A Study on
Adaptive Algorithms of Mobile Agents on
Dynamic Environments

Submitted to

Graduate School of Information Science and Technology

Osaka University

January 2021

Tsuyoshi GOTOH

List of Related Publications

Journal Papers

1. Tsuyoshi Gotoh, Yuichi Sudo, Fukuhito Ooshita, and Toshimitsu Masuzawa, “Dynamic ring exploration with (H, S) view," *Algorithms*, MDPI, Vol. 13, No. 6, p.141, 2020.

2. Tsuyoshi Gotoh, Yuichi Sudo, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa, “Exploration of dynamic tori by multiple agents," *Theoretical Computer Science*, Elsevier, Vol. 850, p.202–220, 2021.

Conference Papers

3. Tsuyoshi Gotoh, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa, “How to simulate message passing algorithms by mobile agents with faults," *Proceedings of the 19th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, LNCS 10616, p.234–249, Boston USA, Nov. 2017.

4. Tsuyoshi Gotoh, Yuichi Sudo, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa, “Group exploration of dynamic tori," *Proceedings of the 38th IEEE International Conference on Distributed Computing Systems*, p.775–785, Vienna Austria, July 2017.

5. Tsuyoshi Gotoh, Yuichi Sudo, Fukuhito Ooshita, and Toshimitsu Masuzawa, “Exploration of dynamic ring networks by a single agent with the H-hops and S-time step view," *Proceedings of the 21st International Symposium on Stabilization, Safety, and Security of Distributed Systems*, LNCS 11914, p.165–177, Pisa Italy, Oct. 2019.

6. Tsuyoshi Gotoh, Paola Flocchini, Toshimitsu Masuzawa, and Nicola Santoro, “Tight bounds on distributed exploration of temporal graphs," *Proceedings of the 23rd International Conference on Principles of Distributed Systems*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, p.22:1–22:16 Neuchatel Switzerland, Dec. 2019.

Technical Reports

7. Tsuyoshi Gotoh, Yuichi Sudo, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa, “Exploration of dynamic tori by mobile agents," *Technical Report of IEICE*, COMP2017-25, p.37–44, Oct. 2017.

Abstract

A distributed computing system consists of a collection of individual autonomous computers that are connected through a network. The networked computers communicate with each other, cooperate toward common tasks or solution of a shared problem, and act autonomously and spontaneously. Due to its scalability and flexibility, the distributed computing system has attracted considerable attention, and is widely prevailing, for example, the Internet, wireless sensor networks, cloud computing, and inter-vehicle networks. As a system model of distributed computing systems, *message-passing model*, in which the computers communicate by sending and receiving bounded sequences of bits, is traditionally applied in many systems. However, the design of distributed systems based on the message passing model is growing increasingly complex as the network becomes larger, more dynamic and diverse.

As a programming paradigm, an agent can be considered as encapsulation of data and actions and allows a new philosophy of protocol and communication software design. This makes algorithm design easier in mobile agent systems than in message-passing systems where nodes communicate with each other by sending and receiving messages. As a computational universe, an agent opens a variety of new challenging problems, and many researchers continue to study the principles and algorithms of the mobile agent based distributed systems. So far many agent-based algorithms have been proposed for several tasks, such as leader election, naming, locating agents, rendezvous, stabilization, termination detection, exploration, topology recognition, black-hole search, network decontamination, and intruder capture.

While most of the above works assume that a network is static, recent large-scale distributed systems can no longer make such an assumption. For example, as systems become larger, they are subject to faults or in an inter-vehicle network, the network topology changes with time due

to the flexibility of the system caused by movement of its nodes (or its vehicle). For this reason, it gets important to design algorithms adapting by themselves to dynamic changes, e.g., the changes of the network topology and the faults which remove agents from the network, called *crash faults* (of agents). The networks whose topology changes with time are called *dynamic networks*.

In this dissertation, we consider algorithms adapting to dynamic networks or crash faults.

For dynamic networks, we consider *exploration* which requires that all the nodes should be visited by at least one agent. The exploration is important for solving foundational tasks because it can be used for network maintenance, data collection, or data distribution. There are two kinds of exploration: one is perpetual exploration (considered in Chapter 3) which requires every node should be visited infinite times; the other is exploration with termination (considered in Chapters 4 and 5) which requires all the agents should stop their actions in finite time after every node is visited at least once.

