explain this later.

In the deployment phase, each agent a_h determines its target node and moves there. Let *disBase* be the distance from its home node $v_{HOME}(a_h)$ to v_{base}. In addition, a_h

considers that it is the *rank*-th agent ($0 \leq rank \leq k-1$) from v_{base} (the agent staying at v_{base} is considered as the 0-th agent). Then, agents firstly moves *disBase* times and reaches v_{base}. After this, a_h moves its target node by moving $rank \times n/k$ times and terminates the algorithm. Note that if multiple base nodes are selected like Fig. 5.3, the following properties are satisfied: 1) The distance between every pair of two adjacent base nodes is identical, and 2) the number of agents and their locations between every pair of adjacent base nodes are also identical. Thus the base nodes can be reference nodes of the uniform deployment, and each agent can determine its base node and target node uniquely.

The pseudocode is described in Algorithm 5.1. We have the following theorem.

Theorem 5.3.1. *For agents with knowledge of k, Algorithm 5.1 solves the uniform deployment problem with termination detection. This algorithm requires $O(k \log n)$ memory space per agent, $O(n)$ time, and $O(kn)$ total moves.*

Proof. It is obvious that Algorithm 5.1 solves the uniform deployment problem, and in the following we analyze the complexity measures.

At first, we evaluate the memory requirement per agent. Each agent eventually gets the distance sequence $D = (d_0, d_1, \ldots, d_{k-1})$. Since each d_i is at most n, this sequence requires $O(k \log n)$ memory space . Moreover, the other variables require $O(\log n)$ bit memory. Therefore, the memory requirement per agent is $O(k \log n)$.

Next, we analyze the time complexity and the total moves. In the selection phase, each agent travels once around the ring to get D, which takes n unit times and n moves. In the deployment phase, each agent moves to its own target node, which takes at most $2n$ unit times and $2n$ moves. Thus, the time complexity is $O(n)$ and the total number of moves is $O(kn)$. □

The uniform deployment for the case of $n \neq ck$

To remove the restriction of $n = ck$ imposed in Section 5.3.1, only the parts for determining the target nodes and for moving to a target node should be modified. In the case that n is not a multiple of k, the distance between some adjacent target nodes should be

Algorithm 5.1 A time optimal algorithm for agents with knowledge of k

Main behavior of Agent a_h

/* selection phase */
$i = 0$
release a token at its home node $v_{HOME}(a_h)$
while $i \neq k$ **do**
  move to the nearest token node and get the distance *dis* between two token nodes
  $D[i] = dis$
  $i = i + 1$
end while
// a_i completes travelling once around the ring and gets the number of nodes
$n = D[0] + D[1] + \cdots + D[k-1]$

/* deployment phase */
let D_{min} be the lexicographically minimum sequence among $\{shift(D, x) | 0 \leq x \leq k-1\}$
$rank = \min\{x \geq 0 | shift(D, x) = D_{min}\}$
$disBase = D[0] + D[1] + \cdots + D[k-1-rank]$
move $disBase + rank \times n/k$ times
terminate the algorithm

$\lceil n/k \rceil$ or $\lfloor n/k \rfloor$.

The target nodes should be determined by each agent so that the decisions of different agents should be identical. Since all the agents recognize the same nodes as the base nodes, the common target nodes can be determined using the base nodes as reference nodes: Let b be the number of the base nodes, and $r = n \bmod k$. The distance of every pair of adjacent base nodes is identical even in the case of $n \neq ck$, and is $n/b = (\lfloor n/k \rfloor \times k + r)/b = \lfloor n/k \rfloor \times k/b + r/b$ (notice that k/b and r/b are integers). This implies that we should select $k/b - 1$ target nodes between two adjacent base nodes so that the first r/b intervals between adjacent target nodes should be $\lceil n/k \rceil$ and others should be

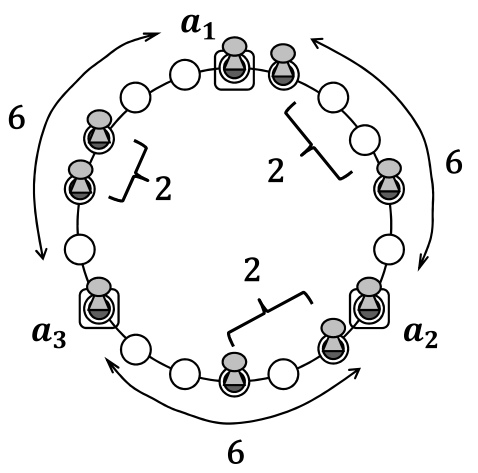

Figure 5.4: An example of the base node condition ($n = 18, k = 9, d = 2$)

$\lfloor n/k \rfloor$. With considering the above, each agent can determine its own target node by local computation so that all the agents can spread over the ring to achieve the uniform deployment.

5.3.2 An algorithm with $O(\log n)$ agent memory

In this section, we propose an algorithm to solve the uniform deployment problem with termination detection which reduces the memory space per agent to $O(\log n)$, but allows $O(n \log k)$ time, and requires $O(kn)$ total moves. The algorithm consists of two phases: selection phase and deployment phase. For simplicity we assume $n = ck$ for some positive integer c in the following description, and this restriction is removed similarly in Section 5.3.1.

Selection phase

In this phase, some of home nodes are selected as the base nodes, and they are used as reference nodes for the uniform deployment. The selected base nodes should satisfy the following condition called the *base node condition*: 1) There exists at least one base node, 2) the distance between every pair of adjacent base nodes is identical, and 3) the number of home nodes between every pair of adjacent base nodes is identical. The last condition is introduced to guarantee that the number of the selected base nodes is a divisor of k. For example, let us consider the initial locations of agents like Fig. 5.4. Then, distances from $v_{HOME}(a_1)$ to $v_{HOME}(a_2)$, $v_{HOME}(a_2)$ to $v_{HOME}(a_3)$, and $v_{HOME}(a_3)$ to $v_{HOME}(a_1)$ are all 6, and the number of home nodes between $v_{HOME}(a_1)$ and $v_{HOME}(a_2)$, $v_{HOME}(a_2)$ and $v_{HOME}(a_3)$, and $v_{HOME}(a_3)$ and $v_{HOME}(a_1)$ are all 2. Thus, $v_{HOME}(a_1)$, $v_{HOME}(a_2)$, and $v_{HOME}(a_3)$ satisfy the base node condition. Agents select such base nodes with $O(\log n)$ memory. When the selection phase is completed, each agent stays at its home node and knows whether its home node is selected as a base node or not. We call an agent a *leader* (but probably not unique) when its home node is selected as a base node, and call it a *follower* otherwise.

Now, we explain the way to select the base nodes satisfying the base node condition. The state of an agent is *active*, *leader* or *follower*. Active agents are candidates for leaders, and initially all agents are active. Once an agent becomes a follower or a leader, it never changes its state. In the following, we say that a node v is active (resp., a follower) when v is the home node of an active (resp., a follower) agent. At the beginning of the algorithm, each agent a_h releases its token at its home node $v_{HOME}(a_h)$. The selection phase consists of at most $\lceil \log k \rceil$ *sub-phases*. At the beginning of each sub-phase, each agent stays at its own home node. During the sub-phase, if the agent is a follower, it stays at its home node. If the agent is active, it travels once around the ring and decides whether it remains active or not in the next sub-phase using *IDs*.[3] Concretely, the ID (not necessarily unique) of an active agent a_h is given as $(d_h, fNum_h)$, where d_h is the distance from its home node $v_{HOME}(a_h)$ to the next active node in the sub-phase, say v_{next}, and $fNum_h$ is the number of follower nodes between $v_{HOME}(a_h)$ and v_{next}. For example in Fig. 5.5, when agent a_h moves from its home node v_j to the next active node v'_j, it visits five nodes and observes two follower nodes. Hence, a_h gets its own ID $ID_i = (5, 2)$. We compare two IDs by the lexicographical order: for $ID_1 = (d_1, fNum_1)$ and $ID_2 = (d_2, fNum_2)$, we say $ID_1 < ID_2$ if $(d_1 < d_2) \vee ((d_1 = d_2) \wedge (fNum_1 < fNum_2))$ holds. Each active agent decides whether it remains active or not using such IDs. Notice

[3] Active agents can detect they traveled once around the ring or not since they have knowledge of k.

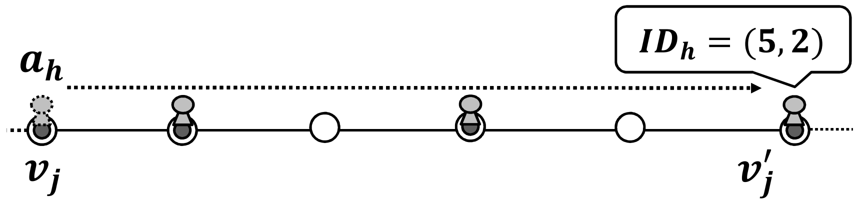

Figure 5.5: An ID of an active agent a_h

that in different sub-phases, the IDs of the same agent are different since the number of active agents is reduced in every sub-phase.

