

Self-stabilizing Rendezvous of Synchronous Mobile Agents in Graphs

Fukuhito Ooshita^{1(✉)}, Ajoy K. Datta², and Toshimitsu Masuzawa³

¹ Graduate School of Information Science,
Nara Institute of Science and Technology, Ikoma, Japan
`f-ooshita@is.naist.jp`

² Department of Computer Science, University of Nevada, Las Vegas, USA

³ Graduate School of Information Science and Technology,
Osaka University, Suita, Japan

Abstract. We investigate self-stabilizing rendezvous algorithms for two synchronous mobile agents. The rendezvous algorithms make two mobile agents meet at a single node, starting from arbitrary initial locations and arbitrary initial states. We study deterministic algorithms for two synchronous mobile agents with different labels but without using any whiteboard in the graph. First, we show the existence of a self-stabilizing rendezvous algorithm for arbitrary graphs by providing a scheme to transform a non-stabilizing algorithm to a self-stabilizing one. However, the time complexity of the resultant algorithm is not bounded by any function of the graph size and labels. This raises the question whether there exist polynomial-time self-stabilizing rendezvous algorithms. We give partial answers to this question. We give polynomial-time self-stabilizing rendezvous algorithms for trees and rings.

Keywords: Mobile agents · Self-stabilization · Rendezvous · Gathering

1 Introduction

1.1 Background

In the *rendezvous problem*, two mobile agents (or simply, agents) initially located at different nodes must eventually meet at a single node. If the number of agents is more than two, the problem is called the *gathering problem*. Mobile agents may be software programs that can autonomously move in a distributed system, or robots that can move in a real world. The reason to achieve a rendezvous or gathering may be to share information previously collected by each mobile agent, or to divide and assign tasks to agents. The rendezvous and gathering problems are fundamental problems of mobile agents, and many algorithms have been proposed on various models [16].

This work was supported by JSPS KAKENHI Grant Numbers 26280022 and 26330084.

Since agents move around different places in a distributed system or a real world, they are exposed to various faults. To overcome such faults, recently some attempts have been made to design fault-tolerant rendezvous algorithms. Some notable fault-tolerant algorithms are delay faults [3], Byzantine faults [2, 9, 20], and crash faults [17].

In this paper, we focus on transient faults such as temporal memory corruption and erroneous initialization. To tolerate such faults, we develop *self-stabilizing* rendezvous algorithms. An algorithm is called self-stabilizing [11] if, starting from an arbitrary initial configuration, the system eventually reaches a legitimate configuration. Self-stabilizing rendezvous algorithms guarantee that even if each mobile agent starts from an arbitrary location and an arbitrary initial state, two agents will eventually meet at a single node. From this property, even when two agents become inconsistent due to transient faults, they can eventually achieve a rendezvous.

1.2 Related Work

The rendezvous and gathering problems have been extensively studied with various assumptions [16]. Various solutions are also considered to reduce various costs, e.g., time, number of moves, and memory requirements.

For fault-free systems, many rendezvous algorithms have been proposed for two synchronous agents. To achieve a rendezvous in symmetric graphs, it is necessary to make some assumptions to break the symmetry. In [8, 14, 19], rendezvous algorithms for arbitrary graphs were proposed on the assumption that two agents have different labels. For the case of no different labels, memory-efficient rendezvous algorithms were proposed for trees [5, 12] and arbitrary graphs [4] on the assumption that two agents start from some non-symmetric locations.

Recently fault-tolerant algorithms for agents are being explored. Chalopin et al. [3] proposed algorithms tolerant to delay faults, which prevent an agent from moving for some rounds. Dieudonné et al. [9] and Bouchard et al. [2] proposed Byzantine-tolerant algorithms, in which all correct agents meet at a single node even if some agents behave arbitrarily. Tsuchida et al. [20] reduced the time complexity of Byzantine-tolerant algorithms by assuming a whiteboard (a node memory where agents can leave information) and an authentication mechanism. Pelc [17] studied crash faults for systems such that agents can move at different speeds.

A few self-stabilizing algorithms have been proposed for mobile agents [1, 15]. Blin et al. [1] studied self-stabilizing naming and leader election, and Masuzawa and Tixeuil [15] studied self-stabilizing gossiping. Since an algorithm proposed in [15] guarantees that agents can meet each other, the algorithm also solves the rendezvous problem of two agents. However, unlike this work, these algorithms assume whiteboards where agents can leave information in nodes.

In a different context, gathering of oblivious mobile robots has been thoroughly studied in planes [10, 18] and in graphs [6, 7, 13]. Since oblivious robots do not have memories, the algorithms are almost self-stabilizing. However, differ-

ent from our work, these algorithms assume that a robot can obtain the locations of all other robots instantaneously.

1.3 Our Contributions

In this paper, we give several self-stabilizing rendezvous algorithms for graphs. We make some very common assumptions. Two agents have different labels ℓ_1 and ℓ_2 , behave synchronously, can start at different times, and cannot leave any information in nodes. The graph size (i.e., the number of nodes) is denoted by n , and it is unknown to agents.