For crash faults, as an approach to realize agent-based algorithms for many tasks, we focus on, in Chapter 6, simulation of *message-passing algorithms* in mobile agent systems. Such simulation is important in design of mobile agent systems since many message-passing algorithms proposed so far can be utilized in mobile agent systems by the simulation.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Background

A distributed computing system consists of a collection of individual autonomous computers that are connected through a network. The networked computers communicate with each other, cooperate toward common tasks or solution of a shared problem, and act autonomously and spontaneously. Due to its scalability and flexibility, the distributed computing system has attracted considerable attention, and is widely prevailing, for example, the Internet, wireless sensor networks, cloud computing, and inter-vehicle networks. As a system model of distributed computing systems, *message-passing model*, in which the computers communicate by sending and receiving bounded sequences of bits, is traditionally applied in many systems. However, the design of distributed systems based on the message passing model is growing increasingly complex as the network becomes larger, more dynamic and diverse.

As a programming paradigm, a *mobile agent* (or simply *agent*) can be considered as encapsulation of data and actions and allows a new philosophy of protocol and communication software design. This makes algorithm design easier in mobile agent systems than in message-passing systems where nodes communicate with each other by sending and receiving messages. As a computational universe, an agent opens a variety of new challenging problems, and many researchers continue to study the principles and algorithms of the mobile agent based distributed systems. So far many agent-based algorithms have been proposed for several tasks, such as leader

election, naming, locating agents, rendezvous, stabilization, termination detection, exploration, topology recognition, black-hole search, network decontamination, and intruder capture [1].

While most of the above works assume that a network is static, recent large-scale distributed systems can no longer make such an assumption. For example, as systems become larger, they are subject to faults or in an inter-vehicle network, the network topology changes with time due to the flexibility of the system caused by movement of its nodes (or its vehicle). For this reason, it gets important to design algorithms adapting by themselves to dynamic changes, e.g., the changes of the network topology and the faults which remove an agent from the network, called *crash faults* (of agents). The networks whose topology changes with time are called *dynamic networks*.

1.2 Overview of This Dissertation

In this dissertation, we consider algorithms adapting to dynamic networks or crash faults.

For dynamic networks, we consider *exploration* which requires that all the nodes should be visited by at least one agent. The exploration is important for solving foundational tasks because it can be used for network maintenance, data collection, or data distribution. There are two kinds of exploration: one is perpetual exploration (considered in Chapter 3) which requires every node to be visited infinitely many times; one is exploration with termination (considered in Chapters 4 and 5) which requires all the agents to stop their actions in finitely many times after every node is visited at least once.

For crash faults, as an approach to realize agent-based algorithms for many tasks, we focus on, in Chapter 6, simulation of *message-passing algorithms* in mobile agent systems. Such a simulation is important in design of mobile agent systems since many message-passing algorithms proposed so far (e.g. those in [2, 3]) can be utilized in mobile agent systems by the simulation.

1.2.1 Exploration of dynamic networks with arbitrary footprints

In Chapter 3, we consider perpetual exploration by mobile agents in the most general dynamic networks, that is, whose underlying topology is arbitrary and whose dynamic changes are least restrictive. Actually, the dynamic changes only guarantee, for any two nodes, by choosing its path cleverly, an agent can always reach one of the nodes from the other nodes. The dynamic networks

satisfying the condition is said *temporally connected*.

We focus on solvability of the exploration of such dynamic networks, and specifically on the number of agents that are necessary and sufficient for exploration under different assumptions on synchrony, the FSYNC and SSYNC.

Clearly, if the graph is not temporally connected, exploration is trivially impossible to achieve. We thus start our investigation with the class $\mathcal{H}$ of temporally connected dynamic networks. We first prove that the number of agents sufficient to perform exploration is related to the number of its transient edges which eventually cease to appear, a parameter $\eta(\mathcal{G})$ we call *evanescence* of the network. More precisely, we prove that any $\mathcal{G} \in \mathcal{H}$ can be explored by a team of $k \geq 2\eta(\mathcal{G}) + 1$ identical agents. We show that this bound is tight by proving that there are $\mathcal{G} \in \mathcal{H}$ that cannot be explored by $2\eta(\mathcal{G})$ agents.