In the following, we explain the implementation of the sub-phase. In the sub-phase, each active agent a_h travels once around the ring. While travelling, a_h executes the followings:

1. Get its own ID $ID_h = (d_h, fNum_h)$:
   Agent a_h gets its own ID ID_h by moving from its home node $v_{HOME}(a_h)$ to the next active node v_{next} with counting the numbers of nodes and follower nodes (Fig. 5.5). Since all active agents are traversing the ring and all follower agents are staying at their home nodes, a_h can detect its arrival at the next active node when it visits a node with a token but with no agent. Note that this statement holds even in asynchronous systems because active agents do not pass other active agents from the FIFO property of links and the atomicity of the execution.

2. Get the ID $ID_{next} = (d_{next}, fNum_{next})$ of its next active agent:
   Similarly, with counting the numbers of nodes and follower nodes, a_h moves from v_{next} to the next active node (i.e., the node with a token but with no agent). Then, a_h gets the ID of $a_h's$ next active agent and stores it to ID_{next}.

3. Compare ID_h with those of all active agents:
   During the traversal of the ring, a_h compares ID_h with IDs of all active agents one by one, and checks 1) whether ID_h is the minimum and 2) whether the IDs of all active agents are identical. To check these, agent a_h keeps boolean variables

Algorithm 5.2 The behavior of active agent a_h

Behavior of Agent a_h

/*selection phase*/
$phase = 1$, $identical = true$, $min = true$
release a token at its home node $v_{HOME}(a_h)$
while $phase \neq \lceil \log k \rceil$ **do**
  move to the next active node and get its own ID $ID_h = (d_h, fNum_h)$
  if a_h is at $v_{HOME}(a_h)$ **then** change its state to a leader state // only a_h is active
  move to the next active node and get ID $ID_{next} = (d_{next}, fNum_{next})$ of the next active agent
  if $ID_h \neq ID_{next}$ **then** $identical = false$
  if $ID_h > ID_{next}$ **then** $min = false$ // there exists an agent with smaller ID
  while a_h is not at $v_{HOME}(a_h)$ **do**
    move to the next active node and get ID $ID_{other} = (d_{other}, fNum_{other})$ of the next active agent
    if $ID_h \neq ID_{other}$ **then** $identical = false$
    if $ID_h > ID_{other}$ **then** $min = false$ // there exists an agent with smaller ID
  end while
  if $identical = true$ **then** change its state to a leader state // all active agents have the same IDs
  if $(min = false) \vee (ID_h = ID_{next})$ **then** change its state to a follower state
  $phase = phase + 1$, $identical = true$, $min = true$
end while

min ($min = true$ means ID_h is the minimum among ever-found IDs) and $identical$ ($identical = true$ means that ever-found IDs are identical), and it updates the variables (if necessary) every time it finds an ID of another active agent.

When a_h completes the traversal, it determines its state for the next sub-phase. If $identical = true$ holds, this means that all active agents have the same IDs. In this case, a_h (and the other active agents) becomes a leader and completes the selection phase. If *identical* = *false* holds, a_h remains active if $min = true$ and $ID_h < ID_{next}$ hold. The second condition means that, when active agents with the minimum ID appear consecutively, only one of them (or the last agent in the consecutive agents) remains active. This guarantees that the number of active agents is at least halved in each sub-phase. If a_h does not satisfy any of the above conditions, it becomes a follower. By repeating such sub-phase at most $\lceil \log k \rceil$ times, all the remaining active agents have the same IDs in some phase and they are selected as leaders so that their home nodes (or the base nodes) should satisfy the base node condition.

The pseudocode is described in Algorithm 5.2. Note that in the first sub-phase of Algorithm 5.2, each agent can get the number n of nodes when it finishes travelling once around the ring, but we omit the description.

Deployment phase

In this phase, each agent determines its target node and moves there. From the base node conditions, the base nodes are first selected as the target nodes. Hence, letting b be the number of the base nodes, other $k - b$ target nodes are selected so that the distance between two adjacent target nodes should be n/k.

While the leaders know the completion of the selection phase, followers do not know the fact. Hence, at the beginning of the deployment phase, each leader notifies followers that the selection phase is completed. To do this, each leader moves to the next base node. During the movement, if there exists an agent, the leader informs the agent of the number of tokens *tBase* to the next base node. If the leader arrives at the next base node, it terminates the algorithm there since the current base node is its target node.

When each follower receives the value of *tBase*, it knows the completion of the selection phase. Then, it starts the deployment phase. Each follower moves in the ring until it observes *tBase* tokens, and then it reaches the nearest base node. After this, the agent traverses the ring until it finds a vacant target node: every time the agent moves n/k times, it reaches a target node and stays there if the node is vacant (i.e., no agent is staying), otherwise (i.e., when the target node is already occupied by another agent) it keeps moving to the next target node. Note that from the atomicity of the execution,

Algorithm 5.3 The behavior of leader or follower agent a_h

Behavior of Agent a_h

/*deployment phase*/
// the behavior of leader agents
if a_h is in the leader state **then**
    $t = 0$
    while $t \neq \mathit{fNum}_h$ **do**
        move to the next node where a token exists // look for a follower agent
        send $\mathit{tBase}\,(= \mathit{fNum}_h - t)$ to the agent at the current node
        $t = t + 1$
    end while
    move to the next node where a token exists // move to the next base node
    terminate the algorithm
end if

// the behavior of follower agents
if a_h is in the follower state **then**
    wait at the current node until a_h receives the value of *tBase*
    move until it observes *tBase* tokens // a_h reaches the nearest base node
    while true **do**
        move n/k times // move to the next target node
        if there exists no agent at the current node **then** terminate the algorithm
    end while
end if

it does not happen that two follower agents arrive at the same target node at the same time, that is, exactly one follower stays at each target node. The pseudocode is described in Algorithm 5.3. We have the following theorem about the presented algorithm.

Theorem 5.3.2. *For agents with knowledge of k, Algorithms 5.2 and 5.3 solve the uniform deployment problem with termination detection. This algorithm requires $O(\log n)$*

memory space per agent, $O(n \log k)$ time, and $O(kn)$ total moves.

Proof. It is obvious that Algorithms 5.2 and 5.3 solve the uniform deployment problem even in periodic rings, and in the following we analyze the complexity measures.

At first, we evaluate the memory requirement per agent. Each agent a_h has three variables ID_i, ID_{next} ID_{other} to store IDs, each of which requires $O(\log n)$ memory. Since other variables require $O(\log n)$ memory or less, each agent requires $O(\log n)$ memory.

Next, we consider the time complexity. The selection phase requires at most $n\lceil \log k \rceil$ unit times because each sub-phase requires n unit times and agents execute at most $\lceil \log k \rceil$ sub-phases. In addition, the deployment phase requires at most $2n$ unit times. Hence, the time complexity is $O(n \log k)$.

Lastly, we consider the total moves. First, we consider the selection phase. In each sub-phase, each active agent travels once around the ring, and then at least half active agents become followers or all active agents become leaders. Hence, in the beginning of the x-th sub-phase, the number of active agents is at most $k/2^{x-1}$. Since follower agents and leader agents never move in the selection phase, the total number of moves in the selection phase is at most $\sum_{1 \leq x \leq \log k}(k/2^{x-1})n \leq 2kn$. In the deployment phase, each leader moves to the next base node and each follower moves to a target node to achieve the uniform deployment. Each leader obviously moves at most n times, and each follower moves at most $2n$ times since it first moves to the nearest base node, which requires at most n moves, and then moves to a vacant target node, which requires at most n moves. Thus, the total moves in the deployment phase is $O(kn)$. Therefore, the total moves is $O(kn)$. □

5.4 Agents with no knowledge of k or n

In this section, we consider the uniform deployment problem for agents with no knowledge of k or n. We consider cases with termination detection and without termination detection in this order.

5.4.1 Uniform deployment problem with termination detection

When termination detection is required, we show that there exists no algorithm to solve the problem. Intuitively, it is due to impossibility of finding correct k or n when some part of the initial configuration has symmetry: when an agent misestimates these at smaller numbers than actual ones, it prematurely terminates and the uniform deployment cannot be achieved.

Theorem 5.4.1. *There exists no algorithm to solve the uniform deployment problem with termination detection even if agents can communicate with another agent at the same node.*

Proof. We use the similar idea in [25], which shows that for agents without any knowledge there exist no algorithms to solve the rendezvous problem with termination detection. We prove the theorem by contradiction, that is, we assume that there exists algorithm $\mathcal{A}$ to solve the uniform deployment problem with termination detection.

At first, let us consider n-node ring R and the initial configuration C_0 such that k agents $a_0, a_1, \ldots, a_{k-1}$ exist in this order. Let $V = \{v_0, v_1, \ldots, v_{n-1}\}$ and assume that $d = n/k$ is a positive integer. From hypothesis, there is an execution E_R of $\mathcal{A}$ to solve the uniform deployment problem in R. We define $T(E_R)$ as the length of E_R and denote $E_R = C_0, C_1, \ldots, C_{T(E_R)}$. Note that in $C_{T(E_R)}$, all agents are in the halt states and every distance between two adjacent agents is d.