First, we show the existence of a self-stabilizing rendezvous algorithm for arbitrary graphs. We show the proposition by designing a scheme to transform a non-stabilizing rendezvous algorithm to a self-stabilizing one. Since non-stabilizing rendezvous algorithms for arbitrary graphs are available in [8, 14, 19], this scheme gives a self-stabilizing rendezvous algorithm. However, the time complexity (i.e., the time required to achieve a rendezvous after both agents start the algorithm) is not bounded by any function of n , ℓ_1 , or ℓ_2 . This raises the question whether there exist polynomial-time self-stabilizing rendezvous algorithms.

Next, we give partial answers to the above question. That is, we give polynomial time self-stabilizing rendezvous algorithms for trees and rings. For trees, we give a self-stabilizing rendezvous algorithm with the time complexity of $O(n \cdot \min\{|\ell_1|, |\ell_2|\})$ rounds, which is a polynomial of the graph size and the length of the smaller label. For rings, we give a self-stabilizing rendezvous algorithm with the time complexity of $O(n\ell_1\ell_2)$ rounds, which is a polynomial of the number of nodes and the two labels.

1.4 Outline

In Sect. 2, we present the computing model and the problem we consider in this paper. In Sect. 3, we show the existence of a self-stabilizing rendezvous algorithm for arbitrary graphs. We give polynomial-time self-stabilizing rendezvous algorithms for trees and rings in Sects. 4 and 5, respectively. In Sect. 6, we briefly discuss an extension of our proposed algorithms to gathering of more than two agents. Concluding remarks are presented in Sect. 7.

2 Preliminaries

2.1 Network and Agents

A network is modeled by a connected undirected graph $G = (V, E)$, where V is a set of nodes and E is a set of communication links. The graph size is denoted by $n = |V|$. The degree of node v is defined as the number of incident links of v , and is denoted by \deg_v . A node v is a neighbor of w if $(v, w) \in E$ holds. A set of neighbors of v is denoted by N_v , i.e., $N_v = \{w | (v, w) \in E\}$. Nodes are anonymous, i.e., they do not have unique labels (or identifiers). On the

other hand, each link incident to node v is numbered locally at v by bijection $\lambda_v : \{(v, w) | w \in N_v\} \rightarrow \{1, 2, \dots, \deg_v\}$. Note that $\lambda_v(v, u) \neq \lambda_v(v, w)$ holds for distinct neighbors u and w of v . The numbering function is independent of that of other nodes. For a link (v, w) , $\lambda_v(v, w) \neq \lambda_w(v, w)$ may hold. We say $\lambda_v(v, w)$ is a port number (or port) of link (v, w) at node v .

There exist two agents a_1 and a_2 in the network. We assume that they start their actions from two different nodes. Every agent has its own memory, and they move with their memory. On the other hand, an agent cannot leave any information in any node. Each agent a_i is assigned a unique label, denoted by ℓ_i . We define $|\ell|$ as the length of label ℓ , i.e., $|\ell| = \lceil \log \ell \rceil$. An agent knows its own label, but does not know the label of the other agent. An agent can move from a node to its neighbor by choosing an outgoing port. That is, when an agent is at v and moves via port p , it moves to node w such that $p = \lambda_v(v, w)$ holds. When the agent reaches w , it can read the incoming port $\lambda_w(v, w)$. Agents know neither n nor the upper bound of n .

Each agent is modeled as a state machine (S, δ) . The first element S is a set of agent states, where each agent state is determined by the values of its variables in its memory. We assume that the memory of agents is unbounded, that is, S could be an infinite set. The second element δ is a deterministic state transition function, which decides the behavior of an agent. The input of δ is the current agent state, the label of the agent, the degree of the current node, and the incoming port. The output of δ is the next agent state, whether the agent stays or leaves, and the outgoing port if the agent leaves.

Two agents spontaneously start an algorithm possibly at different times. After agents start an algorithm, they execute in synchronous rounds. That is, if an agent decides to move to a neighbor in a round, it completes the movement before the beginning of the next round. If agent a_i starts before a_j , we say a_i is the *first agent* and a_j is the *second agent*. The first agent does not meet the second agent before the second agent starts the algorithm.

2.2 Self-stabilizing Rendezvous

The goal of the *rendezvous problem* is to make two agents meet at a single node, i.e., two agents stay at the same node at the same time. As it is often assumed in the literature in the synchronous setting, two agents cannot meet or notice that when they move through the same link in the opposite directions. In this paper, we solve the rendezvous problem in a self-stabilizing manner. That is, even if agents start an algorithm from an arbitrary (inconsistent) initial state, they eventually meet at a single node. We assume that, when two agents stay at the same node at the same time, they can notice this fact. Thus, they can notice the completion of the rendezvous problem and terminate the algorithm. We define the time complexity as the number of rounds required to achieve a rendezvous after the second agent starts an algorithm.

3 A Self-stabilizing Rendezvous Algorithm for Arbitrary Graphs

In this section, we show the existence of a self-stabilizing rendezvous algorithm for arbitrary graphs. We present a scheme to transform a non-stabilizing rendezvous algorithm to a self-stabilizing one. As described in Sect. 1, many non-stabilizing rendezvous algorithms are proposed in literature [8, 14, 19]¹. In particular, algorithms in [14, 19] guarantee that two agents achieve a rendezvous in a polynomial time of the graph size and labels. Let Alg be such a non-stabilizing rendezvous algorithm and $Alg(\ell)$ be the procedure that the agent with label ℓ executes in algorithm Alg . The algorithm guarantees that, when two agents execute $Alg(\ell_1)$ and $Alg(\ell_2)$ from their designated initial states, they eventually meet at a single node. In addition, the time required to achieve a rendezvous is bounded by function F of the graph size and labels, i.e., two agents meet at a single node in $F(n, \ell_1, \ell_2)$ rounds after the second agent starts the algorithm. For example, we have $F(n, \ell_1, \ell_2) = \tilde{O}(n^{15} + (\min\{|\ell_1|, |\ell_2|\})^3)$ and $F(n, \ell_1, \ell_2) = \tilde{O}(n^5 \cdot \min\{|\ell_1|, |\ell_2|\})$ for the algorithms in [14, 19], respectively.