The impossibility holds under very strong conditions: FSYNC scheduler, agents and nodes with distinct IDs, knowledge on n and k, unbounded-size whiteboards. On the other hand, the proposed exploration algorithm works under very weak conditions: SSYNC scheduler (under the weakest transport condition), anonymous agents, no knowledge of topological parameters, and $O(\log \delta_v)$ bits whiteboard at node v (where δ_v denotes the degree of v in the footprint of the dynamic network). Our exploration algorithms are based on the classical rotor router technique, which was introduced as a deterministic alternative to random walk and was studied in a variety of contexts including static graph exploration (e.g., [4, 5, 6, 7, 8]).

We then turn our attention to the stronger assumption on the dynamics of the network, *1-interval connectivity*: the network is always connected. Let $\mathcal{W}(\ell) \subset \mathcal{H}$ be the class of these always-connected dynamic networks where the number of missing edges at each time is at most ℓ.

We start by considering the case of *anonymous* agents. We first prove a tight bound of $2\ell + 1$ agents under the SSYNC scheduler. The proposed algorithm performs exploration even if the network size and the number of agents are not known, and with the weakest transport condition; the impossibility with 2ℓ agents holds even if the network size and the number of agents are known and with whiteboards of unbounded-size.

We then clarify a difference between FSYNC and SSYNC when the network size and the number of agents are known. In fact, in this case, we show a tight bound of 2ℓ for FSYNC, which is smaller

by one than that for SSYNC. Moreover, we show that with $2\ell+1$ agents *exploration with termination* is possible in FSYNC.

Finally, we consider the case of non-anonymous agents, assuming the presence of a *leader* agent. While the lower bound on the number of agents needed for the exploration of $\mathcal{H}$ holds regardless of the existence of a leader, we prove that non-anonymity has an impact on the exploration of $\mathcal{W}(\ell)$. In fact, by exploiting the presence of a leader, the tight bound on the number of agents decreases by one both in FSYNC and in SSYNC. Moreover, we show that, with a leader, 2ℓ agents can *explore with termination* in FSYNC.

Our results indicate, among other things, that the much weaker condition of semi-synchrony (with respect to full-synchrony) is enough to undermine the advantages provided by the much stronger connectivity assumption of $\mathcal{W}$ (with respect to $\mathcal{H}$). Indeed, when considering the class $\mathcal{H}(\ell)$ of temporally connected graphs with at most ℓ transient edges and the class $\mathcal{W}(\ell) \subset \mathcal{H}(\ell)$ of ℓ-bounded 1-interval connected network, we have that the bound on the number of agents for $\mathcal{H}(\ell)$ is the same as the one for $\mathcal{W}(\ell)$ for SSYNC, while the two differs by one in the case of FSYNC.

1.2.2 Exploration of dynamic tori

In Chapter 4, we consider the influence of a very weak view, focusing on exploration with termination on $\nu \times \mu$ dynamic tori under the settings of full synchrony and globally-consistent node labelings. An agent with the view can see which incident links of its current node is present at the current time. Such a view is called the *link presence detection*. Note that in Chapter 3, exploration by agents without any view (agents cannot see any information about the presence of links) is considered. A torus is a natural extension of rings and grids which have many applications and it is worth considering. We consider an $\nu \times \mu$ dynamic torus with a constraint, that is, each graph appearing in a TVG is a torus consisting of ν row rings and μ column rings where each ring is *1-interval connected*. The notion of 1-interval connected is that every static graph in TVG is connected. Note that the dynamic torus is 1-interval connected and is a strict subset of 1-interval connected tori.

We analyze the necessary and sufficient number of agents to explore a dynamic torus and present time-optimal exploration algorithms in a variety of settings. Especially, we consider

exploration with and without the *link presence detection*: an agent can detect which incident links of the current node are present or not before determining its next move.

Without the link presence detection, we prove that $\nu + 1$ agents are necessary and sufficient to explore the $\nu \times \mu$ dynamic torus ($3 \leq \nu \leq \mu$). We propose two algorithms for this setting. One is an algorithm by which $\nu + 1$ agents explore the $\nu \times \mu$ dynamic torus in $O(\nu\mu(\mu - \nu + 1))$ rounds. This algorithm is optimal with respect to the number of agents. Moreover, the algorithm is asymptotically optimal with respect to the time complexity when $\mu - \nu = O(1)$. The other one is an algorithm by which $\nu + 2$ agents explore the $\nu \times \mu$ dynamic torus in $O(\nu\mu)$ rounds. This algorithm is asymptotically optimal with respect to the time complexity. We can see a trade-off between the number of agents and time complexities: $\nu + 2$ agents can explore the $\nu \times \mu$ dynamic torus faster than $\nu + 1$ agents.