Next, let us consider a larger ring R' consisting of $2qn + 2n$ nodes, where q is the minimum integer such that $qn \geq T(E_R)$ holds. Let $V' = \{v'_0, v'_1, \ldots, v'_{2qn+2n-1}\}$. We consider the initial configuration C'_0 such that $kq + k$ agents $a'_0, a'_1, \ldots a'_{kq+k-1}$ exist in this order in R'. Then in R', the interval of the uniform deployment is $2d$. In addition, we define the initial position of each agent in R' as follows. Let $v_{f(h)}$ be the node where agent a_h initially stays in R. Then, we assume that agent a'_h initially stays at node $v'_{f(h \bmod k)+n\cdot\lfloor h/k \rfloor}$. That is, the initial positions for R are repeated from v'_0 to v'_{qn+n-1}, and there is no agent from v'_{qn+n} to $v'_{2qn+2n-1}$. For each node v'_j in R', we define $C_v(v'_j) = v_{j \bmod n}$ as the corresponding node of v'_j in R. In the following, we show that each agent $a'_h\,(0 \leq h \leq k-1)$ behaves in the exactly same way as agent a_h in R

and a'_h enters a halt state at the same time as a_h. Then, the distance between the two adjacent agents is d, which contradicts that the interval of the uniform deployment in R' is $2d$.

At first, we have the following lemma. We define the *local configuration* of node v as the 2-tuple that consists of the state of v and the states of all agents at v.

Lemma 5.4.1. *Let us consider execution $E_{R'} = C'_0, C'_1, \ldots, C'_{T(E_R)}, \ldots$ for ring R'. We define $V'_t = \{v'_t, v'_{t+1}, \ldots, v'_{qn+n-1}\}$. For any $t \leq T(E_R)$, configuration C'_t satisfies the following condition: for each $v'_j \in V'_t$, the local configuration of v'_j in C'_t is the same as that of $C_v(v'_j)$ in C_t.* □

Proof. We prove Lemma 5.4.1 by induction on t. For $t = 0$, Lemma 5.4.1 holds from the definition of R'. Next, we show that when Lemma 5.4.1 holds for $t\,(t < T(E_R))$, Lemma 5.4.1 holds for $t + 1$.

From the hypothesis, for each $v'_j \in V'_{t+1}$ the local configurations of v'_{j-1} and v'_j in C'_t are the same as those of $C_v(v'_{j-1})$ and $C_v(v'_j)$ in C_t respectively. Hence, agents at v'_{j-1} and v'_j in C'_t behave in the exactly same way as those at $C_v(v'_{j-1})$ and $C_v(v'_j)$ in C_t. Since only agents at nodes v'_{j-1} and v'_j can change the local configuration of v'_j in unidirectional rings, the local configuration of v'_j in C'_{t+1} is the same as that of $C_v(v'_j)$ in C_{t+1}.

Therefore, we have the lemma. □

From Lemma 5.4.1, in $C'_{T(E_R)}$ local configuration of each node in $V^* = \{v'_{qn}, v'_{qn+1}, \ldots, v'_{qn+n-1}\} \subseteq V'_{T(E_R)}$ is the same as that of the corresponding node in $C_{T(E_R)}$. Note that the set of nodes corresponding to nodes in V^* is equal to V, and every agent in V^* also stops in the halt state in configuration $C'_{T(E_R)}$. Hence in $C'_{T(E_R)}$, there exist k agents in the halt states in V^*. Then, the distance between the adjacent agents in V^* is d, which is a contradiction.

Therefore, we have the theorem. □

5.4.2 Uniform deployment problem without termination detection

In this section, we propose an algorithm to solve the uniform deployment problem without termination detection which requires $O((k/l)\log(n/l))$ memory space per agent, $O(n/l)$ time, and $O(kn/l)$ total moves, where l is the symmetry degree of the initial configuration. This result means that when the initial configuration has higher symmetry degree, agents can solve the problem more efficiently. At first, we consider the case for aperiodic rings (The definitions of periodic and aperiodic rings are described in Section 3.5.1). After this, we show that our proposed algorithm achieves the uniform deployment also in periodic rings.

Case for aperiodic rings

In Section 5.3, since agents have knowledge of k, they can detect whether they traveled once around the ring or not. However in this section, agents cannot do this since they have no knowledge of k or n. Hence, at first agents estimate the number of nodes in the ring, and after this they move to their target nodes based on the estimations. Concretely, the algorithm consists of three phases: estimating phase, patrolling phase, and deployment phase. In the estimating phase, each agent a_h moves in the ring and estimates the number of nodes. At the end of this phase, we can show that at least one agent estimates the correct number n of nodes. In the patrolling phase, a_h moves in the ring several times depending on its estimated number of nodes. During the movement, if a_h visits the node where another agent exists, this agent may misestimate the number of nodes and prematurely stop at an incorrect target node. Hence, a_h sends its estimated number of nodes (with some information) to the agent. By this behavior, we can show that every agent eventually gets the correct number n of nodes and the location of its correct target node. In the deployment phase, a_h moves to its target node and enters a suspended state. After this, if a_h receives a message and recognizes that it misestimates the number of nodes, a_h decides its new target node from the message and moves there. For simplicity we assume $n = ck$ for some positive integer c in the following description, and this restriction can be removed similarly as in Section 5.3. In addition for sequence

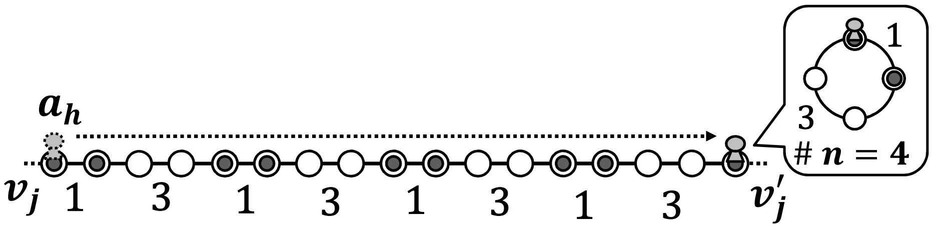

Figure 5.6: An example that an agent estimates the number of nodes

Y, we define $Y^1 = Y$ and $Y^{l+1} = Y^l \cdot Y$ (or concatenation of $(l+1)$ Ys).

Estimating phase. In this phase, each agent a_h firstly releases its token at its home node $v_{HOME}(a_h)$. After this, a_h moves in the ring, memorizes the distance *dis* between two adjacent token nodes, and stores *dis* to an array D for memorizing the distance sequence. Agent a_h continues such a behavior until it completes estimating the number of nodes. Concretely, a_h continues to move until it observes the same distance sequence four times consecutively. Let $4n'$ be the number of nodes that a_h ever visited by the time. Then, a_h considers it travelled four times around the ring and estimates the number of nodes in the ring at n'. For example, let us consider Fig. 5.6. Each number in the figure represents the distance between two adjacent token nodes. Agent a_h moves from node v_j to v'_j and gets the distance sequence $D = (1,3,1,3,1,3,1,3) = (1,3)^4$. Then, a_h estimates the number of nodes at 4. By this behavior, we can show that 1) at least one agent estimates the correct number n of nodes (in the aperiodic ring), and 2) if the estimated number n' is not correct, $n' \leq n/2$ holds. The pseudocode is described in Algorithm 5.4. During the estimating phase, a_h uses a variable k' for storing the estimated number of agents (tokens) and a variable *nodes* for storing the number of nodes that a_h has ever visited. These variables (including n' and D) are also used in the patrolling phase and the deployment phase.

Patrolling phase. In this phase, a_h moves $8n'$ times. Then, a_h considers it traveled twelve times around the ring from the beginning with respect to its estimated number of nodes n'. During the movement, a_h may observe some agent a_h staying at some node. In

Algorithm 5.4 The behavior of agent a_h in the estimating phase

Behavior of Agent a_h

/* estimating phase */
$n' = 0, k' = 0, nodes = 0, i = 0$
release a token at its home node $v_{HOME}(a_h)$
while $n' = 0$ **do**
  move to the next token node and get the distance *dis* between two token nodes
  $D[i] = dis$, $i = i + 1$
  if $(i \mod 4 = 0) \wedge (\forall x\ (0 \leq x \leq i/4 - 1)$ $D[x] = D[x + i/4] = D[x + 2 \times i/4] = D[x + 3 \times i/4])$ **then**
    // completing the estimation of the numbers of nodes and tokens
    $k' = i/4$
    $n' = D[0] + D[1] + \cdots + D[k' - 1]$
    $nodes = 4n'$
  end if
end while
change to the patrolling phase

this case, a_h may misestimate the number of nodes and prematurely stop at an incorrect target node. Hence if a_h observes such an agent, a_h sends $n', k', nodes$, and D to a_h. By this behavior, we can show that every agent eventually gets the correct number n of nodes and the location of its correct target node. The pseudocode is described in Algorithm 5.5.

