IEICE TRANS. INF. & SYST., VOL.E102–D, NO.3 MARCH 2019

PAPER Special Section on Foundations of Computer Science — Algorithm, Theory of Computation, and their Applications —

Partial Gathering of Mobile Agents in Arbitrary Networks*

Masahiro SHIBATA†a), **Daisuke NAKAMURA**††, *Nonmembers*, **Fukuhito OOSHITA**†††b), *Member*,
Hirotsugu KAKUGAWA††c), *Nonmember*, *and* **Toshimitsu MASUZAWA**††d), *Member*

SUMMARY In this paper, we consider the partial gathering problem of mobile agents in arbitrary networks. The partial gathering problem is a generalization of the (well-investigated) total gathering problem, which requires that all the agents meet at the same node. The partial gathering problem requires, for a given positive integer g, that each agent should move to a node and terminate so that at least g agents should meet at each of the nodes they terminate at. The requirement for the partial gathering problem is no stronger than that for the total gathering problem, and thus, we clarify the difference on the move complexity between them. First, we show that agents require $\Omega(gn + m)$ total moves to solve the partial gathering problem, where n is the number of nodes and m is the number of communication links. Next, we propose a deterministic algorithm to solve the partial gathering problem in $O(gn + m)$ total moves, which is asymptotically optimal in terms of total moves. Note that, it is known that agents require $\Omega(kn+m)$ total moves to solve the total gathering problem in arbitrary networks, where k is the number of agents. Thus, our result shows that the partial gathering problem is solvable with strictly fewer total moves compared to the total gathering problem in arbitrary networks.

key words: *distributed system, mobile agent, gathering problem, partial gathering problem*

1. Introduction

1.1 Background

A *distributed system* consists of a set of computers (*nodes*) connected by communication links. Recently, 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 [1], [2]. Agents can traverse the system carrying information collected at nodes that they are visiting, and process tasks on each node using the information. In other words, agents can encapsulate the process code and data, which simplifies design of distributed systems [3], [4].

The *total gathering problem* (or the rendezvous problem) is a fundamental problem for agents' coordination. This problem requires all agents to meet at a single node in finite time. By meeting at a single node, all agents can share information or synchronize behaviors among them. The total gathering problem has been considered in various kinds of networks such as rings [5], [6], trees [7], [8], tori [9], and arbitrary networks [10]–[15]. The total gathering problem for *synchronous agents* in arbitrary networks is considered in [10], [11]. While Czyzowicz et al. [10] considered it for two agents with distinct IDs, Dieudonné and Pelc [11] considered it for multiple agents with no distinct IDs (or anonymous agents) but with ability to communicate with the agents staying at the same node. The total gathering problem for *asynchronous agents* in arbitrary networks is considered in [12]–[15]. These works assume that agents cannot mark nodes in any way. De Marco et al. [12] considered it for the first time. Czyzowicz et al. [13] considered it for two distinct agents, and Guilbault and Pelc [14] considered it for two anonymous agents. While in [13], [14] agents require exponential total moves to solve the problem, Dieudonné and Pelc [15] improved the result so that agents could solve the problem in polynomial total moves.

Recently, a variant of the total gathering problem, called the *partial gathering problem* [16], has been considered. This problem does not require all agents to meet at a single node, but allows agents to meet partially at several nodes. More precisely, we consider the problem which requires, for a given positive integer g, that each agent should move to a node and terminate so that at least g agents should meet at each of the nodes they terminate at. We define this problem as the *g-partial gathering problem*. From a practical point of view, the g-partial gathering problem is still useful especially in large-scale networks. In a large scale network where a large number of mobile agents are deployed, it is impractical or unnecessary to gather all the agents at a single node. Instead, gathering some agents (say g agents) at a node is sufficient for many applications; the agents gathered at a node can share information and tasks among them. Moreover, g-partial gathering allows agents to partition the network into several subnetworks so that each subnetwork contains at least g agents. This leads to distributed manage-

---

Manuscript received March 22, 2018.
Manuscript revised July 27, 2018.
Manuscript publicized November 1, 2018.

[†] The author is with the Department of Computer Science and Electronics, Kyushu Institute of Technology, Iizuka-shi, 820–8502 Japan.

[††] The authors are with the Graduate School of Information Science, NAIST, Ikoma-shi, 630–0192 Japan.

[†††] The author is with the Graduate School of Information Science and Technology, Osaka University, Suita-shi, 565–0871 Japan.

[*] This work was partially supported by JSPS KAKENHI Grant Number 16K00018, 17K19977, 18K11167, and 18K18031, and Japan Science and Technology Agency (JST) SICORP.

a) E-mail: shibata@cse.kyutech.ac.jp (Corresponding author)
b) E-mail: f-oosita@is.naist.jp
c) E-mail: kakugawa@ist.osaka-u.ac.jp
d) E-mail: masuzawa@ist.osaka-u.ac.jp

DOI: 10.1587/transinf.2018FCP0008

ment of the network; each group of at least g agents collaboratively manages their own subnetwork. The network partition is useful especially when the meeting nodes are widely separate from each other. While g-partial gathering has no request on locations of the meeting nodes, it is expected that the meeting nodes are moderately distributed in the network when starting from the initial configuration such that agents are moderately distributed.

The g-partial gathering problem is interesting to investigate also from theoretical point of view. Let k be the number of agents. Clearly, if $k/2 < g \leq k$ holds, the g-partial gathering problem is equivalent to the total gathering problem. On the other hand, if $2 \leq g \leq k/2$ holds, the requirement for the g-partial gathering problem is no stronger than that for the total gathering problem. Thus, there exists possibility that the g-partial gathering problem can be solved with strictly fewer total moves (i.e., lower costs) compared to the total gathering problem.

1.2 Previous Works on Partial Gathering

As previous works, the g-partial gathering problem is considered in ring networks [16] and tree networks [17]. In [16], we considered the g-partial gathering problem in unidirectional ring networks with whiteboards (or memory spaces) at nodes. We considered three model variants. The first model assumes the deterministic algorithm for agents with distinct IDs. Then, our algorithm solves the g-partial gathering problem in $O(gn)$ total moves, where n is the number of nodes. The second model assumes the randomized algorithm for anonymous agents with knowledge of k. Then, our algorithm also solves the g-partial gathering problem in $O(gn)$ expected total moves. The third model assumes the deterministic algorithm for anonymous agents with knowledge of k. Then, we showed that there exist unsolvable initial configurations. In addition, we proposed an algorithm to solve the g-partial gathering problem from any solvable configurations in $O(kn)$ total moves. Note that, since the total gathering problem in ring networks requires $\Omega(kn)$ total moves [16], the first and the second results show that the g-partial gathering problem can be solved with strictly fewer total moves compared to the total gathering problem.