We construct a self-stabilizing rendezvous algorithm by using Alg . The pseudocode is given in Algorithm 1. This algorithm consists of two simple ideas. First, since each agent may start Alg from an arbitrary initial state, it breaks rounds into multiple phases and resets variables for Alg in the beginning of each phase. After two agents reset their states, if both agents execute Alg for $F(n, \ell_1, \ell_2)$ rounds without resetting, they can achieve a rendezvous. To achieve this, each agent doubles the duration of a phase whenever it starts a new phase. So, the duration of a phase eventually becomes sufficiently long and two agents can achieve a rendezvous.

Algorithm 1. SSgraph

Variables

- 1: **var** k ; // the current phase number
- 2: **var** h ; // the current round number in the current phase
- 3: **var** var ; // variables for Alg

Behavior of Agent a_i in each round

- 4: **if** another agent stays at the same node **then**
 - 5: terminate;
 - 6: **if** $h \geq 2^k$ **then**
 - 7: $k = k + 1$; $h = 0$; initialize var ; // start a new phase
 - 8: **end if**
 - 9: // execute the k -th phase
 - 10: $h = h + 1$;
 - 11: execute the h -th round of $Alg(\ell_i)$;
-

¹ Some algorithms in literature may be actually self-stabilizing. However, since their self-stabilizing property is not proven explicitly, we regard them as non-stabilizing algorithms.

Theorem 1. *Algorithm **SSgraph** is a self-stabilizing rendezvous algorithm for arbitrary graphs.*

Proof. Let r_0 be the first round such that both agents reset *Alg* at least once. Assume that, in round r_0 , a_1 executes the h_1 -th round of the k_1 -th phase and a_2 executes the h_2 -th round of the k_2 -th phase. Without any loss of generality, we assume that $k_1 > k_2$ or $k_1 = k_2 \wedge h_1 \geq h_2$. Let $r_d = \sum_{h=k_2}^{k_1} 2^h$. From the algorithm, for each $k \geq k_1$, agent a_2 starts the k -th phase at most r_d rounds later than a_1 .

Let k^* be the minimum k such that $k > k_1$ and $r_d + F(n, \ell_1, \ell_2) \leq 2^k$ holds. Since a_2 starts the k^* -th phase at most r_d rounds later than a_1 and the duration of the k^* -th phase is 2^{k^*} , both agents simultaneously execute the k^* -th phase for at least $F(n, \ell_1, \ell_2)$ rounds after a_2 starts the k^* -th phase. Therefore, a_1 and a_2 can achieve a rendezvous in the k^* -th phase or earlier. \square

Remark 1. In the model of this paper, when an agent enters a node, it can obtain the incoming port number (i.e., the port number at which it enters the node). However, since an algorithm in [19] does not use the incoming port number, **SSgraph** based on this algorithm also does not use the incoming port number. This means a self-stabilizing rendezvous algorithm exists even when agents cannot obtain the incoming port number.

Unfortunately the time complexity of Algorithm **SSgraph** is not bounded in spite of the fact that non-stabilizing algorithms in [14, 19] achieve a rendezvous in polynomial time from some designated initial states. This is because every non-stabilizing rendezvous algorithm uses an estimation of the graph size and the time complexity depends on the estimation. To explain the details, we give a common behavior of every non-stabilizing rendezvous algorithm. In such an algorithm, agents use a variable, say *est*, to store an estimated graph size. Initially agents store a small value in *est*, and behave as if the graph size is at most *est*. The number of rounds depends on *est*. If the actual graph size is at most *est*, agents achieve a rendezvous. If agents do not achieve a rendezvous, they increase *est* gradually. Eventually, *est* exceeds the actual graph size, and at that time agents achieve a rendezvous. In non-stabilizing algorithm, *est* does not become so large, and hence the time complexity is bounded by some function of n and other parameters. However, in self-stabilizing algorithms, agents may start the algorithm from an initial state such that *est* is much higher than n . In this case, the number of required rounds cannot be bounded by any function of n and other parameters. If variable k in **SSgraph** is large, agents can execute such an algorithm for a long time.

Remark 2. Note that Algorithm **SSgraph** requires an unbounded memory. However, if agents know the upper bound of the graph size n , we can obtain a simple self-stabilizing algorithm that uses a bounded memory. Let N be the known upper bound of the graph size. We consider a non-stabilizing rendezvous algorithm *Alg* such that $F(n, \ell_1, \ell_2)$ depends on only n and $\min\{\ell_1, \ell_2\}$ like [14, 19]. In this case, each agent a_i can compute the upper bound of $F(n, \ell_1, \ell_2)$, say F_i^* .