Deployment phase. In this phase, a_h selects its target node and moves there as follows. Let $D = (d_0, d_1, \ldots, d_{k'-1})$[4] be the distance sequence that a_h obtained in the estimating phase. Then, a_h selects its base node similarly to Section 5.3.1, that is, letting D_{min} be the lexicographically minimum distance sequence among $\{shift(D, x) | 0 \leq x \leq k' - 1\}$, a_h selects base node v_{base} where the agent whose distance sequence is D_{min} initially stays. In addition, a_h determines its target node and moves there similarly to Section 5.3.1. Let *disBase* be the distance from the current node to v_{base}, and a_h

Algorithm 5.5 The behavior of agent a_h in the patrolling phase

Behavior of Agent a_h

```
/* patrolling phase */
while nodes ≠ 12n' do
    move to the forward node
    nodes = nodes + 1
    if there exists another agent a_h then send (n', k', nodes, D[]) to a_h
end while
change to the deployment phase
```

considers that it is *rank*-th agent ($0 \leq rank \leq k' - 1$) from v_{base} (the agent staying at v_{base} is considered as the 0-th agent). Then, a_h firstly moves *disBase* times and reaches v_{base}. After this, a_h moves to its target node by moving $rank \times n'/k'$ times and enters a suspended state. When all agents enter suspended states, agents solve the uniform deployment problem.

However, a_h may stay at an incorrect target node when it misestimates the number of nodes. In this case, a_h eventually receives a message from another agent a_ℓ. Let n'_ℓ, k'_ℓ, $nodes_\ell$, and D_ℓ be the estimated number of nodes, the estimated number of agents, the number of nodes ever visited, and the distance sequence included in a message from a_ℓ respectively. If $n' \leq n'_\ell/2$ holds and there exists t such that $(\forall i\,(0 \leq i \leq 4k' - 1)\ D[i] = D_\ell[i+t]) \wedge (D_\ell[0] + \cdots D_\ell[t-1] = nodes_\ell - nodes)$ hold, it means that a_ℓ estimates at least twice number of nodes than a_h and memorizes a_h's whole distance sequence D as a part of D_ℓ. Then, a_h recognizes that it misestimates the number of nodes and resumes its behavior. Concretely, a_h firstly moves $12n'_\ell - nodes$ times. We can show that $12n'_\ell - nodes$ is always positive, and the proof is described in Lemma 5.4.5. Then, a_h considers it traveled twelve times around the ring from the beginning with respect to the new estimated number of nodes n'_ℓ. This guarantees that agents can achieve the uniform deployment even in periodic rings, and we explain this later. After this, it decides the new base node and its new target node from n'_ℓ, k'_ℓ, $nodes_\ell$ and D_ℓ, moves to its new target node as mentioned before, and enters a suspended state again. The pseudocode

Algorithm 5.6 The behavior of agent a_h in the deployment phase

Behavior of Agent a_h

1: /* deployment phase */
2: let D_{min} be the lexicographically minimum sequence among $\{shift(D, x) | 0 \leq x \leq k' - 1\}$
3: $rank = \min\{x \geq 0 | shift(D, x) = D_{min}\}$
4: $disBase = D[0] + D[1] + \cdots + D[k - 1 - rank]$
5: move $disBase$ times
6: $nodes = nodes + disBase$
7: move $rank \times n'/k'$ times
8: $nodes = nodes + rank \times n'/k'$
9: change its state to a suspended state
10:
11: /* behavior in the suspended state */
12: wait at the current node until a_h receives $(n'_\ell, k'_\ell, nodes_\ell, D_\ell[\,])$ from some agent a_ℓ
13: **if** $(n' \leq n'_\ell/2) \wedge$ (there exists t such that $(\forall i\, (0 \leq i \leq 4k' - 1)$ $D[i] = D_\ell[i + t]) \wedge (D_\ell[0] + \cdots D_\ell[t - 1] = nodes_\ell - nodes)$ hold) **then**
14: // a_h recognizes that it misunderstands the number of nodes
15: $n' = n'_\ell$, $k' = k'_\ell$, $D[\,] = shift(D_\ell[\,], t)$
16: move $12n' - nodes$ times
17: $nodes = 12n'$
18: go to line 2
19: **end if**

is described in Algorithm 5.6. When all agents enter suspended states, agents solve the uniform deployment problem.

An example As an example, let us consider the ring in Fig. 5.7. This ring is aperiodic but has some *periodic subsequence*, that is, some agent observes a 4-times repeated subsequence before it travels once around the ring. In such a ring, some agent misestimates the number of nodes and enters a suspended state at an incorrect target

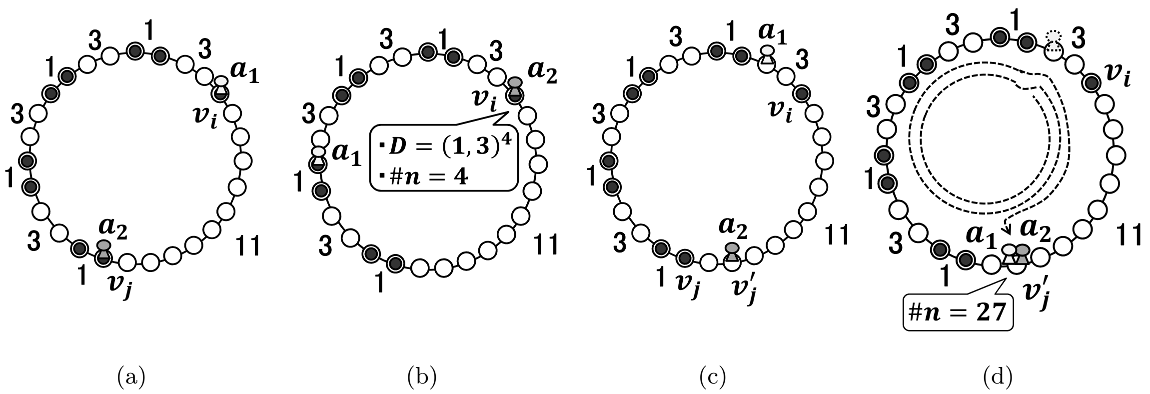

Figure 5.7: An example in the ring having some periodic subsequence ($n = 27, k = 9, d = 3$)

node. However in this case, we can show that at least one agent a_h estimates the correct number n of nodes and informs prematurely suspending agents of n during the patrolling phase. Let us consider the behavior of agents a_1 and a_2. For simplicity, we assume that they behave in a synchronous manner. In the estimating phase, agent a_2 gets the distance sequence $D = (1, 3, 1, 3, 1, 3, 1, 3) = (1, 3)^4$ and estimates the number of nodes at 4, which is incorrect (Fig. 5.7 (a) to Fig. 5.7 (b)). After this a_2 executes the patrolling and deployment phases, and enters a suspended state at incorrect target node v'_j (Fig. 5.7 (b) to Fig. 5.7 (c)). On the other hand, agent a_1 is still in the estimating phase. When a_1 observes $D = (11, 1, 3, 1, 3, 1, 3, 1, 3)^4$, it completes the estimating phase and estimates the correct number of nodes 27. After this in the patrolling phase, a_1 observes a_2 at v'_j, sends its estimated number of nodes with other information to a_2 (Fig. 5.7 (c) to Fig. 5.7 (d)), and moves to its target node. When a_2 receives the message from a_1, it recognizes that it misestimates the number of nodes and resumes its behavior.

In the following, we show that every agents eventually gets the correct number n of nodes and its correct target node. To show this, we use the following lemma.

Lemma 5.4.2. [25] *Consider an p-length sequence $A = a_0, \ldots, a_{p-1}$ and an p'-length sequence $B = b_0, \ldots b_{p'-1}$ such that $p' < p$ holds. If B^3 is the prefix of A^3, either $p' \leq p/2$ holds or B is periodic.*

Then, we have the following lemmas.