In [17], we considered the g-partial gathering problem in tree networks. Since trees have lower symmetry than rings and no harder to solve problems, we considered the problem in weaker models than that for rings and clarified what condition is needed to achieve g-partial gathering with the same performance as that for rings. To do this, we considered agents that are anonymous and have no knowledge of k or n, and we considered three model variants. The first and the second models assume that nodes have no memory space (or whiteboards) but are different in the multiplicity detection ability. The first model assumes the weak multiplicity detection where each agent can detect whether another agent exists staying at the current node or not but cannot count the exact number of the agents. Then, we showed that, for asymmetric trees agents can solve the g-partial gathering problem in $O(kn)$ total moves, and for symmetric trees agents cannot solve the g-partial gathering problem for the case of $g \geq 5$. The second model assumes the strong multiplicity detection where each agent can count the number of agents staying at the current node. In this case, we proposed a deterministic algorithm to solve the g-partial gathering problem in $O(kn)$ total moves regardless of the initial configuration. The third model assumes the weak multiplicity detection and assumes that agents can use removable identical tokens, which implies that each node has a whiteboard of only one bit. In this case, we proposed a deterministic algorithm to solve the g-partial gathering problem in $O(gn)$ total moves. This result shows that it is sufficient to use weak multiplicity detection and removable tokens to achieve g-partial gathering in $O(gn)$ total moves, which is the strictly weaker assumption than the whiteboard model for rings.

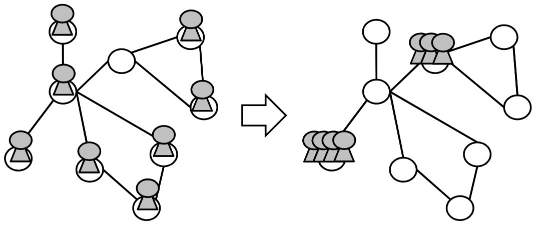

Fig. 1 An example of the g-partial gathering problem ($k = 8, g = 3$)

1.3 Our Contributions

As a natural extension, in this paper we consider the g-partial gathering problem in arbitrary networks (e.g., Fig. 1), and similarly to the previous works we aim to propose an algorithm to solve the g-partial gathering problem with strictly fewer total moves compared to the total gathering problem. Similarly to the first model of [16], we assume that agents have distinct IDs and each node has a whiteboard. First, we show that agents require $\Omega(gn + m)$ total moves to solve the g-partial gathering problem, where m is the number of communication links. Next, we propose a deterministic algorithm to solve the g-partial gathering problem in $O(gn+m)$ total moves, which is asymptotically optimal in terms of total moves. Note that, even when agents have distinct IDs and each node has a whiteboard, agents require $\Omega(kn + m)$ total moves to solve the total gathering problem in arbitrary networks. Thus, our result shows that the g-partial gathering problem is solvable with strictly fewer total moves compared to the total gathering problem also in arbitrary networks.

The paper is organized as follows. Section 2 presents the system model and the problem to be solved. In Sect. 3, we show the lower bound of the total moves, and present our algorithm to solve the g-partial gathering problem. Section 4 concludes the paper.

2. Preliminaries

2.1 Network Model

A network is represented by a general graph $G = (V, L)$, where V is a set of nodes and L is a set of communication links. We denote by n $(= |V|)$ the number of nodes and by m $(= |L|)$ the number of communication links. We denote by d_v the degree of node v and by Δ the maximum degree of the graph. Nodes have no distinct IDs (i.e., are anonymous), but each link l incidents to v is uniquely labeled at v with a label chosen from the set $\{1, 2, \ldots, d_v\}$. We call this label *port number*. Since each communication link connects two nodes, it has two port numbers, one at each end nodes. However, port numbering is *local*, that is, there is no coherence between the two port numbers. The *path* $P(v_0, v_p) = (v_0, v_1, \ldots, v_p)$ with length p is a sequence of nodes from v_0 to v_p such that $\{v_i, v_{i+1}\} \in L$ $(0 \leq i < p)$ and $v_i \neq v_j$ if $i \neq j$. Every node $v_i \in V$ has a whiteboard that agents on node v_i can read from and write into. We define W as a set of all possible states (contents) of a whiteboard.

2.2 Agent Model

Let $A = \{a_1, a_2, \ldots, a_k\}$ be a set of $k\,(\leq n)$ agents. Agents do not have knowledge of k or n, but have distinct IDs, and execute a deterministic algorithm. Each agent cannot detect whether there exists another agent staying at the current node or not. We model an agent as a state machine $(S, \delta, s_{initial}, s_{final})$. The first element S is the set of all agent states, which includes initial state $s_{initial}$ and final state s_{final}. When a_i changes its state to s_{final}, it terminates execution of the algorithm. The second element δ is the state transition function and represented by $S \times W \times P \rightarrow S \times W \times P$. Set $P = \{\perp, 1, 2 \ldots, \Delta\}$ represents the agent's movement. In the left side of δ, the value of P represents the port number assigned at the current node to the link through which the agent entered the current node. The value is $\perp$ in the first activation at the initial location. In the right side of δ, the value of P 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. The staying agent may execute an action following δ and leave the current node when the value of W changes. Notice that $S, \delta, s_{initial}$, and s_{final} can be dependent on the agent's ID.

We assume that agents move instantaneously, that is, agents always exist at nodes (do not exist on links). This assumption is introduced for simplicity and does not cause any loss of generality even in the asynchronous model[†]. During execution of an algorithm, each agent executes the following four operations in an atomic *step*: 1) Agent a_i reads the contents of its current node's whiteboard, 2) agent a_i executes local computation (or changes its state), 3) agent a_i updates the contents of the current node's whiteboard, and 4) agent a_i leaves the current node and arrives at the next node, or stays at the current node. In the last action, if a_i decides to leave the current node, it decides the port number through which it leaves.

2.3 System Configuration

In an agent system, (global) *configuration c* is defined as a product of states of agents, states of nodes (whiteboards' contents), and locations of agents. We define C as a set of all configurations. In initial configuration $c_0 \in C$, we assume that no pair of agents stay at the same node. The node where agent a is located in c_0 is called the *home node* of a and is denoted by $v_{HOME}(a)$. Moreover, each node v_j has boolean variable v_j.*initial* at the whiteboard that indicates existence of an agent in initial configuration c_0. If there exists an agent on node v_j in c_0, the value of v_j.*initial* is true. Otherwise, the value of v_j.*initial* is false.

Let A_i be an arbitrary non-empty set of agents. When configuration c_i changes to c_{i+1} by making every agent in A_i take a step as mentioned before, we denote the transition by $c_i \xrightarrow{A_i} c_{i+1}$. If multiple agents at the same node are included in A_i, the agents take steps in an arbitrary order. When $A_i = A$ holds for 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 paper, we consider the asynchronous model.