To transform Alg to a self-stabilizing algorithm, agent a_i repeatedly executes a phase in which it executes Alg for $2F_i^*$ rounds and then initializes its variables. By this behavior, both agents can execute Alg for $\min\{F_1^*, F_2^*\}$ rounds without resetting, and thus they can achieve a rendezvous. Since agents execute Alg for a bounded number of rounds, the required memory is bounded. Note that the time complexity depends on N , which may be much higher than n .

Since the time complexity of **SSgraph** is unbounded, we need a self-stabilizing rendezvous algorithm with a polynomial time complexity. In the following sections, we give such self-stabilizing rendezvous algorithms for trees and rings.

4 A Polynomial-Time Self-stabilizing Rendezvous Algorithm for Trees

In this section, we give a polynomial time self-stabilizing rendezvous algorithm **SStree** for trees. We develop the algorithm by extending algorithm **Extend-Labels** [8], which realizes rendezvous in a two-node graph. In **Extend-Labels**, for each round, each agent decides to move or stay based on its label. Algorithm **Extend-Labels** guarantees that in some round, one agent moves to its neighbor and another agent stays at a node, thereby two agents achieve a rendezvous. We apply this decision mechanism to our algorithm. In **SStree**, each agent explores a tree instead of a single move or stay, and decides the direction of the exploration based on its label. The decision mechanism of **Extend-Labels** guarantees that two agents eventually explore the tree in the opposite directions at the same time. During this exploration, two agents achieve a rendezvous.

We give the details of **SStree**. The pseudocode is given in Algorithm 2. First, we explain the behavior of **Extend-Labels**. For label ℓ of an agent, its extended label $M(\ell)$ is defined as follows. Let $a_1a_2 \cdots a_{|\ell|}$ be the binary representation of ℓ , $M(\ell) = (10a_1a_1a_2a_2 \cdots a_{|\ell|}a_{|\ell|})^*$ where s^* is an infinite sequence that repeats sequence s infinite times. For example, since the binary representation of 5 is 101, we have $M(5) = 1011001110110011 \cdots$. Agents can start **Extend-Labels** at different times. After an agent starts the algorithm, a_i moves in the k -th round if the k -th bit of $M(\ell_i)$ is 1; otherwise, it stays for one round. The following lemma guarantees the correctness of **Extend-Labels**.

Lemma 1 [8]. *Let ℓ_1 and ℓ_2 be different labels and $\ell^* = \min\{\ell_1, \ell_2\}$. Assume that M_1 is a suffix of $M(\ell_1)$. There exists an index k such that the k -th bits of M_1 and $M(\ell_2)$ are different and $k \leq 2|\ell^*| + 6$.*

Note that, when each agent a_i starts the algorithm from an arbitrary initial state, a_i may refer to extended label $M(\ell_i)$ from the middle. Even in this case, the agents can achieve a rendezvous by the following lemma.

Lemma 2. *Let ℓ_1 and ℓ_2 be different labels and $\ell^* = \min\{\ell_1, \ell_2\}$. Assume that M_1 and M_2 are suffixes of $M(\ell_1)$ and $M(\ell_2)$, respectively. There exists an index k such that the k -th bits of M_1 and M_2 are different and $k \leq 4|\ell^*| + 7$.*

Algorithm 2. SStree**Variables**

```

1: var mode; // which part  $a_i$  executes ( $mode \in \{init, phase\}$ )
2: var k; // the current phase number
3: var h; // the current round number in the current phase
4: var n; // the estimated graph size
5: var Top; // the topology information
Behavior of Agent  $a_i$  at each round
6: // check completion of rendezvous
7: if another agent stays at the same node then
8:   terminate;
9: end if
10: // check consistency of the topology information
11: if Top is inconsistent with the current node then
12:   mode = init; initialize Top;
13: // collect the topology information if mode = init.
14: if mode = init then
15:   update the topology information in Top;
16:   if Top includes the complete topology then
17:     mode = phase; k = 1; h = 0;
18:     n = the graph size in Top;
19:   else
20:     execute one basic move;
21:   end if
22: // execute the k-th phase if mode = phase
23: else
24:   if  $h \geq 8(n - 1) + 2$  then
25:     k = k + 1; h = 0; // start a new phase
26:   end if
27:   h = h + 1;
28:   if the k-th bit of  $M(\ell_i)$  is 1 then //  $M(\ell_i)$  is the extended label of  $\ell_i$ 
29:     // basic phase
30:     execute one basic move;
31:   else
32:     // reverse phase
33:     if  $h \neq 2(n - 1) + 1$  and  $h \neq 6(n - 1) + 2$  then
34:       execute one reverse move;
35:     else
36:       stay for one round;
37:     end if
38:   end if

```

Proof. Without any loss of generality, we assume $\ell_1 > \ell_2 = \ell^*$. For infinite sequence $s = s_1 s_2 \dots$ and positive integer x , we define $S(s, x)$ as suffix $s_x s_{x+1} \dots$ of s . Since $M(\ell_2)$ is a repetition of a sequence of length $2|\ell_2| + 2 = 2|\ell^*| + 2$, there exists $k' \leq 2|\ell^*| + 2$ such that $S(M_2, k') = M(\ell_2)$. Since $S(M_1, k')$ is a suffix of $M(\ell_1)$, from Lemma 1, there exists an index k'' such that the k'' -th bits of $S(M_1, k')$ and $S(M_2, k')$ are different and $k'' \leq 2|\ell^*| + 6$. This implies

that when $k = k' + k'' - 1$, the k -th bits of M_1 and M_2 are different. From $k = k' + k'' - 1 \leq 4|\ell^*| + 7$, the lemma holds. \square