Lemma 5.4.3. *If agent a_ℓ estimates the incorrect number of nodes n_ℓ (i.e., $n_\ell \neq n$ holds), $n_\ell \leq n/2$ holds.*

Proof. Let $k_\ell\,(< k)$ be the number of agents (tokens) estimated by a_ℓ. Since a_ℓ observes $4k_\ell$ tokens in the estimating phase, it stores the same distance sequence $(D[0], \ldots, D[k_\ell - 1])$ four times, that is, $(D[0], \ldots\ , D[4k_\ell - 1]) = (D[0], \ldots, D[k_\ell - 1])^4$ holds. Then, $n_\ell = D[0] + \cdots + D[k_\ell - 1]$ holds. On the other hand since the number of tokens in the ring is $k > k_\ell$, sequence $(D[0], \ldots, D[k_\ell - 1])^4$ is the prefix of $(D[0], \ldots, D[k - 1])^4$. Note that, $n = D[0] + \cdots + D[k - 1]$ holds. Then from Lemma 5.4.2, $(D[0], \ldots, D[k_\ell - 1])$ is periodic or $k_\ell \leq k/2$ holds. If $D([0], \ldots, D[k_\ell - 1])$ is periodic, there exists $k'_\ell < k_\ell$ such that $(D[0], \ldots, D[4k'_\ell - 1]) = (D[0], \ldots, D[k'_\ell - 1])^4$ holds. This is a contradiction because a_ℓ should estimate the number of nodes at n_ℓ. Hence, $k_\ell \leq k/2$ holds. Then since $(D[0], \ldots, D[k_\ell - 1])$ is the prefix of $(D[0], \ldots, D[k - 1])$, $(D[0], \ldots, D[k - 1]) = (D[0], \ldots, D[k_\ell - 1], D[0], \ldots, D[k_\ell - 1], D[2k_\ell], D[2k_\ell + 1], \ldots)$ holds. Thus, $(D[0] + \cdots + D[k_\ell - 1]) \leq (D[0] + \cdots + D[k - 1])/2$ holds, that is, $n_\ell \leq n/2$ holds. Therefore, we have the lemma. □

Lemma 5.4.4. *If ring R is aperiodic, at least one agent estimates the correct number n of nodes and gets distance sequence D of the initial configuration in R.*

Proof. We show that at least one agent estimates the correct number n of nodes. Then from Algorithm 5.4 to 5.6, the agent clearly gets the distance sequence D for the initial configuration in R. We prove the lemma by contradiction, that is, we assume that the number of nodes estimated by each agent is less than n. We assume that in the initial configuration agents $a_0, a_1, \ldots, a_{k-1}$ exist in this order. We define n_i as the number of nodes estimated by a_h and D_i as the distance sequence observed by a_h. In addition, let S_i be the distance sequence such that $D_i = S_i^4$ holds.

Let a_m be the agent that estimates the maximum number of nodes $n_m\,(< n)$ among all agents, and let $\ell = |S_m|\,(< k)$. We assume that the distance sequence a_m observes in Algorithm 5.4 is $D_m = (d_0^m,\ \ldots\ , d_{\ell-1}^m, d_\ell^m, \ldots\ , d_{2\ell-1}^m, d_{2\ell}^m, \ldots\ , d_{3\ell-1}^m, d_{3\ell}^m, \ldots\ d_{4\ell-1}^m) =$

$(d_0^m, \dots, d_{\ell-1}^m)^4 = S_m^4$. Note that, $S_m = (d_0^m, \dots, d_{\ell-1}^m)$ is aperiodic and $\forall j\ (0 \leq j \leq \ell-1)$ $d_j^m = d_{j+\ell}^m = d_{j+2\ell}^m = d_{j+3\ell}^m$ holds.

Next, let us consider the agent $a_{m+\ell}$. Then, either $n_{m+\ell} < n_m$ or $n_{m+\ell} = n_m$ holds because n_m is the maximum. We show that $n_{m+\ell} = n_m$ always holds by contradiction, that is, we assume that $n_{m+\ell} < n_m$ holds. Then, $|S_{m+\ell}| < |S_m|$ clearly holds. Consequently, $S_{m+\ell}^3$ is the prefix of S_m^3 because $a_{m+\ell}$ gets the distance sequence $(d_\ell^m, \dots, d_{2\ell-1}^m) = S_m$ when it observes ℓ tokens. Then from Lemma 5.4.2, either $|S_{m+\ell}| \leq |S_m|/2$ holds or $S_{m+\ell}$ is periodic. If $|S_{m+\ell}| \leq |S_m|/2$ holds, agent a_m observes $S_{m+\ell}^4$ before observing S_m^4 because $(d_0^m, \dots, d_{2\ell-1}^m) = (d_\ell^m, \dots, d_{3\ell-1}^m)$ contains $S_{m+\ell}^4$ as its prefix. Consequently, a_m estimates the number of nodes at $n_{m+\ell} < n_m$, which is a contradiction. If $S_{m+\ell}$ is periodic, $S_{m+\ell} = (S'_{m+\ell})^t$ holds for some distance sequence $S'_{m+\ell}$ and some positive integer t ($S'_{m+\ell}$ is aperiodic and $|S'_{m+\ell}| \leq |S_{m+\ell}|/2$ holds). Hence, a_m observes $(S'_{m+\ell})^4$ before observing S_m^4 and the number of nodes a_m estimates is less than n_m, which is also a contradiction. Therefore, $n_{m+\ell} = n_m$ holds.

Let $m(i) = m + i\ell$ and $A_m = \{a_{m(i)} \mid i \geq 0\}$. As mentioned above, $n_m = n_{m+\ell}$ and $S_{m(0)} = S_{m(1)} = S_m$ hold. In addition, $a_{m(1)}$ observes the same distance sequence of length $4|S_m|$ as $a_{m(0)}$. Hence recursively, $a_{m(i+1)}$ observes the same distance sequence of length $4|S_m|$ as $a_{m(i)}$ and consequently each agent in A_m observes S_m as the first ℓ consecutive distances. When k is divided by ℓ, since every agent $a_{m(i)}$ observes S_m as the first ℓ consecutive distances and $\ell < k$ holds, the ring is periodic, which is a contradiction. In the following, we consider the case that k is not divided by ℓ and show that $S_{m(0)}(= S_m)$ is periodic in this case. When k is not divided by ℓ, $k = \alpha\ell + \beta\,(0 < \beta < \ell)$ holds for some positive integers α and β. Then, the prefix of $S_{m(0)}$ is identical to the suffix of $S_{m(\alpha)}$ because the trajectories of $a_{m(0)}$ and $a_{m(\alpha)}$ include the same part of the ring. We assume that t elements are overlapped, that is, $(d_0^{m(0)}, \dots, d_{t-1}^{m(0)}) = (d_{\ell-t}^{m(\alpha)}, \dots, d_{\ell-1}^{m(\alpha)})$ holds. Let be T be the sequence consisted of the t overlapped elements and T'_0 (resp., T'_α) be the sequence consisted of the other $(\ell - t)$ elements in $S_{m(0)}$ (resp., $S_{m(\alpha)}$). Then, $S_{m(0)} = TT'_0$ (resp., $S_{m(\alpha)} = T'_\alpha T$) holds (Fig. 5.8). In addition, $T'_0 = T'_\alpha$ holds because agent $a_{m(\alpha)}$ observes $S_{m(\alpha)}^4 = (T'_\alpha T)^4$ and T'_α that $a_{m(\alpha)}$ observes for the second time is equivalent to T'_0 that agent $a_{m(0)}$ observes for the first time. Then, since $S_{m(0)} = S_{m(\alpha)}$

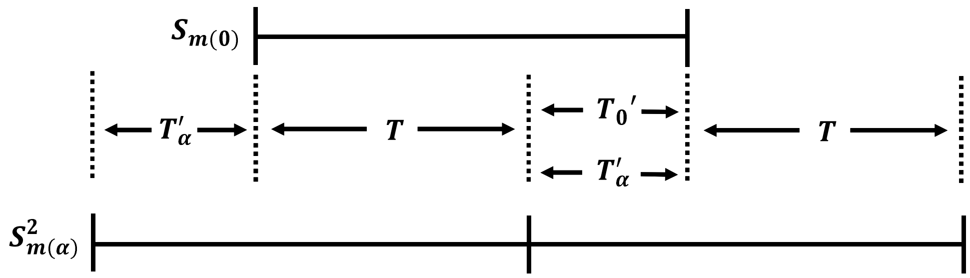

Figure 5.8: An examples of $S_{m(0)}$ and $S_{m(\alpha)}$

holds, *shift*$(S_{(m(0))}, t) = T'_0 T = T'_\alpha T = S_{m(\alpha)} = S_{m(0)}$ holds. Therefore, $S_{m(0)}$ is periodic since $0 < t < \ell$ holds. However, this contradicts the assumption that $S_{m(0)} (= S_m)$ is aperiodic.

Therefore, we have the lemma. □

Lemma 5.4.5. *If ring R is aperiodic, every agent eventually gets the correct number n of nodes and distance sequence D of the initial configuration in R.*

Proof. We show that all agents eventually get the correct number n of nodes. Then from Algorithms 5.4 to 5.6, all agents can clearly get distance sequence D of the initial configuration in R. We prove the lemma by contradiction, that is, we assume that when all agents are in the suspended states, there exists at least one agent a_h whose estimated number of nodes n' is less than n. Then from Lemma 5.4.3, $n' \leq n/2$ holds. On the other hand from Lemma 5.4.4, at least one agent a_c estimates the correct number n of nodes. In the following we show that a_c observes a_h during the patrolling phase and sends its estimated number of nodes n to a_h, which contradicts the assumption of $n' < n$.