If a 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 by *schedule* $A_0, A_1, \ldots$. We consider only fair schedules, where each agent is activated after a finite (unknown) amount of time when E is finite or infinite, and infinitely many times when E is infinite. 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}.

2.4 Partial Gathering Problem

The requirement of the partial gathering problem is that, for a given positive integer g, each agent should move to a node and terminate so that at least g agents should meet at the nodes (e.g., Fig. 1). Formally, we define the g-partial gathering problem as follows.

Definition 1: Execution E solves the g-partial gathering problem when the following conditions hold:

- Execution E is finite.
- In the final configuration, for each node v_j where there exists an agent, at least g agents exist on v_j. □

[†] This is because agents are asynchronously activated at nodes and are unaware of other agents at the same node. For example, consider the case where agents a_1 and a_2 are at node v. At that time, if a_1 is activated before a_2, a_1 is unaware of a_2 and consequently the state transition of a_1 is not affected by a_2. This can be considered as the situation where a_2 is in transit to v.

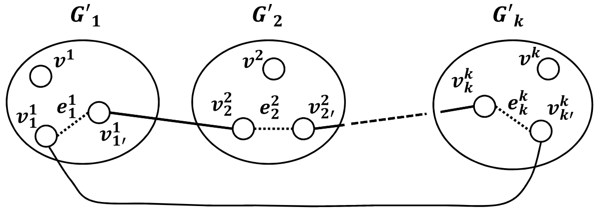

Fig. 2 Network G''

3. Partial Gathering Algorithm in Arbitrary Networks

In this section, we consider the g-partial gathering problem in arbitrary networks. First, we show a lower bound of the total number of moves, and then we present the algorithm to solve the g-partial gathering problem with the asymptotically optimal number of total moves.

3.1 Lower Bound of the Total Moves

Theorem 1: The total number of moves required to solve the g-partial gathering problem in arbitrary networks is $\Omega(gn + m)$.

proof. At first, we show that there exists a configuration such that agents require $\Omega(m)$ total moves to solve the problem. To show this, we have the following lemma.

Lemma 1: Let G' be an arbitrary network consisting of n' nodes and m' links, v be any node of G' and $\mathcal{A}$ be any algorithm to solve the g-partial gathering problem on arbitrary networks. Consider the following k executions $E_i\,(1 \le i \le k)$: in E_i, only agent a_i exists in G' and starts execution of $\mathcal{A}$ at v. Then, there exists an execution $E_j\,(1 \le j \le k)$ such that a_j passes all the links of G'.

proof. We show the lemma by contradiction, that is, we assume that for each agent $a_i\,(1 \le i \le k)$ there exists some link $e_i = (v_i, v_{i'})\,(1 \le i \le k)$ in G' that a_i does not pass during execution of $\mathcal{A}$. We assume that a_i stays at node $v^i_s\,(1 \le i \le k)$ in the initial configuration. We consider the following network G'' as follows: Let $G'_1, \dots, G'_k$ be k networks with the same topology as G', and $e^i_i = (v^i_i, v^i_{i'})$ $(1 \le i \le k)$ be the link in G'_i corresponding to e_i in G'. Then, network G'' is constructed by deleting each e^i_i and connecting $v^1_{1'}$ to v^2_2, $v^2_{2'}$ to $v^3_3, \dots, v^k_{k'}$ to v^1_1 (Fig. 2). Let v^i $(1 \le i \le k)$ be the node in G'_i corresponding to v^i_s in G'. We consider the following initial configuration c'_0 such that each agent $a_i\,(1 \le i \le k)$ is located at v^i. Then, since each agent a_i that does not pass e^i_i cannot distinguish G' from G'', agent a_i staying in the G'_i part of G'' never leaves G'_i. However, this contradicts the assumption that $\mathcal{A}$ solves the g-partial gathering problem. Thus, we have the lemma. □

Then, we have the following lemma.

Lemma 2: Let $\mathcal{A}$ be any algorithm to solve the g-partial gathering problem on arbitrary networks. Then, for any n and $m \ge 2(k-1)$ there exists an n-node and m-link network $G_{n,m}$such that the total moves for executing $\mathcal{A}$ to solve the g-partial gathering problem on $G_{n,m}$ is $\Omega(m)$.

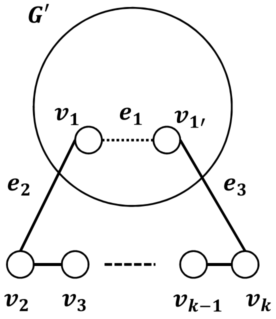

Fig. 3 Network $G_{n,m}$

proof. Let G' be an arbitrary network consisting of n' nodes and m' links $(m' \ge k-1)$. By Lemma 1, there exists some agent that passes all the link of G' when it starts execution of $\mathcal{A}$ at node v'. Without loss of generality, let a_1 be the agent. We assume that a_1 is located at v' in the initial configuration. Let $e_1 = (v_1, v_{1'})$ be the link that a_1 passes for the first time after a_1 passes every other link at least once during execution of $\mathcal{A}$. Now, we consider the following network $G_{n,m}$ by 1) deleting e_1, 2) adding $k-1$ nodes $v_2, v_3, \dots v_k$, 3) connecting v_2 to v_3, v_3 to v_4, …, v_{k-1} to v_k, and 4) connecting v_1 to v_2 and v_k to $v_{1'}$ (Fig. 3). Let $e_2 = (v_1, v_2)$ and $e_3 = (v_{1'}, v_k)$, and let v be the node in $G_{n,m}$ corresponding to v' in G'. We consider the following initial configuration such that a_1 is located at v and the other agents are located at $v_2, v_3, \dots, v_k$, respectively. Then, since we consider the asynchronous model, there exists execution of $\mathcal{A}$ such that a_1 firstly passes all the link in the G' part of $G_{n,m}$ except for the link corresponding e_1 in G' (i.e., e_2 or e_3 in $G_{n,m}$), and then passes e_2 or e_3. This requires m' moves. Since $m = m' + k - 1$ and $m' \ge k-1$ hold, this requires $m' \ge m/2 = \Omega(m)$ total moves. □

Next, we show that agents require $\Omega(gn)$ total moves for the case of $gn = \omega(m)$. Let N_1 be a $n/2$-node ring and N_2 be some network consisting of $n/2$ nodes and $m - (n/2 + 1)$ links. We consider the network N_3 connecting some node in N_1 and some node in N_2 (N_3 consists of n nodes and m links), and consider the initial configuration c_0 such that all agents ($n/2$ agents when $k > n/2$) are deployed evenly in the $n/2$-node ring part of N_3. Then, by argument similar to that in [16] showing that agents in the $n/2$-ring part require $\Omega(gn)$ total moves to solve the g-partial gathering problem, we can show that agents require $\Omega(gn)$ total moves to solve the g-partial gathering from c_0. By this fact and Lemma 2, agents require $\Omega(m + gn)$ total moves to solve the problem. Thus, we have the theorem. □