Algorithm **SStree** consists of multiple phases, and agent a_i decides the behavior of the k -th phase based on the k -th bit of $M(\ell_i)$. In each phase, an agent explores the tree by using *basic moves* or *reverse moves*. The basic move is a traditional technique, which makes an agent explore the tree using the DFS traversal. In the basic move, when the agent arrives at node v from port p (i.e., it arrives at v via edge (u, v) such that $\lambda_v(u, v) = p$), it leaves v from port $(p \bmod \deg_v) + 1$ in the next move (i.e., it leaves v via edge (v, w) such that $\lambda_v(v, w) = (p \bmod \deg_v) + 1$). The agent starts the first move of the basic move by leaving port 1. The reverse move is the opposite move of the basic move. That is, when the agent arrives at node v from port p , it leaves v from port $p - 1$ if $p > 1$ and port \deg_v if $p = 1$. The agent starts the first move of the reverse move by leaving port \deg_v , where v is its current node. Since the length of the DFS traversal is $2(n - 1)$, an agent can explore the tree by $2(n - 1)$ basic moves or $2(n - 1)$ reverse moves.

In the k -th phase of **SStree**, agent a_i explores a tree by the basic moves if the k -th bit of $M(\ell_i)$ is 1; otherwise, it explores the tree by the reverse moves. Lemma 2 guarantees that eventually one agent executes basic moves and the other agent executes reverse moves at the same time. However, one exploration is not sufficient to achieve a rendezvous because the starting rounds of each phase are not synchronized. In addition, agents may move through the same link in the opposite directions without achieving a rendezvous. To overcome these problems, agent a_i behaves in its k -th phase as follows.

- Assume that the k -th bit of $M(\ell_i)$ is 1. In this case, a_i executes $8(n - 1) + 2$ basic moves. That is, a_i explores the tree four times by basic moves and executes two additional basic moves. We call it a basic phase.
- Assume that the k -th bit of $M(\ell_i)$ is 0. a_i first explores the tree once by reverse moves and then stays for one round. After that, a_i explores the tree two times by reverse moves and then stays for one round. Finally, a_i explores the tree once by reverse moves. We call it a reverse phase.

Later, we will prove that these behaviors achieve a rendezvous.

To execute the above procedures, agent a_i should obtain the value of n . To do this, before a_i executes the above phases, a_i executes basic moves and records topology information of the tree in variable *Top*. a_i records every visited node, every observed port (associating with a node), every passed link (associating with nodes and ports) in *Top*. Eventually, a_i explores the tree and obtains the complete topology information. That is, a_i can realize that it has passed through every port (i.e., every node and every link) in the tree. This is done in $2(n - 1)$ rounds. However, a_i may start the algorithm from an arbitrary initial state, that is, it may have wrong topology information in *Top*. For this reason, a_i checks consistency of the topology information in *Top* after each movement. That is, when a_i moves to an already visited node, it compares the incoming port and the degree of the node with the recorded ones in *Top*. If these are different, the

recorded topology information in Top is inconsistent. In this case, a_i discards the current information in Top and collects the topology information again. Note that, if the topology information in Top is inconsistent, a_i finds the inconsistency before it completes one exploration.

In the following, we show the correctness and analyze the time complexity.

Lemma 3. *Each agent obtains consistent topology information in at most $8(n - 1)$ rounds from an arbitrary initial state.*

Proof. From an arbitrary initial state, an agent finishes recording the topology information and moves to the first phase in $2(n - 1)$ rounds. Note that this topology information may be inconsistent because an agent can start the algorithm with an inconsistent partial topology information. Once an agent starts a phase, it finds inconsistency before completing a single exploration if the topology information is inconsistent. This requires at most $4(n - 1)$ rounds because an agent makes $2(n - 1)$ successive basic moves or $2(n - 1)$ successive reverse moves during $4(n - 1)$ successive rounds. After an agent restarts the algorithm, it obtains consistent topology information in $2(n - 1)$ rounds. Therefore, each agent obtains consistent topology information in at most $8(n - 1)$ rounds. \square

Theorem 2. *Algorithm $SStree$ is a self-stabilizing rendezvous algorithm for trees. The time complexity of $SStree$ is $O(n \cdot |\ell^*|)$, where $\ell^* = \min\{\ell_1, \ell_2\}$.*

Proof. Let round r_0 be the first round such that both agents start phases with consistent topology information. From Lemma 3, such a round comes in at most $8(n - 1)$ rounds after the second agent starts the algorithm. After round r_0 , since the duration of each phase is $8(n - 1) + 2$ rounds, each phase of an agent overlaps with a phase of the other agent for at least $4(n - 1) + 1$ rounds. From Lemma 2, there exist such overlapped phases in which one agent executes a basic phase and the other agent executes a reverse phase, and these phases come within $4|\ell^*| + 7$ phases after round r_0 . Without any loss of generality, a_1 executes a basic phase and a_2 executes a reverse phase.