With the link presence detection, we prove that $\lceil \nu/2 \rceil + 1$ agents are necessary and sufficient except for $\nu = 4$ and $\lceil \nu/2 \rceil + 2$ agents are necessary and sufficient for $\nu = 4$ to explore the $\nu \times \mu$ dynamic torus. We propose two algorithms for this setting. One is an algorithm by which $\lceil \nu/2 \rceil + 1$ agents explore the $\nu \times \mu$ dynamic torus in $O(\nu\mu(\mu - \nu + 1))$ rounds. This algorithm is optimal with respect to the number of agents. Moreover, the algorithm is asymptotically optimal with respect to the time complexity when $\mu - \nu = O(1)$. The other one is an algorithm by which $\lceil \nu/2 \rceil + 2$ agents explore the $\nu \times \mu$ dynamic torus in $O(\nu\mu)$ rounds. This algorithm is asymptotically optimal with respect to the time complexity. For this case, we can see a similar trade-off between the number of agents and time complexities: $\lceil \nu/2 \rceil + 2$ agents can explore the $\nu \times \mu$ dynamic torus faster than $\lceil \nu/2 \rceil + 1$ agents.

1.2.3 Exploration of dynamic rings with (H, S) view

In Chapter 5, we further investigate the influence of a partial view.

Most of the works about the exploration of dynamic networks consider two extreme cases: an agent has the a priori complete knowledge about changes of all the links for all the future time steps [9, 10, 11, 12]; or an agent can only see whether the links adjacent to its current node are present or not at the moment [13, 14, 15, 16, 17]. The former one models the situation where the network changes are completely predictable as the public transportation networks in which the network changes are introduced by totally scheduled movements of the nodes. The latter one

models the situation where the network changes are caused by unscheduled events, for example, faults or unscheduled movements of the nodes.

Although the above two models are plausible and also theoretically important, the intermediate model, i.e., an agent with partial information or, in other words, capability to know link changes within some distance in the near future should be considered due to the following reasons: even in the totally scheduled situation (if exists), computing all the future changes often costs computation time and it is desirable to compute only the necessary information to solve a problem to save computing time or memories; the ability of an agent to monitor whether there are faults or environmental changes roughly depends on the quality (or costs) of its sensor and it can save some costs to compute only the necessary information for a problem. Moreover, such a model is so interesting from a theoretical viewpoint: how the amount of information available for an agent influences the solvability or the time complexity of problems.

In Chapter 5, we consider the exploration of dynamic networks by a single agent with partial information about network changes. As a first step in this research direction, we focus on 1-interval connected rings as dynamic networks. To formalize the concept of partial information and analyze its influences, in this chapter, we first propose the (H, S) *view* such that the agent with the view can see the *link scheduling* (when and which links disappear or appear) of the links within H hops from its location for S time steps from the current time. Then, we consider how the value of H or S influences the possibility or the time complexity of the exploration by a single agent of 1-*interval connected rings* in which at most one link is missing at each time step. While the 1-interval connected rings are probably too restrictive from a practical point of view, they are adequate targets to investigate in the novel direction as investigated in many works (e.g., in the field of mobile agents on dynamic networks, [11, 15, 18, 19, 20, 21] consider 1-interval connected rings).

For the proposed model, we show that $H + S \geq n$ and $S \geq \lceil n/2 \rceil$ (n is the size of networks) are the necessary and sufficient conditions to explore 1-interval connected rings by a single agent. We also show that in the case where the above conditions holds, the exploration can be achieved within $O(n^2)$ time if $2H' - 1 > S$ or otherwise $O(n^2/H + nH)$ time where $H' = \min(H, \lfloor n/2 \rfloor)$. Moreover, we show that when $S \geq n - 1$, the exploration time can be reduced to $O(n^2/H + n \log H)$. This leads to $O(n \log n)$ time when $H = \Theta(n/\log n)$. Finally, we show a lower bound of the exploration

time, $\Omega(n^2/H)$, for any S. This implies that we have tight bound $\Theta(n^2/H)$ when $H + S \geq n$, $\max(\lceil n/2 \rceil, 2H' - 1) \leq S$, and H is $O(n^{0.5})$ and when $S \geq n - 1$ and $H = O(n/\log n)$.