3.2 The Algorithm

In this section, we present our proposed algorithm for the

g-partial gathering problem. The basic idea is as follows. First, agents make a spanning tree [18] and then they execute the g-partial gathering algorithm for trees [17]. Note that since the algorithm for making a spanning tree [18] is executed by nodes, we modify the algorithm to be executed by agents. However, the simply modified algorithm requires $\Omega(n \log k + m)$ total moves to make a spanning tree, and it cannot achieve g-partial gathering in asymptotically optimal total moves (i.e. $O(gn+m)$). To solve the g-partial gathering problem in $O(gn + m)$ total moves, agents stop execution of the spanning tree construction algorithm [18] in the middle so that the total moves could be bounded by $O(n \log g + m)$. Then, a spanning forest of the network is constructed so that each fragment (or each tree in the forest) contains at least g agents. Thus, agents can solve the problem by executing the g-partial gathering algorithm for trees [17] in each fragment independently. By [17], the total moves of the g-partial gathering in fragments is $O(gn)$. In addition, since the total moves to construct the spanning forest is $O(n \log g + m)$, agents can solve the g-partial gathering problem in $O(gn+m)$ total moves.

The algorithm consists of three parts. In the first part, each agent creates its own fragment. In the second part, agents merge fragments so that at least g agents should exist in each of the resulting fragments. In the third part, agents execute the g-partial gathering algorithm for trees in each fragment independently.

3.3 The First Part: Fragment Creation

In this part, agents move in the network and expand their own fragments. A fragment is a region managed by an agent, and when this part completes, each node belongs to exactly one fragment in the form of a tree. Let F_i be the fragment managed by agent a_i. Each fragment F_i consists of a set V_i of nodes and a set L_i of links. At the beginning of this part, F_i consists of only $v_{HOME}(a_i)$. We use a common depth-firth graph exploration method to create fragments [19], [20]. While agents in [19], [20] are anonymous and use whiteboards to mark which port was used, agents in this paper have distinct IDs and additionally write the IDs on whiteboards to determine which fragment they should merge with in the next part. Concretely, during execution of this part, each agent a_i explores the network in the depth-first manner. When a_i visits some node v_j such that no agent ID is written on the whiteboard, it adds v_j and the link in visiting v_j to V_i and L_i, respectively. In addition, a_i writes its ID $a_i.id$ and the sequence number $a_i.seq$ on variables $(v_j.agent, v_j.seq)$ of v_j's whiteboard, respectively. We call this tuple a *node ID*, and denote by $v_j.id = (v_j.agent, v_j.seq)$. During execution of our algorithm, the value of each node ID does not change and hence we use node IDs as unique IDs (constants) in the next part. When a_i visits some node such that a node ID including a_i's ID is already written on the whiteboard, a_i returns to the previous node and resumes its exploration. When a_i visits some node with a node ID including the ID of another agent, a_i returns to the previous node and resumes its exploration. Then, it memorizes $v_j.id$ as information of *neighboring fragments*. This information is not used in this part but used in the next part. When a_i stays at $v_{HOME}(a_i)$ and there exists no port p of $v_{HOME}(a_i)$ such that a_i does not leave $v_{HOME}(a_i)$ through p, a_i completes its exploration. Then, in F_i, V_i and L_i form a tree. An example is given in Fig. 4.

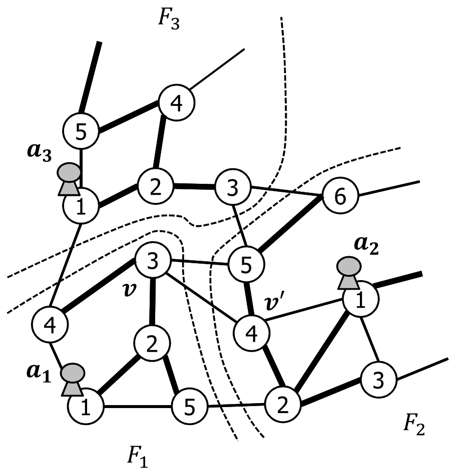

Fig. 4 An example of the fragment creation. Agent a_i manages its own fragment F_i $(i = 1, 2, 3)$, respectively. Numbers in circles represent sequence numbers, and bold lines represent links belonging to fragments.

The pseudocode is described in Algorithm 1. In Algorithm 1, we compare two node IDs by the lexicographical order: for $v_j.id = (v_j.agent, v_j.seq)$ and $v_\ell.id = (v_\ell.agent, v_\ell.seq)$, $v_j.id < v_\ell.id$ holds if $(v_j.agent < v_\ell.agent) \lor ((v_j.agent = v_\ell.agent) \land (v_j.seq < v_\ell.seq))$ holds. Node v_j and agent a_i have the following variables:

- $v_j.unsearched$ is a variable representing the set of unsearched port numbers.
- $v_j.parent$ is a variable representing the port number connecting to its parent in the tree rooted at $v_{HOME}(a_i)$, which is one through which a_i visits v_j for the first time (the value of $v_j.parent$ at $v_{HOME}(a_i)$ is 0).
- $a_i.tmpNodeID$ is a variable for storing the node ID (i.e., a pair of an agent ID and a sequence number) recorded at a node belonging to a neighboring fragment.
- $a_i.NF$ is an array for storing the information of neighboring fragments. We assume that agent a_i visits node v'_j in a neighboring fragment from node v_j in a_i's fragment. Then, one component of $a_i.NF$ is represented by $\min\{(v_j.id, v'_j.id), (v'_j.id, v_j.id)\}$.

Notice that information about $a_i.tmpNodeID$ and $a_i.NF$ are not necessary in this part but used in the next part. When a_i visits some node v_j such that $v_j.initial = true$ and $v_j.agent = \perp$, this means that some agent stays at v_j but does not start execution of the algorithm yet. In this case, a_i waits at v_j until an ID is written on v_j's whiteboard (line 10). Note that, when a_i finishes its exploration, it writes the topology information of F_i (e.g., nodes and links with their port numbers in F_i) on the whiteboard of $v_{HOME}(a_i)$ (line 30), but we omit the detail in the algorithm.