First, consider the case when the overlapped phase of a_1 starts earlier than a_2 . Let round r_1 be the round in which a_2 starts the overlapped phase. After r_1 , while a_1 executes $4(n - 1) + 1$ basic moves, a_2 executes $2(n - 1)$ reverse moves, stays for one round, and executes $2(n - 1)$ reverse moves. During the first $2(n - 1)$ rounds, a_1 and a_2 explore a tree once in the opposite directions. This implies that a_1 and a_2 achieve a rendezvous, or a_1 and a_2 move through the same link in the opposite directions. In the latter case, a_1 and a_2 explore the tree once more but a_2 changes its visiting timing in one round. Consequently, a_1 and a_2 achieve a rendezvous during the second exploration.

Next, consider the case that the overlapped phase of a_1 starts no earlier than a_2 . Let round r_1 be the round in which a_2 starts the $(4(n - 1) + 1)$ -th round of the overlapped phase. After r_1 , while a_1 executes $4(n - 1) + 1$ basic moves, a_2 executes $2(n - 1)$ reverse moves, stays for one round, and executes $2(n - 1)$ reverse moves. Hence, similar to the first case, a_1 and a_2 achieve a rendezvous.

From the above discussions, two agents achieve a rendezvous in the overlapped phase. Therefore, after the second agent starts the algorithm, two agents achieve a rendezvous in $8(n - 1) + (4|\ell^*| + 7)(8(n - 1) + 2) = O(n \cdot |\ell^*|)$ rounds. \square

Remark 3. Algorithm **SStree** requires an unbounded memory. However, if agents know the upper bound of the graph size, we can bound the memory size of **SStree**.

5 A Polynomial-Time Self-stabilizing Rendezvous Algorithm for Rings

In this section, we give a polynomial-time self-stabilizing rendezvous algorithm **SSring** for rings. Unlike the trees, agents cannot compute the ring size without leaving marks on nodes. This implies that agents cannot recognize the completion of an exploration, and thus, we must use an approach different from the one for the trees. In **SSring**, two agents achieve a rendezvous by moving at different speeds. In the following, we explain the details of **SSring**. The pseudocode of **SSring** is given in Algorithm 3. In this section, we assume that each agent decides its forward and backward direction at each node by its port numbers. However, this direction is not identical for two agents. That is, two agents may decide opposite directions as their forward directions.

For simplicity, first assume two agents decide the same direction as their forward directions. In this case, the following algorithm achieves a rendezvous.

- Each a_i repeats the following: a_i stays for ℓ_i rounds and then moves forward.

Clearly, a_1 and a_2 move forward once in $\ell_1 + 1$ and $\ell_2 + 1$ rounds, respectively. This implies that the distance between a_1 and a_2 decreases by at least $|\ell_1 - \ell_2| \geq 1$ in $(\ell_1 + 1)(\ell_2 + 1)$ rounds, and thus, they achieve a rendezvous in $n(\ell_1 + 1)(\ell_2 + 1)$ rounds.

However, two agents may decide opposite directions as their forward directions, and in this case, they can move through the same link in the opposite directions. To overcome this situation, we introduce a sweeping operation. In a sweeping operation, an agent moves forward to the next node and then moves backward to the current node. With this change, whenever a_i needs to move forward to the next node, it first repeats the sweeping operation ℓ_i times; then it moves forward to the next node. By this behavior, when two agents try to move through the same link at the same time, they repeat the sweeping operation at different times. Thus, an agent cannot miss the other agent and achieves a rendezvous.

Figure 1 shows an example with the sweeping operations. The solid and dotted arrows represent the planned behaviors of agents with labels four and three, respectively. The figure shows the situation when the two agents decide opposite directions as their forward directions and appear in two neighboring nodes. A horizontal arrow means the agent stays at its current node, and a diagonal

Algorithm 3. SSring**Variables**

1: **var** h ; // the current round in the current phase

Behavior of Agent a_i at each round

2: // check completion of rendezvous

3: **if** another agent stays at the same node **then**

4: terminate;

5: **end if**

6: **if** $h \geq 3\ell_i + 1$ **then**

7: $h = 0$; // start a new phase

8: **end if**

9: $h = h + 1$;

10: **if** $1 \leq h \leq \ell_i$ **then**

11: stay for one round;

12: **else if** $\ell_i + 1 \leq h \leq 3\ell_i$ **then**

13: **if** $(h - \ell_i) \bmod 2 = 1$ **then**

14: move forward;

15: **else**

16: move backward;

17: **end if**

18: **else if** $h = 3\ell_i + 1$ **then**

19: move forward;

20: **end if**

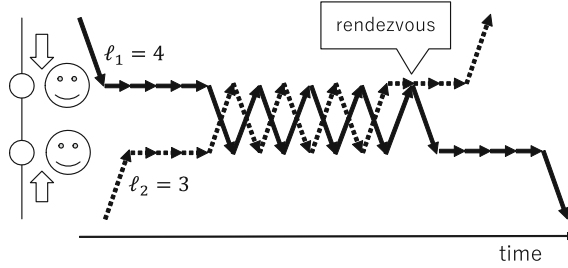


Fig. 1. An example of SSring.

arrow means the agent moves to its forward or backward node. That is, each agent with label ℓ_i stays for ℓ_i rounds and then repeats a sweeping operation ℓ_i times (i.e., it repeats a forward and a backward moves ℓ_i times). After that, it moves forward to the next node. If the end points of the arrows overlap, the two agents can achieve a rendezvous. Readers can observe that two agents achieve a rendezvous even if they appear in two neighboring nodes at any time.