1.2.4 Fault-tolerant simulation of message-passing algorithms by mobile agents

In Chapter 6, we turn our attention to simulation algorithms in the network where crash faults happen. We propose two fault-tolerant simulating algorithms by k asynchronous agents with distinct IDs. The algorithms have fewer numbers of agent moves than the previous work [22]. We classify message-passing algorithms into two types depending on whether they eventually terminate (with a finite number of messages) or they never terminate (with an infinite number of messages). The first class contains algorithms for spanning tree construction, coloring and so on, and the second class contains mutual exclusion, token circulation and so on. Our algorithms assume at most f agents crash for a given $f \leq k - 1$ (in the analysis, when f is not given, f is replaced by $k - 1$).

For the message-passing algorithms with a finite number of messages, we propose a simulating algorithm with $O((m + M)f)$ total agent moves and thus $O(f)$ agent moves per message when $m = O(M)$ where m is the number of links and M is the number of messages of the simulated algorithm. Note that because crashed agents cannot be distinguished from those moving very slowly in asynchronous systems (where the time required to move along a link is unbounded and unpredictable), every message should be delivered by $f + 1$ agents and every link should be passed by $f + 1$ agents to tolerate f faulty agents in the worst case. This means our algorithm is asymptotically optimal concerning of the number of agent moves.

The improvement in the number of agent moves from [22] is achieved by adopting a different strategy to determine the order in delivering messages. Intuitively, we adopt the depth-first simulation while the previous one adopts the breadth-first one. More precisely, the previous algorithm simulates the synchronous execution of a message-passing algorithm. To realize a synchronous round, each agent traverses the network to find messages to transfer in the round, which requires $O(n)$ redundant moves per message in the worst case. To avoid such redundant moves, our algorithm traces a message to find another message to transfer. That is, the algorithm allows each agent to deliver messages in the depth-first fashion; when an agent visits a node with carrying a message (to deliver it to the node) and finds another message to transfer from the

node, it takes the message and transfers it to the destination node. Note that these two simulating algorithms simulate different executions of the message-passing algorithm, each of which is a possible execution.

For message-passing algorithms with an infinite number of messages, the above depth-first strategy does not work: agents may trace an infinitely long message chain and messages not on the chain (if exist) are never delivered. Thus, we propose another simulating algorithm that delivers messages based on the above depth-first strategy but with a limited number of message deliveries. By repeating such depth-first deliveries with a limited number of message deliveries, the algorithm can make simulation with $O(f)$ agent moves per message. As for this algorithm, the number of agent moves per message is asymptotically optimal.

1.3 Related Works

1.3.1 Dynamic networks

In the recent (and now pervasive) generation of *highly dynamic networks*, the topological changes are not sporadic or anomalous; rather they are extensive, continuous, inherent in the nature of the network. These networks, variously called delay-tolerant, disruptive-tolerant, challenged, epidemic, opportunistic, have been long and extensively investigated by the engineering community and, more recently, by distributed computing researchers. Various models have been proposed to describe some of their aspects, under a variety of names. A unifying model that describes these networks in a simple and natural way is the one of *time-varying graph* (TVG), formally defined in [23], where main classes of systems studied in the literature and their computational relationship were identified.

When time is assumed to be *discrete* (i.e., the system is *synchronous*), the dynamics of the network can be equivalently described as a sequence of static graphs, $\langle G_0, G_1, G_2, ...\rangle$, called *evolving graph* or *temporal graph*, where G_i describes the topology of the network at time $t = i$; this representation was originally suggested in [24] and first formalized in [25]. Each G_i is called a *snapshot*, while the aggregate graph $G = \cup_i\{G_i\}$ is called the *footprint* of the temporal graph.

Computations in temporal graphs have been investigated in distributed computing quite extensively. If the dynamics of the changes is arbitrary and unrestricted, clearly any non-trivial

computation is unfeasible and any non-trivial problem is unsolvable. Hence, all the studies are carried out under some assumptions restricting the arbitrariness of the dynamics.

The minimal (i.e., less restrictive) assumption is *temporal connectivity*: starting at any time, from any node there exists a temporal path, called journey, to any other node (e.g., [13, 14, 26, 27]). Let us stress that, if temporal connectivity does not hold, any non-trivial task and computation is impossible.

Stronger assumptions include *periodicity* : the network is temporally connected and there is (a known) $p > 1$ such that, for all $i \geq 0$, $G_i = G_{i+p}$ (e.g., [11, 28, 29, 30, 31]); *1-interval connectivity* : every G_i is connected (e.g., [32, 33, 34]); and *T-interval connectivity* : for every i, the graphs $G_i, G_{i+1}, ..., G_{i+(T-1)}$ contain the same spanning-tree (e.g., [11, 32]). A classification of the most common assumptions was done in [23].