Algorithm 1 Fragment creation (v_j is the current node of a_i)

```
Behavior of Agent $a_i$
 1: $V_i = \{v_{HOME}(a_i)\}, L_i = \emptyset, a_i.NF = \emptyset$
 2: $v_j.agent = a_i.id, v_j.seq = a_i.seq = 1, a_i.seq = a_i.seq + 1$
 3: $v_j.unsearched = \{1, 2, \cdots, d_v\}, v_j.parent = 0$
 4: while $(v_j.unsarched \neq \perp) \vee (v_j.parent \neq 0)$ do
 5:   if $v_j.unsearched \neq \perp$ then
 6:     choose a port $p$ from $v_j.unsearched$
 7:     $v_j.unsearched = v_j.unsearched \backslash \{p\}$
 8:     leave $v_j$ through the port $p$
        // arrive at the next node and $v_j$ is updated
 9:     let $q$ be the port through which $a_i$ visits $v_j$
10:     if $(v_j.initial = true) \wedge (v_j.agent = \perp)$ then wait until $v_j.agent \neq \perp$
11:     if $(v_j.agent \neq a_i.id)$ then // another agent already visited $v_j$
12:       $a_i.tmpNodeID = (v_j.agent, v_j.seq)$
13:       return to the previous node through the port $q$
14:       $a_i.NF = a_i.NF \bigcup$
          $\{\min\{(v_j.id, a_i.tmpNodeID), (a_i.tmpNodeID, v_j.id)\}\}$
15:     else if $v_j.agent = a_i.id$ then // $a_i$ already visited $v_j$
16:       $v_j.unsearched = v_j.unsearched \backslash \{q\}$
17:       return to the previous node through the port $q$
18:     else // $a_i$ visits $v_j$ for the first time
19:       $v_j.parent = q, v_j.unsearched = \{1, 2, \ldots d_v\} \backslash \{q\}$
20:       $v_j.id = (v_j.agent, v_j.seq) = (a_i.id, a_i.seq)$
21:       $a_i.seq = a_i.seq + 1$
22:       let $\ell$ be the link used in visiting $v_j$
23:       $V_i = V_i \bigcup \{v_j\}, L_i = L_i \bigcup \{\ell\}$
24:     end if
25:   else
26:     // there is no unsearched port at $v_j$
27:     if $v_j.parent \neq 0$ then
28:       return to the previous node through the port $v_j.parent$
29:     else // $a_i$ is at $v_{HOME}(a_i)$
30:       write $V_i, L_i, a_i.NF$ and topology information about $F_i$ on the
          current whiteboard
31:       terminate the fragment creation part and start the fragment
          merge part
32:     end if
33:   end if
34: end while
```

We have the following lemmas for Algorithm 1.

Lemma 3: Algorithm 1 eventually terminates. When Algorithm 1 terminates, each node belongs to exactly one fragment and each fragment forms a tree.

proof. In Algorithm 1, each agent a_i explores the network by the depth-first search but it returns to the previous node when it visits a node already visited by some agent (including a_i). Thus, each agent a_i eventually completes its exploration. In addition, from lines 15 to 24 of Algorithm 1, only when agent a_i visits some node v_j for the first time (including other agents), v_j and the link used in visiting v_j are added to its fragment. This means that v_j belongs to exactly one fragment and there exists exactly one link added to its fragment when a_i visits v_j. Thus, we have the lemma. □

Lemma 4: The total number of agent moves to execute Algorithm 1 is $O(m)$.

proof. In Algorithm 1, each link connecting two fragments is passed by four time, and the other links (e.g., links between nodes in the same fragment) are passed by twice. Hence, we have the lemma. □

3.4 The Second Part: Fragment Merge

In this part, agents merge their fragments so that each fragment should contain at least g agents. We denote by $a_i.level$ the number of merges a_i has executed. We borrow the basic idea of the merge from [18], which is as follows. At first, each agent a_i with fragment F_i selects the fragment with the smallest node ID among its neighboring fragments, say F_j (managed by agent a_j), and requests to merge with F_j. If $a_i.level < a_j.level$ holds, F_i is absorbed and becomes a part of F_j. If $a_i.level = a_j.level$ holds and a_j also requests to merge with F_i, F_i and F_j are merged and the new fragment is created. Then, the agent with a smaller ID manages the new fragment and increases its level by one. Otherwise, (i.e., $a_i.level > a_j.level$ holds), a_i waits until either of the above two cases occurs. Agent a_i repeats such merge processes at most $\lceil \log g \rceil$ times.

Before explaining the detail of the merge process, we introduce a weight of links. We assume that $u.id < v.id$ holds for link $l = (u, v)$. Then, we define the link weight of link l as $(u.id, v.id)$. Since each node ID is unique, each link weight is also unique[†].

Now, we describe the detail of the merge process. We define the *minimum outgoing edge (MOE)* of agent a_i (or fragment F_i) as the link having the lexicographically minimum link weight among links connecting a node in F_i and a node in F_i's neighboring fragment, say F_j. We assume that link (v^i_m, v^j_m) is the MOE of agent a_i, and v^i_m (resp., v^j_m) is in fragment F_i (resp., F_j). Notice that the MOE of a_i is selected using $a_i.NF$ in the previous section. Agent a_i goes to v^j_m and requests to merge with F_j by writing its ID and level on v^j_m's whiteboard. For example, in Fig. 4 we assume that $a_1.id < a_2.id < a_3.id$ holds and there are three nodes in F_2 connecting to nodes in F_1. Then, since link (v, v') is the MOE of a_1, it goes to v' and requests the merge by writing its ID and level on the whiteboard of the node.

After this, a_i returns to v^i_m and determines its next behavior from the following three cases. The first case is that $a_i.level < a_j.level$ holds. In this case, F_i is absorbed and becomes a part of F_j. Then, the level of F_j that absorbed F_i does not change to guarantee a lower bound of the number of agents in a fragment with some level (Lemma 5). The detail treatment of an absorption is explained in the next case. Agent a_i goes to $v_{HOME}(a_i)$ and waits for the next instruction (Sect. 3.5).

The second case is that $a_i.level = a_j.level$ holds and a_j writes its ID on v^i_m's whiteboard. This case means that a_j also requests to merge with F_i. Then, the new fragment consisting of F_i, F_j, and link (v^i_m, v^j_m) is created. If $a_i.id < a_j.id$ holds, a_i manages the new fragment. Agent a_i firstly increments $a_i.level$ by one, moves to $v_{HOME}(a_j)$, and obtains informations about F_j. Then, a_i traverses in the new fragment

---

[†] In [18], each link is assumed to have a unique weight, but we realize it by node IDs.