At first, let us consider the number of nodes a_h visits. Let n_1 be the number of nodes a_h estimates in the estimating phase. From Algorithms 5.4 to 5.6, a_h moves at most $14n_1$ times by the time a_h enters a suspended state for the first time. After this, we assume that a_h receives messages and updates its estimated number of nodes to $n_2, n_3, \ldots, n_l = n'$ in this order. When a_h updates it estimated number of node to n_2, a_h's total moves at that point (i.e, *nodes*) is at most $7n_2$ since $n_1 \leq n_2/2$ holds. Hence, $12n_2 - nodes$ is clearly positive. Then, a_h firstly moves in the ring until its total moves becomes $12n_2$ by moving $12n_2 - nodes$ times. After this, a_h moves to a new target node and enters a suspended state again. This requires at most $14n_2$ total moves. Then since $n_3 \leq n_2/2$ holds from Algorithm 5.6, *nodes* is at most $7n_3$ and $12n_3 - nodes$ is clearly positive. Thus recursively, we can show that $12n_i - nodes$ is always positive ($2 \leq i \leq l$) and a_h's total moves unless it does not get the correct number n of nodes is at most $14n' \leq 7n$. On the other hand, agent a_c moves $8n$ times in the patrolling phase. Thus, a_c clearly observes a_h during the patrolling phase and sends its estimated number n of nodes to a_h, which is a contradiction.

Therefore, we have the lemma. □

Then, we have the following lemma for aperiodic rings.

Lemma 5.4.6. *When ring R is aperiodic, agents solve the uniform deployment problem without termination detection.*

Proof. From Lemma 5.4.5, all agents eventually get the correct number n of nodes and distance sequence D for the initial configuration in R. Then, each agent can compute its correct target node from D and move there. Thus, we have the lemma. □

Case for periodic rings

Next, we consider the case for periodic rings. Let R' be a periodic ring and D' be the distance sequence of the initial configuration in R'. We say R' is a (N, l)-node ring if there exists an aperiodic distance sequence D such that $D' = D^l$ holds and the total sum of elements of D is N. Then, $n = Nl$ holds and l is equivalent to the symmetry degree of the initial configuration in R'. We call the ring R with the distance sequence D the *fundamental ring of R'* (e.g., Fig. 5.9). Note that an aperiodic ring can be denoted by a $(n, 1)$-node ring. In addition for each agent a_h in R, there exist l agents in R' such that the distance sequence of each agent is l-times repetition of the distance sequence of a_h. We say such agents in R' are *corresponding agents* of agent a_h in R and denote by a_h^i

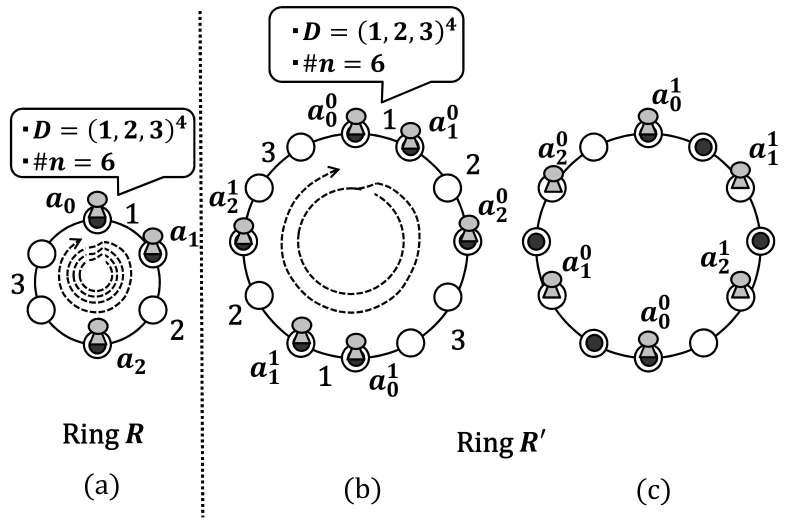

Figure 5.9: An example for the periodic ring

$(0 \leq i \leq l-1)$. We assume that agents $a_i^0, a_i^1, \ldots, a_h^{l-1}$ exist in this order and operations to an above index of a_h^i assume calculation under modulo l. Then, the distance from a_h^i to a_h^{i+1} is N. In this case, all agents eventually estimate the incorrect number $N = n/l$ of nodes, but we can show that agents can achieve the uniform deployment similarly to in R. Concretely from algorithms in Section 5.4.2, each agent moves to its target node after considering, based on the estimated number N of nodes, it traveled twelve times around the ring. This means that each agent stays at its target node during its twelfth or thirteenth circulations in the ring with respect to the estimated size N, which guarantees that when all agents are in the suspended states, no agents stay at the same node and they can achieve the uniform deployment. For example, let us consider rings in Fig. 5.9. Ring R' is the (6,2)-node periodic ring and R is the fundamental ring of R'. In R, each agent estimates the correct number 6 of nodes in the estimating phase and moves to its correct target node (Fig. 5.9 (a)). On the other hand in R', each agent also estimates the number 6 of nodes, which is incorrect (Fig. 5.9 (b)). By algorithms in Section 5.4.2, each agent moves to its target node after considering, based on the estimated size 6, it travelled twelve times around the ring, that is, after each agent moves 72 times (actually, each agent travelled six times around ring R'). This guarantees that when all agents are in the suspended states, no agents stay at the same node and they can achieve the uniform deployment (Fig. 5.9 (c)).

Now, we have the following lemmas, which can be proved similarly to the case of aperiodic rings.

Lemma 5.4.7. *Let R' be a (N, l)-node periodic ring and R be the fundamental ring of R'. Let a_h in R be the agent estimating the number N of nodes in the estimating phase. Then in R', agent $a_h^i\,(0 \le i \le l-1)$ corresponding to a_h also estimates the number N of nodes.*

Proof. From the definition of R', a_h^i observes the same distance sequence as that of a_h. In addition since agents have no knowledge of k or n, agents determine their estimated number of nodes depending only on the distance sequence they observe. Thus, a_h^i estimates the same number of nodes as that of a_h. □

Lemma 5.4.8. *Let R' be a (N, l)-node periodic ring and R be the fundamental ring of R'. Then in R', every agent eventually gets the number N of nodes and distance sequence D of the initial configuration in R.*

Proof. We show that all agents eventually get the number N of nodes. Then from Algorithms 5.4 to 5.6, all agents can clearly get distance sequence D of the initial configuration in R. We prove the lemma by contradiction, that is, we assume that when all agents are in the suspended states, there exists at least one agent a_h whose estimated number of nodes n' is less than N. On the other hand from Lemma 5.4.7, there exists agent $a_c^i\,(0 \le j \le l-1)$ estimating the number N of nodes in the estimating phase. Let $A_c = \{a_c^0, a_c^1 \ldots, a_c^{l-1}\}$. In the following, we show that some agent in A_c observes a_h during the patrolling phase and sends its estimated number N of nodes to a_h, which contradicts the assumption of $n' < N$.

At first, let us consider the number of nodes a_h visits. Similarly to the case for aperiodic rings, when a_h updates its estimated number of nodes from n'' to n', it firstly moves in the ring until its total moves becomes $12n'$ by moving $12n' - nodes$ times. After this, a_h moves to a new target node and enters a suspended state again. This requires at most $14n'$ total moves. Hence unless a_h does not get the number N of nodes, its total moves is at most $14n' \leq 7N$.

On the other hand from Lemma 5.4.7, there exists agent a_c^i in A_c such that it estimates the number of nodes at N and the distance from $v_{HOME}(a_c^i)$ to $v_{HOME}(a_h)$ is less than N. Recall that, $v_{HOME}(a)$ is the home node of agent a. Then, let us consider the behavior of agent a_c^{i-4}. Agent a_c^{i-4} firstly moves $4N$ times and finishes the estimating phase at node $v_{HOME}(a_c^i)$. After this, a_c^{j-4} moves $8N$ times from $v_{HOME}(a_c^j)$ in the patrolling phase. On the other hand, a_h moves at most $7N$ times from $v_{HOME}(a_h)$. Since the distance from $v_{HOME}(a_c^i)$ to $v_{HOME}(a_h)$ is less than N, a_c^{i-4} observes a_h during the patrolling phase and sends the number N of nodes to a_h, which is a contradiction.

Therefore, we have the lemma. □

Lemma 5.4.9. *Even when ring R' is periodic, agents solve the uniform deployment problem without termination detection.*

Proof. From Lemma 5.4.8, all agents eventually get the number N of nodes and distance sequence D of the initial configuration in R, where R is the fundamental ring of R'. From Algorithm 5.6, when agent a_h^i gets the number N of nodes it firstly moves in the ring until its total moves becomes $12N$. Then, a_h^i is at $v_{HOME}(a_h^{i+12})$. After this, a_h^i computes its target node from D and moves there, which requires at most $2N$ moves. Hence, a_h^i eventually stays between $v_{HOME}(a_h^{i+12})$ and $v_{HOME}(a_h^{i+14})$. This mean that letting v_{base} (resp., v'_{base}) be the base node existing between $v_{HOME}(a_h^{i+12})$ and $v_{HOME}(a_h^{i+13})$ (resp., $v_{HOME}(a_h^{i+13})$ and $v_{HOME}(a_h^{i+14})$) a_h^i eventually stays between v_{base} and v'_{base}. Moreover, it clearly holds total moves of each of a_h^i $(0 \leq i \leq l-1)$ are the same. Thus when all agents are in the suspended states, no agents stay at the same node and agents can achieve the uniform deployment.

Therefore, we have the lemma. □

Finally, we have the following theorem for (N, l)-node rings.