In the following, we show the correctness and analyze the time complexity.

Theorem 3. *Algorithm SSring is a self-stabilizing rendezvous algorithm for rings. The time complexity of SSring is $O(n\ell_1\ell_2)$.*

Proof. First, assume that two agents decide the same direction as their forward directions. In this case, a_1 and a_2 move once in the same direction in $3\ell_1 + 1$ and $3\ell_2 + 1$ rounds, respectively. This implies that the distance between a_1 and a_2 decreases by at least $|\ell_1 - \ell_2| \geq 1$ in $(3\ell_1 + 1)(3\ell_2 + 1)$ rounds, and thus, they achieve a rendezvous in $n(3\ell_1 + 1)(3\ell_2 + 1) = O(n\ell_1\ell_2)$ rounds.

Next assume that two agents decide opposite directions as their forward directions. In this case, a_1 and a_2 move once in the opposite directions in $3\ell_1 + 1$ and $3\ell_2 + 1$ rounds, respectively. This implies that the distance between a_1 and a_2 decreases by at least $3\ell_1 + 3\ell_2 + 2$ in $(3\ell_1 + 1)(3\ell_2 + 1)$ rounds, and thus, the distance between them becomes one in $n(3\ell_1 + 1)(3\ell_2 + 1)/(3\ell_1 + 3\ell_2 + 2) = O(n \cdot \max\{\ell_1, \ell_2\})$ rounds. After that, each agent repeats a forward and backward move at different times before moving to its next node. In addition, each agent stays for ℓ_i rounds before and after it repeats the forward and backward moves. This implies that an agent cannot miss the other agent, and thus, they achieve a rendezvous.

In both cases, two agents achieve a rendezvous in $O(n\ell_1\ell_2)$ rounds. Therefore, the theorem holds. \square

Remark 4. Note that Algorithm **SSring** uses a bounded memory. The memory size of a_i is $O(|\ell_i|)$ because the value of h is at most $3\ell_i + 1$.

6 Extension to Gathering of More Than Two Agents

In this section, we discuss extension to gathering of more than two agents. We assume that the number of agents is m , and a set of agents is denoted by $A = \{a_1, a_2, \dots, a_m\}$. We also assume that the agents have different labels and the label of a_i is denoted by ℓ_i . The underlying model is the same as one described in Sect. 2. In addition, the agents can observe states of other agents when they stay at the same node at the same time.

As described in [14], it is easy to extend a rendezvous algorithm for two agents to a gathering algorithm for more than two agents. That is, for given rendezvous algorithm Alg , we can construct a gathering algorithm as follows.

Let $Alg(\ell)$ be the procedure that the agent with label ℓ executes in Alg . Each agent a_i executes $Alg(\ell)$ until it meets another agent. After some agents meet, they follow the agent with the smallest label among them. That is, when ℓ_s is the smallest label among them, a_s with label ℓ_s continues $Alg(\ell_s)$ as if it does not meet any agent, and all other agents stick to a_s (i.e., they move to the node which a_s moves to).

By using the above technique, we can transform a self-stabilizing rendezvous algorithm to a self-stabilizing gathering algorithm. That is, eventually all agents can meet and move together. However, its termination critically depends on the knowledge of the number of agents. If agents know the number of agents m , they can terminate the algorithm when m agents stay at the same node. On the other hand, if agents do not know the number of agents, we have the following impossibility similar to the case of the gossip problem in [15].

Theorem 4. *When agents do not know the number of agents, there exists no self-stabilizing gathering algorithm such that all agents can terminate at the same node.*

Proof. We prove it by contradiction. Assume that such a self-stabilizing gathering algorithm exists. Let L_1 and L_2 be disjoint sets of labels. When $|L_1|$ (resp., $|L_2|$) agents with labels in L_1 (resp., L_2) exist, all agents meet at a single node, denoted by v_1 (resp., v_2), and terminate there in terminal states. Next, we consider a graph that includes two different nodes v'_1 and v'_2 such that $\deg_{v_1} = \deg_{v'_1}$ and $\deg_{v_2} = \deg_{v'_2}$ hold. We assume that $|L_1| + |L_2|$ agents with labels in $L_1 \cup L_2$ execute the algorithm from the initial configuration such that $|L_1|$ agents with labels in L_1 stay at v'_1 in terminal states and $|L_2|$ agents with labels in L_2 stay at v'_2 in terminal states. Since all agents are in terminal states and never move, they cannot achieve gathering. This is a contradiction. \square

Note that the proof of Theorem 4 does not depend on the weakness of the underlying model. That is, even if agents know the graph size, use randomization, and can leave some information on nodes, no self-stabilizing algorithm achieves gathering and termination.

7 Conclusions

In this paper, we have studied self-stabilizing deterministic rendezvous algorithms for graphs with no whiteboard. We first showed the existence of a self-stabilizing rendezvous algorithm for arbitrary graphs. However, the time complexity of this algorithm is not bounded by any function of the graph size and labels. This raised the question whether there exist polynomial time self-stabilizing rendezvous algorithms. We gave partial answers to the problem by providing a self-stabilizing algorithms for trees and rings. For trees, we gave a self-stabilizing rendezvous algorithm with a time complexity of a polynomial of the graph size and the length of the smaller label. For rings, we gave a self-stabilizing rendezvous algorithm with a time complexity of a polynomial of the graph size and labels.