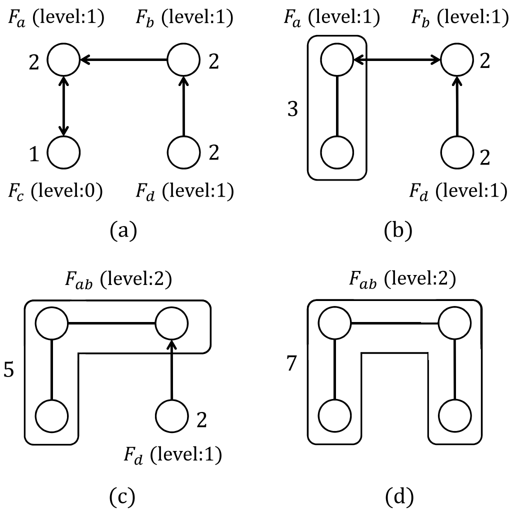

Fig. 5 An example of the fragment merge. Arrows represent merge requests.

and writes the updated level on every node. While traversing, when a_i observes a merge request by agent a_ℓ having a lower level than a_i, it absorbs fragment F_ℓ without changing its level. Concretely, it moves to $v_{HOME}(a_\ell)$, obtains information about F_ℓ, and makes F_ℓ as a part of the new fragment. If $a_i.id > a_j.id$ holds, a_i goes to $v_{HOME}(a_i)$ and waits for the next instruction (Sect. 3.5).

The last case is that $a_i.level > a_j.level$ holds, or $a_i.level = a_j.level$ holds but a_j does not request to merge with F_i. In this case, a_i stays at v_m^i until either of the above two cases occurs. While waiting, when agent a_j having a lower level than a_i requests to merge with F_i, a_i absorbs F_j and looks for the new fragment to merge.

By these behaviors, we can show that there exists at least 2^i agents in a fragment of level i (Lemma 5). Thus, by executing such merge processes at most $\lceil \log g \rceil$ times or until there exist no neighboring fragment, that is, all agents belong to the same fragment, at least g agents exist in each merged fragment. We call such fragments *final fragments*.

An example of merge processes is given in Fig. 5. For simplicity, in Fig. 5 (a) each fragment is represented as a circle. Each number near circles represents the number of agents in the fragment. In Fig. 5 (a), since F_b (resp., F_d) requests to merge with F_a (resp., F_b) but F_a (resp., F_b) tries to merge with another fragment, they wait until the configuration of F_a (resp., F_b) changes. On the other hand, F_a and F_c try to merge with each other but F_c's level is lower than F_a's level. In this case, F_c is absorbed and becomes part of F_a (Fig. 5 (a) to (b)). Then, F_a does not change its level and the number of agents in F_a that absorbed F_c is 3. Note that if F_a increases its level by one, it does not satisfy the condition that at least 2^i agents exist in a fragment with level i. After this, F_a tries to merge with another fragment F_b. Since they have the same level, they are merged to form the new fragment, say F_{ab} (Fig. 5 (b) to (c)). Then, the level of F_{ab} is incremented by one and the number of agents in F_{ab} is 5. After this, F_{ab} updates the contents in F_{ab} and then it finds that F_d requests to merge. Then, F_{ab} absorbs F_d and F_d becomes a part of F_{ab} (Fig. 5 (c) to (d)).

The pseudocode is described in Algorithm 2. Agent a_i and node v_j use the following variables:

- $a_i.candID$ and $a_i.candLevel$ are variables for storing the ID and the level of the agent managing the fragment that a_i tries to merge with.
- $a_i.isManager$ is a boolean variable to represent whether a_i is a manager of a fragment or not. That is, when a_i is a manager of some fragment, $a_i.isManager = true$ holds. When a_i's fragment is absorbed and a_i becomes a non-manager, $a_i.isManager$ is set to *false*.
- $a_i.lastAbsorb$ is a boolean variable to represent whether a_i absorbed some fragment in the last movement. The initial value of $a_i.lastAbsorb$ is *false*.
- $v_j.level$ is a variable for storing the level of the agent that manages a fragment including v_j. The initial value of $v_j.level$ is 0.
- $v_j.fUpdate$ is a boolean variable to represent whether the content of the fragment that some agent tries to merge is updated or not. The initial value of $v_j.fUpdate$ is *false*.
- $v_j.request_{merge}[\,]$ is an array for storing IDs and levels of agents that try to merge with the fragment containing v_j.

In Algorithm 2, a_i uses procedures *merge*() and *absorb*() to merge or absorb a fragment, whose pseudocodes are given in Procedures 1 and 2, respectively. Note that, in Algorithm 2 and Procedures 1, each agent basically traverses its fragment in the depth-first manner, but we omit the detail of the description (the movement is executed similarly to that of Algorithm 1). Particularly, after a_i with fragment F_i absorbs some fragment, it tries to merge with another neighboring fragment such that it finds for the first time in the depth-first search, instead of the fragment F'_j such that the MOE of a_i connects F_i and F'_j (lines 7-12). This is because, a_i requires $O(n)$ moves to find the MOE (or F'_j). Hence, if a_i absorbs fragments many times and requests to merge with such fragments, it may require $O(kn + m)$ total moves. On the other hand, if a_i executes the depth-first search and requests to merge with the fragment found for the first time, the total moves can be bounded by $O(n \log g)$ (Lemma 6). In addition, when $a_i.level = 0$ holds and it tries to merge with some fragment F_j, the manager agent a_j of fragment F_j may be still executing the first part. Then, since $v_m^j.level = 0$ holds, a_i waits at v_m^i until the a_j's ID is written on v_m^i's whiteboard or $v_m^i.fUpdate = true$ holds (lines 20 and 21). We have the following lemmas for Algorithm 2.

Lemma 5: When Algorithm 2 finishes, there exist at least g agents in each final fragment.

proof. At first, we show that there exist at least 2^l agents in the fragment managed by some agent a_i with $a_i.level = l$. We prove it by induction on the levels of agents. For the case of $a_i.level = 0$, clearly there exists only one ($= 2^0$) agent a_i in F_i. For the case of $a_i.level = l$, we assume that there exist