Theorem 5.4.2. *For agents with no knowledge of k or n, the proposed algorithm solves the uniform deployment problem without termination detection. This algorithm requires $O((k/l)\log(n/l))$ memory space per agent, $O(n/l)$ time, and $O(kn/l)$ total moves.*

Proof. From Lemmas 5.4.6 and 5.4.9, agents solve the uniform deployment problem. In the following, we analyze complexity measures.

At first, we evaluate the memory requirement per agent. Each agent eventually gets the distance sequence $D = (d_0, d_1, \ldots, d_{(4k/l)-1})$. Since each d_i is at most n/l, this sequence requires $O((k/l)\log(n/l))$ memory. Moreover, the other variables require $O(\log(n/l))$ bit memory. Therefore, the memory requirement per agent is $O((k/l)\log(n/l))$.

Next, we analyze the time complexity. Let $A_{correct}$ be the set of agents that estimate the number $n/l\,(= N)$ of nodes in the estimating phase. Each agent $a_c \in A_{correct}$ finishes its patrolling phase in $12n/l$ unit times, and moves to its correct target node, which requires at most $14n/l$ unit times from the beginning of the algorithm. In addition from the proof of Lemmas 5.4.5 and 5.4.8, each agent $a_h \notin A_{correct}$ gets the number n/l of nodes within $12n/l$ unit times since each $a_c \in A_{correct}$ finishes its patrolling phase in $12n/l$ unit times. After this, a_h requires at most $14n/l$ unit times to moves to its correct target node from the beginning of the algorithm. Thus, the time complexity is $O(n/l)$.

At last, we analyze the total number of agent moves. Each agent requires at most $14n/l$ moves to move to its target node. Thus, the total number of agent moves is $O(kn/l)$. □

5.5 Concluding Remarks

In this chapter, we considered the uniform deployment problem of mobile agents in asynchronous unidirectional ring networks. The uniform deployment problem, which is a striking contrast to the total gathering problem, is interesting to investigate. We proposed three algorithms to solve the uniform deployment problem from any initial configuration such that all agents are in the initial states and placed at the distinct nodes. These algorithms utilize the essential characteristic of the uniform deployment problem: the problem aims to attain the symmetry, and these algorithms solve the problem without breaking symmetry that the initial agent locations have. Such an approach in designing mobile agent algorithms seems to be applicable to other problems that aim to attain the symmetry.

Chapter 6

Conclusion

6.1 Summary of the Results

In this dissertation, we focused on the coordination of mobile agents. We considered two problems and investigated the total moves and the solvability compared with the total gathering problem.

In Chapter 3 and Chapter 4, we considered the g-partial gathering problem. The goal in these chapters is to clarify the difference of the move complexity between the total gathering problem and the g-partial gathering problem. In Chapter 3, we considered the g-partial gathering problem in ring networks under the assumption that each node has a whiteboard. For a deterministic algorithm for distinct agents or a randomized algorithm for anonymous agents with knowledge of k, we showed that agents achieve the g-partial gathering in $O(gn)$ total moves, which is asymptotically optimal. This means that g-partial gathering problem is solvable in fewer total moves than the total gathering problem. Agents can attain this improvement of the total moves since the g-partial gathering requires less symmetry breaking than the total gathering problem. In Chapter 4, we considered the g-partial gathering problem in tree networks. Since trees have lower symmetry than rings, we aimed to solve the g-partial gathering problem in weaker models than the whiteboard model used in rings. In the case of the weak multiplicity detection and removable-token model, we showed that the proposed algorithm achieves the g-partial gathering problem in $O(gn)$ total moves, which is asymptotically optimal. This means that also in tree networks the g-partial gathering problem is solvable in fewer total moves than the total gathering problem. Note that in the model with the strong multiplicity detection but without tokens, agents require $\Omega(kn)$ total moves. Hence, we showed that the total moves can be reduced dramatically by using tokens.

In Chapter 5, we considered the uniform deployment problem in ring networks under the assumption that each agent does not have a unique ID but has a token. We proposed several algorithms to solve the uniform deployment problem from any initial configuration, including configurations from which the total gathering cannot be achieved. Agents can attain this solvability since the uniform deployment aims to attain the symmetry of agent locations (i.e., requires no symmetry breaking) while the total gathering aims to break the symmetry. Hence, this result means that, while anonymous agents cannot decrease the symmetry degree for several (e.g., periodic) configurations, but they can increase the symmetry degree even from periodic configurations.

6.2 Future Directions

Regarding proposed agent algorithms for a network management, there exist several issues for improving our algorithms from both practical and theoretical points of view.

Partial Gathering In Chapter 3, we proposed two move-optimal algorithms to solve the g-partial gathering problem, that is, a deterministic algorithm for distinct agents and a randomized algorithm for anonymous agents with knowledge of k. However, it is more practical if agents do not have any IDs or global knowledge (i.e., knowledge k or n). Hence, one approach is to consider an algorithms to solve the g-partial gathering problem for such agents. In Section 3.4, a randomized algorithm for anonymous agents with knowledge of k achieves the g-partial gathering in $O(gn)$ expected total moves. This method uses knowledge of k only when consecutive active agents create the same random IDs. Thus, we should consider such a case without knowledge of k.

Another approach is to consider the g-partial gathering problem in general networks since a lot of applications are used for general networks in practice. One possible approach is that agents firstly construct a spanning tree, and then execute the g-partial gathering algorithm for trees in Chapter 4. Note that since the algorithm for constructing a spanning tree [53] is executed by nodes, we modify the algorithm to be executed by agents. However, when agents execute the algorithm for constructing a spanning tree [53], this approach consists of at most $\lceil \log k \rceil$ phases and agents require $\Omega(n \log k + m)$ total moves, where m is the number of communication links. In addition, since we can show clearly that agents requires $\Omega(gn + m)$ total moves to solve the g-partial gathering problem in general networks, this approach cannot achieve the g-partial gathering in asymptotically optimal total moves. To achieve the g-partial gathering in $O(gn + m)$ total moves, agents execute the algorithm [53] to construct a spanning tree partially so that they execute $\lceil \log g \rceil$ phases. Then, the total moves in this part could be bounded by $O(n \log g + m)$. In addition, execution of the $\lceil \log g \rceil$ phases may not complete the spanning tree construction, and thus, the network contains several tree fragments each of which satisfies the following two properties: 1) there exists no cycle, and 2) there exist at least g agents. Thus, by executing the algorithm in Chapter 4 in each fragment independently, agents can solve the g-partial gathering problem, and the total moves in this part is $O(gn)$. Therefore, we conjecture that agents can solve the g-partial gathering problem asymptotically optimal in terms of total moves also in general networks.

Uniform Deployment Similarly to the second approach of the partial gathering as mentioned above, we should consider the uniform deployment problem in networks other than rings, such as tree networks and general networks. This problem may be achieved by simulating the methods in Chapter 5, that is, agents first select several base nodes and then move to their own target nodes based on the base nodes.

Acknowledgments

I have been fortunate to receive assistance from many people. First of all, I deeply would like to appreciate my supervisor Professor Toshimistu Masuzawa for his guidance and encouragement. He has always given me precious and helpful advices. Secondly, I would like to extend my gratitude to Professor Kenichi Hagihara, Professor Shinji Kusumoto, and Associate Professor Hirotsugu Kakugawa for their precious comments on my work and this dissertation. I am grateful to appreciate Associate Professor Fukuhito Ooshita at Nara Institute of Science and Technology for his daily helpful comment and advice. I would like to acknowledge Professor Katsuro Inoue and Professor Yasushi Yagi for their helpful comments on my work.

I would like to thank to Professor Masafumi Yamashita at Kyushu University, Professor Koichi Wada at Hosei University, Professor Yoshiaki Katayama at Nagoya Institute of Technology, Associate Professor Sayaka Kamei at Hiroshima University, Associate Professor Taisuke Izumi at Nagoya Institute of Technology, Lecturer Tomoko Izumi at Ritsumeikan University, Assistant Professor Yukiko Yamauchi at Kyushu University for their useful comments. In particular, I would like to appreciate Project Assistant Professor Junya Nakamura at Toyohashi University of Technology, Assistant Professor Yonghwan Kim at Nagoya Institute of Technology, and Dr. Yuichi Sudo at NTT Communication Science Laboratory for their useful comments and kind supports.

I could not finish this acknowledgement without saying my appreciation for all members of Algorithm Engineering Laboratory, Graduate School of Information Science and Technology, Osaka University. Especially, I would like to thank to Fusami Nishioka and Hisako Suzuki for their daily kindness. Because of their backup, I have been able to focus on my research. I also thank to all the great students in the laboratory, since I have been motivated and relaxed many time by daily enjoyable activities with the students.

Finally, I strongly appreciate my parents, Yasuaki Shibata and Etsuko Shibata, and all of my family for their kind support during my life.

Bibliography

[1] A. Hagit and W. Jennifer. Distributed computing: fundamentals, simulations, and advanced topics, volume 19. John Wiley & Sons, 2004.

[2] G. Sukumar. Distributed systems: an algorithmic approach. CRC press, 2014.