This paper leaves many open problems:

1. Does there exist a polynomial time self-stabilizing rendezvous algorithm for arbitrary graphs with no whiteboard? Since each agent should explore a graph to achieve a rendezvous, it should realize exploration in a polynomial time from an arbitrary state. For this reason, a *strongly universal exploration sequence (SUXS)* [19] may be a useful tool to realize self-stabilizing rendezvous algorithms. The SUXS guarantees that, for some polynomial $p(n)$, any continuous subsequence of length $p(n)$ realizes exploration of any graph of size n . That is, even when an agent starts moving from the middle of the SUXS, it can explore any graph of size n in a polynomial number of moves.
2. Does there exist a self-stabilizing rendezvous algorithm for rings such that the time complexity is polynomial of the graph size and the length of labels?

3. If the previous problems have no solutions, how many bits of whiteboards are required to realize a polynomial time self-stabilizing rendezvous algorithm? It is shown in [15] that $O(|\ell_{max}| + \log n)$ bits are sufficient, where ℓ_{max} is the biggest label of agents. Is it possible to reduce the number of bits?

References

1. Blin, L., Potop-Butucaru, M.G., Tixeuil, S.: On the self-stabilization of mobile robots in graphs. In: Proceedings of 15th International Conference on Principles of Distributed systems, pp. 301–314 (2007)
2. Bouchard, S., Dieudonné, Y., Ducourthial, B.: Byzantine gathering in networks. *Distrib. Comput.* **29**(6), 435–457 (2016)
3. Chalopin, J., Dieudonné, Y., Labourel, A., Pelc, A.: Rendezvous in networks in spite of delay faults. *Distrib. Comput.* **29**, 187–205 (2016)
4. Czyzowicz, J., Kosowski, A., Pelc, A.: How to meet when you forget: log-space rendezvous in arbitrary graphs. *Distrib. Comput.* **25**(2), 165–178 (2012)
5. Czyzowicz, J., Kosowski, A., Pelc, A.: Time versus space trade-offs for rendezvous in trees. *Distrib. Comput.* **27**(2), 95–109 (2014)
6. D’Angelo, G., Navarra, A., Nisse, N.: A unified approach for gathering and exclusive searching on rings under weak assumptions. *Distrib. Comput.* **30**(1), 17–48 (2017)
7. D’Angelo, G., Stefano, G.D., Navarra, A.: Gathering on rings under the look-compute-move model. *Distrib. Comput.* **27**(4), 255–285 (2014)
8. Dessmark, A., Fraigniaud, P., Kowalski, D.R., Pelc, A.: Deterministic rendezvous in graphs. *Algorithmica* **46**, 69–96 (2006)
9. Dieudonné, Y., Pelc, A., Peleg, D.: Gathering despite mischief. *ACM Trans. Algorithms* **11**(1), 1:1–1:28 (2014)
10. Dieudonné, Y., Petit, F.: Self-stabilizing gathering with strong multiplicity detection. *Theoret. Comput. Sci.* **428**, 47–57 (2012)
11. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. *Commun. ACM* **17**(11), 643–644 (1974)
12. Fraigniaud, P., Pelc, A.: Delays induce an exponential memory gap for rendezvous in trees. *ACM Trans. Algorithms* **9**(2), 17:1–17:24 (2013)
13. Klasing, R., Markou, E., Pelc, A.: Gathering asynchronous oblivious mobile robots in a ring. *Theoret. Comput. Sci.* **390**, 27–39 (2008)
14. Kowalski, D.R., Malinowski, A.: How to meet in anonymous network. *Theoret. Comput. Sci.* **399**, 141–156 (2008)
15. Masuzawa, T., Tixeuil, S.: Quiescence of self-stabilizing gossiping among mobile agents in graphs. *Theoret. Comput. Sci.* **411**(14–15), 1567–1582 (2010)
16. Pelc, A.: Deterministic rendezvous in networks: a comprehensive survey. *Networks* **59**, 331–347 (2012)
17. Pelc, A.: Deterministic gathering with crash faults. CoRR abs/1704.08880 (2017). <http://arxiv.org/abs/1704.08880>
18. Suzuki, I., Yamashita, M.: Distributed anonymous mobile robots: formation of geometric patterns. *SIAM J. Comput.* **28**(4), 1347–1363 (1999)
19. Ta-Shma, A., Zwick, U.: Deterministic rendezvous, treasure hunts, and strongly universal exploration sequences. *ACM Trans. Algorithms* **10**(3), 12:1–12:15 (2014)
20. Tsuchida, M., Ooshita, F., Inoue, M.: Byzantine gathering in networks with authenticated whiteboards. In: Proceedings of 11th International Conference and Workshops on Algorithms and Computation, pp. 106–118 (2017)

Stabilization, Safety, and Security of Distributed
Systems

19th International Symposium, SSS 2017, Boston, MA,
USA, November 5–8, 2017, Proceedings

Spirakis, P.; Tsigas, P. (Eds.)

2017, XIII, 496 p. 82 illus., Softcover

ISBN: 978-3-319-69083-4