Algorithm 2 Fragment merge

```
Behavior of Agent $a_i$
1:  $a_i.level = 0, a_i.isManager = true, lastAbsorb = false$
2:  while $(a_i.level \neq \lceil \log g \rceil) \vee (a_i.NF \neq \emptyset)$ do
3:    if ($lastAbsorb = false$) then
4:      let $nf_{min}$ be the lexicographically minimum element in $a_i.NF$
5:      let $v_m^i$ (resp., $v_m^j$) be the node in $F_i$ (resp., $F_j$) included in $nf_{min}$
6:      go to $v_m^j$ through the path along $F_i$
7:    else
8:      $lastAbsorb = false$
9:      let $v_m^i$ be the node that is visited first by $a_i$ in the depth-first traversal of $F_i$ among the nodes adjacent to neighboring fragments. Let $v_m^j$ be the adjacent node in the neighboring fragment, say $F_j$.
10:     if link $(v_m^i, v_m^j)$ has the minimum weight
        among edges connecting $F_i$ and $F_j$ then move to $v_m^j$
11:     else go to line 9
12:   end if
13:   $a_i.candID = v_m^j.agent, a_i.candLevel = v_m^j.level$
14:   add $(a_i.id, a_i.level)$ to $v_m^j.request_{merge}[\,]$ and return to $v_m^i$
15:   if $a_i.level < a_i.candLevel$ then // $a_i$ is absorbed by $F_j$
16:     return to $v_{HOME}(a_i)$ and set $a_i.isManager = false$
17:     terminate the fragment merge part and start the partial gathering part
18:   else if $(a_i.level = a_i.candLevel) \wedge$
      (there exists $a_i.candID$ in $v_m^i.request_{merge}[\,]$) then
19:     // two fragments are merged and the new fragment is created
20:     $merge()$
21:   else if $(a_i.level = a_i.candLevel)$
      $\wedge$ (there does not exist $a_i.candID$ in $v_m^i.request_{merge}[\,]$) then
22:     wait until $a_i.candID$ is added to $v_m^i.request_{merge}[\,]$ or
        $v_m^i.fUpdate = true$ holds
23:     if there exists $a_i.candID$ in $v_m^i.request_{merge}[\,]$ then
24:       $merge()$
25:     else // level of $F_j$ increases and $F_i$ is absorbed
26:       return to $v_{HOME}(a_i)$ and set $a_i.isManager = false$
27:       terminate the fragment merge part and start the partial gathering part
28:     end if
29:   else if ($a_i.level$ ¿ $a_i.candLevel$) then
30:     if there exists $a_i.candID$ in $v_m^i.request_{merge}[\,]$ then
31:       $absorb()$
32:     else
33:       wait until $a_i.candID$ is added to $v_m^i.request_{merge}[\,]$ or
          $v_m^i.fUpdate = true$ holds
34:       if there exists $a_i.candID$ in $v_m^i.request_{merge}[\,]$ then
35:         $absorb()$
36:       else
37:         $v_m^j.fUpdate = false$
38:         go to $v_m^j$ and update values of $a_i.candID$ and $a_i.candLevel$
39:         go to line 14
40:       end if
41:     end if
42:   end if
43: end while
44: terminate the fragment merge part and start the partial gathering part
```

at least 2^l agents in F_i. From lines 17 to 19 of Algorithm 2 and Procedure 1, a_i merges only with the fragment of level l. Then, a_i increases the value of $a_i.level$ by one and after the merge there exist at least $2^l + 2^l = 2^{l+1}$ agents in the merged fragment of level $l + 1$. Thus, after executing merge processes $\lceil \log g \rceil$ times, there exist at least g agents in each final fragment. □

Procedure 1 *merge*()

```
Behavior of Agent $a_i$
1:  if $(a_i.id > a_i.candID)$ then
2:    // another agent becomes a manager of the new fragment
3:    return to $v_{HOME}(a_i)$ and set $a_i.isManager = false$
4:    terminate the fragment merge part and start the partial gathering part
5:  else
6:    // $a_i$ becomes a manager of the new fragment
7:    $a_i.level = a_i.level + 1$
8:    go to $v_{HOME}(a_j)$ through the path along $F_j$
9:    obtain $V_j$, $L_j$, $a_j.NF$, and topology information about $F_j$
10:   $V_i = V_i \bigcup V_j, L_i = L_i \bigcup L_j \bigcup (v_m^i, v_m^j), a_i.NF = a_i.NF \bigcup a_j.NF$
11:   delete elements including $a_j.id$ from $a_i.NF$
12:   while $a_i$ does not visit all nodes in $F_i$ from the begining of this procedure do
13:     leave the current and move to the next node $v_j$ in depth-first manner so that $a_i$ does not visit a node not in $F_i$
14:     if $v_j.level \neq a_i.level$ then $v_j.level = a_i.level$
15:     if there exists an element in $v_j.request_{merge}[\,]$ then
16:       for each element $(a_\ell.id, a_\ell.level)$ do
17:         let $v_m^\ell$ be the node in $F_\ell$ connecting to $v_j$
18:         go to $v_m^\ell$ and set $v_m^\ell.fUpdate = true$
19:         if $a_i.level > a_\ell.level$ then
20:           go to $v_{HOME}(a_\ell)$ through the path along $F_\ell$
21:           obtain $V_\ell$, $L_\ell$, $a_\ell.NF$, and topology information about $F_\ell$
22:           $V_i = V_i \bigcup V_\ell$, $L_i = L_i \bigcup L_\ell \bigcup (v_j, v_m^\ell)$, $a_i.NF = a_i.NF \bigcup a_\ell.NF$
23:           delete elements including $a_\ell.id$ from $a_i.NF$
24:         else
25:           return to $v_j$
26:         end if
27:       end for
28:     end if
29:   end while
30: end if
```

Procedure 2 *absorb*()

```
Behavior of Agent $a_i$
1: $lastAbsorb = true$
2: go to $v_{HOME}(a_j)$ through the path along $F_j$
3: obtain $V_j$, $L_j$, $a_j.NF$, and topology information about $F_j$
4: $V_i = V_i \bigcup V_j, L_i = L_i \bigcup L_j \bigcup (v_m^i, v_m^j), a_i.NF = a_i.NF \bigcup a_j.NF$
5: delete elements including $a_j.id$ from $a_i.NF$
```

Lemma 6: The total number of agent moves to execute Algorithm 2 is $O(n \log g)$.

proof. In the proof, we use the fact that each node is managed by exactly one manager agent at each level and each manager executes the merge process by traversing in its own fragment. Hence, the following analysis holds for the total moves of all the agents.

At the beginning of the merge process of each level, each manager agent a_i with fragment F_i firstly tries to find some fragment F_j and merge it. This requires $O(n)$ total moves for all the agents since each link between nodes in the same fragment is passed by at most once and each link connecting two fragments is passed by at most four times. After this, if F_i is absorbed, a_i goes to $v_{HOME}(a_i)$ and waits for the next instruction, which requires $O(n)$ total moves for all the agents since each link of F_i is passed by at most once. If a_i absorbs F_j, from Procedure *absorb*() a_i traverses F_i until F_i is absorbed or merged. This requires $O(n)$ total moves for all the agents because 1) F_i forms a tree topology, and 2) a_i traverses F_i in the depth-first manner and each link of F_i is passed by at most twice. If the two fragments are merged and a_i becomes a manager of the new fragment, it traverses the new fragment and updates the contents of the whiteboard of every node. This requires $O(n)$ total moves for all the agents since a_i traverses the new fragment in depth-first manner and each link is passed by at most twice. Hence, agents require $O(n)$ total moves to execute the merge process of any level. Since the level grows up to at most $\lceil \log g \rceil$, the total moves is at most $O(n \log g)$. Hence, we have the lemma. □

3.5 The Third Part: Partial Gathering in Each Merged Fragment

By Lemma 5, after finishing the merge agents can see each final fragment as a tree topology containing at least g agents. Hence, they execute the g-partial gathering algorithm for trees [17] in each final fragment independently to solve the problem. In [17], several leader agents instruct non-leader agents which node they should meet at. Thus, if each manager and each non-manager behave as a leader and a non-leader, respectively, they can solve the problem. By [17], this part can be achieved in $O(gn)$ total moves. By this fact and Lemmas 4 and 6, we have the following theorem.