[3] I. Alon and R. Michael. Symmetry breaking in distributed networks. Information and Computation, 88(1):60–87, 1990.

[4] G. Leszek, J. Tomasz, M. Russell, and S. Grzegorz. Deterministic symmetry breaking in ring networks. In Proc. of the 35th International Conference on Distributed Computing Systems, pages 517–526. IEEE, 2015.

[5] Gerard Tel. *Introduction to distributed algorithms*. Cambridge university press, 2000.

[6] B. Leonid, E. Michael, P. Seth, and S.Johannes. The locality of distributed symmetry breaking. In Proc. of the 53rd Annual Symposium on Foundations of Computer Science, pages 321–330. IEEE, 2012.

[7] S. Johannes and W. Roger. A new technique for distributed symmetry breaking. In Proc. of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing, pages 257–266. ACM, 2010.

[8] D. Kotz S. R. Gray, G. Cybenko, A.R. Peterson, and D. Rus. D'agents: Applications and performance of a mobile-agent system, Software: Practice and Experience. 32(6):543–573, 2002.

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

[10] J. Baumann, F. Hohl, K. Rothermel, and M. Straßer. Mole–concepts of a mobile agent system. world wide web, 1(3):123–137, 1998.

[11] G. Cabri, L. Leonardi, and F. Zambonelli. Mobile agent coordination for distributed network management. Journal of Network and Systems Management, 9(4):435–456, 2001.

[12] Y. Sudo, D. Baba, J. Nakamura, F. Ooshita, H. Kakugawa, and T. Masuzawa. A single agent exploration in unknown undirected graphs with whiteboards. IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences, 98(10):2117–2128, 2015.

[13] Y. Dieudonné and A. Pelc. Deterministic network exploration by a single agent with byzantine tokens. Information Processing Letters, 112(12):467–470, 2012.

[14] J. Chalopin, S. Das, and A. Kosowski. Constructing a map of an anonymous graph: Applications of universal sequences, Proc. of the 12th International Conference on Principles of Distributed Systems, LNCS, Vol. 6490. pages 119–134, 2010.

[15] L. Gasieniec, A. Pelc, T. Radzik, and X. Zhang. Tree exploration with logarithmic memory, Proc. of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms. pages 585–594, 2007.

[16] D. Dereniowski, Y. Disser, A. Kosowski, D. Pajkak, and P. Uznański. Fast collaborative graph exploration, Information and Computation. 243:37–49, 2015.

[17] D. Yann, M. Frank, N. Andreas, and S. Nemanja. A general lower bound for collaborative tree exploration. https://www.as.inf.ethz.ch/people/members/moussetf/exploration.pdf, 2016.

[18] 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.

[19] S. Das, P. Flocchini, A. Nayak, and N. Santoro. Effective elections for anonymous mobile agents, Proc. of the 17th International Symposium on Algorithms and Computation. pages 732–743, 2006.

[20] D. Dereniowski and A. Pelc. Leader election for anonymous asynchronous agents in arbitrary networks, Distributing Computing. 27(1):21–38, 2014.

[21] 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.

[22] E. Kranakis, N. Santoro, C. Sawchuk, and D. Krizanc. Mobile agent rendezvous in a ring, Proc. of the 23rd International Conference on Distributed Computing Systems. pages 592–599, 2003.

[23] P. Flocchini, E. Kranakis, D. Krizanc, N. Santoro, and C. Sawchuk. Multiple mobile agent rendezvous in a ring, Proc. of the 6th Latin American Theoretical Informatics, LNCS, Vol. 2976. pages 599–608, 2004.

[24] L. Gasieniec, E. Kranakis, D. Krizanc, and X. Zhang. Optimal memory rendezvous of anonymous mobile agents in a unidirectional ring, Proc. of the 32nd International Conference on Current Trends in Theory and Practice of Computer Science, LNCS, Vol. 3831. pages 282–292, 2006.

[25] S. Kawai, F. Ooshita, H. Kakugawa, and T. Masuzawa. Randomized rendezvous of mobile agents in anonymous unidirectional ring networks, Proc. of the 19th International Colloquium on Structural Information and Communication Complexity, LNCS, Vol. 7355. pages 303–314, 2012.

[26] E. Kranakis, D. Krizanc, and E. Markou. Mobile agent rendezvous in a synchronous torus, Proc. of the 8th Latin American Theoretical Informatics, LNCS, Vol. 3887. pages 653–664, 2006.

[27] E. Kranakis, D. Krozanc, and E. Markou. The mobile agent rendezvous problem in the ring, Synthesis Lectures on Distributed Computing Theory, Vol. 1. pages 1–122, 2010.

[28] A.Kosowski J. Czyzowicz and A. Pelc. How to meet when you forget: Log-space rendezvous in arbitrary graphs, Distributed Computing. 25(2):165–178, 2012.

[29] A. Collins, J. Czyzowicz, L. Gasieniec, A. Kosowski, and R. Martin. Synchronous rendezvous for location-aware agents, Proc. of the 25th International Symposium on Distributed Computing, LNCS, Vol. 6950. pages 447–459, 2011.

[30] Y. Dieudonné and A. Pelc. Anonymous meeting in networks. Algorithmica, 74(2):908–946, 2016.

[31] G. D. Marco, L. Gargano, E. Kranakis, D. Krizanc, A. Pelc, and U. Vaccaro. Asynchronous deterministic rendezvous in graphs, Theoretical Computer Science. 355(3):315–326, 2005.

[32] S. Guilbault and A. Pelc. Asynchronous rendezvous of anonymous agents in arbitrary graphs, Proc. of the 32nd International Symposium on Distributed Computing, LNCS, Vol. 7109. pages 421–434, 2011.

[33] J. Czyzowicz, A. Pelc, and A. Labourel. How to meet asynchronously (almost) everywhere. ACM Transactions on Algorithms (TALG), 8(4):37, 2012.

[34] Y. Dieudonné, A. Pelc, and V. Villain. How to meet asynchronously at polynomial cost. SIAM Journal on Computing, 44(3):844–867, 2015.

[35] P. Flocchini, E. Kranakis, D. Krizanc, F. L. Luccio, N. Santoro, and C. Sawchuk. Mobile agents rendezvous when tokens fail. Proc. of the 11th International Colloquium on Structural Information and Communication Complexity, LNCS, Vol. 3104. pages 161–172, 2004.

[36] S. Das. Mobile agent rendezvous in a ring using faulty tokens. In International Conference on Distributed Computing and Networking, pages 292–297. Springer, 2008.

[37] S. Das, M. Mihalák, R. Šrámek, E. Vicari, and P. Widmayer. Rendezvous of mobile agents when tokens fail anytime. In International Conference on Principles of Distributed Systems, pages 463–480. Springer, 2008.

[38] Y. Dieudonné, A. Pelc, and D. Peleg. Gathering despite mischief. ACM Transactions on Algorithms (TALG), 11(1):1, 2014.

[39] S. Bouchard, Y. Dieudonné, and B. Ducourthial. Byzantine gathering in networks. In International Colloquium on Structural Information and Communication Complexity , pages 179–193. Springer, 2014.

[40] 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.

[41] S. Dobrev, P. Flocchini, G. Prencipe, and N. Santoro. Multiple agents rendezvous in a ring in spite of a black hole, Proc. of the 8th International Conference on Principles of Distributed Systems, LNCS, Vol. 3144. pages 34–46, 2004.

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

[43] P. Fraigniaud and A. Pelc. Deterministic rendezvous in trees with little memory Proc. of the 22nd International Symposium on Distributed Computing, LNCS, Vol. 6950. pages 242–256, 2008.

[44] P. Fraigniaud and A. Pelc. Delays induce an exponential memory gap for rendezvous in trees, ACM Transactions on Algorithms. 9(2):17, 2013.

[45] J. Czyzowicz, A. Kosowski, and A. Pelc. Time vs. space trade-offs for rendezvous in trees, Proc. of the 24th ACM Symposium on Parallelism in Algorithms and Architectures. pages 1–10, 2012.

[46] S. Elouasbi and A. Pelc. Time of anonymous rendezvous in trees: Determinism vs. randomization, Proc. of the 19th International Colloquium on Structural Information and Communication Complexity, LNCS, Vol. 7355. pages 291–302, 2012.

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

[48] N. Santoro. Determining topology information in distributed networks, Proc. of the 11th Southeaster Conference on Combinatorics, Graph Theory and Computing. pages 869–878, 1980.

[49] P. Flocchini, G. Prencipe, and N. Santoro. Self-deployment of mobile sensors on a ring, Theoretical Computer Science. 402(1):67–80, 2008.

[50] Y. Elor and A.M. Bruckstein. Uniform multi-agent deployment on a ring, Theoretical Computer Science. 412(8):783–795, 2011.

[51] L. Barriere, P. Flocchini, E. Mesa-Barrameda, and N. Santoro. Uniform scattering of autonomous mobile robots in a grid, International Journal of Foundations of Computer Science. 22(03):679–697, 2011.

[52] G. Tel. Introduction to distributed algorithms. Cambridge university press, 2000.

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