Theorem 2: In arbitrary networks, the proposed algorithm solves the g-partial gathering problem in $O(gn + m)$ total moves. □

4. Conclusion

We considered the g-partial gathering problem in arbitrary networks. We proposed an algorithm to solve the g-partial gathering problem in $O(gn+m)$ total moves, which is asymptotically optimal in terms of total moves. As a future work, we want to clarify the influence of the memory requirement per agent and per node to the total number of moves.

References

[1] R.S. Gray, G. Cybenko, D. Kotz, A. Peterson, and D. Rus, "D'Agents: Applications and performance of a mobile-agent system," Softw: Pract. Exper., vol.32, no.6, pp.543–573, 2002.

[2] J. Baumann, F. Hohl, K. Rothermel, and M. Straßer, "Mole — Concepts of a mobile agent system," World Wide Web, vol.1, no.3, pp.123–137, 1998.

[3] D.B. Lange and M. Oshima, "Seven good reasons for mobile agents," Commun. ACM, vol.42, no.3, pp.88–89, 1999.

[4] G. Cabri, L. Leonardi, and F. Zambonelli, "Mobile agent coordination for distributed network management," Journal of Network and Systems Management, vol.9, no.4, pp.435–456, 2001.

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

[6] P. Flocchini, E. Kranakis, D. Krizanc, N. Santoro, and C. Sawchuk, "Multiple mobile agent rendezvous in a ring," Proc. 6th Latin American Theoretical Informatics, vol.2976, pp.599–608, 2004.

[7] P. Fraigniaud and A. Pelc, "Deterministic rendezvous in trees with little memory," Proc. 22nd International Symposium on Distributed Computing, vol.6950, pp.242–256, 2008.

[8] D. Baba, T. Izumi, F. Ooshita, H. Kakugawa, and T. Masuzawa, "Linear time and space gathering of anonymous mobile agents in asynchronous trees," Theor. Comput. Sci., vol.478, pp.118–126, 2013.

[9] E. Kranakis, D. Krizanc, and E. Markou, "Mobile agent rendezvous in a synchronous torus," Proc. 8th Latin American Theoretical Informatics, vol. 3887, pp.653–664, 2006.

[10] J. Czyzowicz, A. Kosowski, and A. Pelc, "How to meet when you forget: Log-space rendezvous in arbitrary graphs," Distrib. Comput., vol.25, no.2, pp.165–178, 2012.

[11] Y. Dieudonné and A. Pelc, "Anonymous meeting in networks," Algorithmica, vol.74, no.2, pp.908–946, 2016.

[12] G.D. Marco, L. Gargano, E. Kranakis, D. Krizanc, A. Pelc, and U. Vaccaro, "Asynchronous deterministic rendezvous in graphs," Theor. Comput. Sci., vol.355, no.3, pp.315–326, 2005.

[13] J. Czyzowicz, A. Pelc, and A. Labourel, "How to meet asynchronously (almost) everywhere," ACM Trans. Algorithms, vol.8, no.4, p.37, 2012.

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

[15] Y. Dieudonné, A. Pelc, and V. Villain, "How to meet asynchronously at polynomial cost," SIAM J. Comput., vol.44, no.3, pp.844–867, 2015.

[16] M. Shibata, S. Kawai, F. Ooshita, H. Kakugawa, and T. Masuzawa, "Partial gathering of mobile agents in asynchronous unidirectional rings," Theor. Comput. Sci., vol.617, pp.1–11, 2016.

[17] M. Shibata, F. Ooshita, H. Kakugawa, and T. Masuzawa, "Move-optimal partial gathering of mobile agents in asynchronous trees," Theor. Comput. Sci., vol.705, pp.9–30, 2018.

[18] G. Gallager, A. Humblet, and P.M. Spira, "A distributed algorithm for minimum-weight spanning trees," ACM Trans. Program. Lang. Syst. vol.5, no.1, pp.66–77, 1983.

[19] S. Das, P. Flocchini, A. Nayak, and N. Santoro, "Effective election for anonymous mobile agents," International Symposium on Algorithms and Computation, vol.4288, pp.732–743, 2006.

[20] S. Das, P. Flocchini, S. Kutten, A. Nayak, and N. Santoro, "Map construction of unknown graphs by multiple agents," Theor. Comput. Sci., vol.385, no.1-3, pp.34–48, 2007.

Masahiro Shibata received BE, ME, and DE degrees in Computer Science from Osaka University, in 2012 2014, and 2017, respectively. Since 2017, he has been an Assistant Professor in Kyushu Institute of Technology. His research interests include distributed algorithms.

Daisuke Nakamura received BE degree in Computer Science from Osaka University, in 2015. His research interests include distributed algorithms.

Fukuhito Ooshita received ME and DE degrees in Computer Science from Osaka University, in 2002 and 2006, respectively. Between 2003 and 2015, he worked as an Assistant Professor in the Graduate School of Information Science and Technology at Osaka University. Since 2015, he has been an Associate Professor in the Graduate School of Information Science, Nara Institute of Science and Technology (NAIST). His research interests include parallel algorithms and distributed algorithms. He is a member of ACM, IEEE, IEICE, and IPSJ.

Hirotsugu Kakugawa received a BE degree in Engineering from Yamaguchi University, in 1990, and ME and DE degrees in Information Engineering from Hiroshima University, in 1992 and 1995, respectively. He is currently an Associate Professor in Osaka University. He is a member of IEEE and IPSJ.

Toshimitsu Masuzawa received BE, ME, and DE degrees in Computer Science from Osaka University, in 1982, 1984, and 1987, respectively. Between 1987 and 1994, he worked at Osaka University, and was an Associate Professor in the Graduate School of Information Science, Nara Institute of Science and Technology (NAIST) between 1994 and 2000. He is now a Professor in the Graduate School of Information Science and Technology, Osaka University. He was also a visiting Associate Professor in the Department of Computer Science, Cornell University, USA between 1993 and 1994. His research interests include distributed algorithms, parallel algorithms, and graph theory. He is a member of ACM, IEEE, IEICE, and IPSJ.