1.3.2 Graph Exploration in Static Networks

The *exploration* problem, first introduced by Shannon [35], is a fundamental problem in theoretical computer science, in particular in the field of distributed computing by mobile entities. It requires each node of the graph to be visited by one or more mobile computational entities, called *agents*, a finite number of times (exploration *with termination*) or infinitely often (*perpetual* exploration). In addition to its theoretical importance, exploration is relevant from a practical viewpoint in networked systems supporting mobile entities (e.g., software agents, vehicles, or robots): by visiting all nodes, agents can check whether there are some nodes with problems in the network, propagate some data across the network, or collect (or search) specific information from the whole network.

This problem has been extensively studied over a variety of assumptions and settings depending on whether the nodes have distinct labelings or are anonymous, on whether the agents have Ids or are anonymous, the type of mechanism available to the agents for interaction or communication (i.e., whiteboards, tokens, face-to-face, vision), on the degree of synchronization (i.e., asynchronous, semi-synchronous, fully-synchronous), on the level of knowledge the agents have about the graph, on their memory, etc. (e.g., see [36, 37, 38, 39, 40, 41, 42, 43, 44], and [45] for a recent survey). In spite of all the differences, the existing literature has until very recently made a common assumption: the graph is *static*, i.e., the link structure does not change during

the exploration. Static graphs are a common representation of traditional networks, where the changes are typically due to failures; such graphs however fail to describe the new generation of infrastructure-less highly dynamic networks.

1.3.3 Graph Exploration in Dynamic Networks

Many results on exploration of Dynamic Networks are *centralized* (or off-line); that is, they assume that the exploring agents have complete a priori knowledge of the topological changes and the times of their occurrence. They include: the study of the complexity of computing a foremost exploration schedule under the 1-interval-connectivity assumption [46], generalized and extended in [10] and then in [12]; the computation of an exploration schedule for *rings* under the stronger T-interval-connectivity assumption [11]; the computation of an exploration schedule for *cactuses* under the 1-interval-connectivity assumption [9].

Fewer studies use a *decentralized* (i.e., *distributed*) approach. On the probabilistic side, there is an early seminal work on random walks [47] and a recent work [48]. On the deterministic side, exploration has been studied under particular constraints on the network connectivity and on its underlying topology. Exploration with termination by a single agent of periodic temporal networks, including *carrier networks*, has been studied in [11, 28, 29, 30]. Perpetual exploration by three agents on temporally connected *rings* has been studied in [13, 14]. Exploration with termination of 1-interval connected *rings* by two and three agents has been studied in [15], where, in addition to the traditional *fully-synchronous* (FSYNC) scheduler (where all the agents are active at every round), they considered also the *semi-synchronous* (SSYNC) scheduler where only a subset of the agents is active at each round.

1.3.4 Agents in dynamic networks

The problems other than exploration are also considered on dynamic networks, summarized in [49]; *gathering* on 1-interval connected rings [18, 21] which requires all the agents to gather at one node or at adjacent two nodes; *dispersion* [50] which stipulates that every node must be occupied by exactly one agent where the number of agents is the same as that of nodes on *permuting rings* in which the nodes may be permuted at each time step, i.e., the neighbors of a node may change

at each time step while the topologies are rings or paths at each time step; *patrolling* on 1-interval connected rings [19] which requires the maximum length of the interval between two visits to a node to be minimized; *compacting* on 1-interval connected rings [20] which stipulates that all the agents in a network must be located in a continuous part of the ring and at each node there exists at most one agent.

1.3.5 Simulation of message-passing algorithms by agents

The agent model and the message-passing model have been compared for the first time in [51] from a systems engineering point of view. The fact that any mobile agent algorithm can be simulated in message-passing model has been proved in [52], which immediately implies that all the impossibility results under the message-passing model, also hold for the agent model. Recently, simulation in the other direction was shown in [53]. These results imply that the two models are computationally equivalent. In [54], a simulation algorithm was proposed to simulate a message-passing algorithm for the leader-election. The simulation in a fault-tolerant manner was proposed for the first time in [22]. They propose two algorithms to simulate message-passing algorithms by k asynchronous agents where at most $k-1$ of them crash. One algorithm simulates a message-passing algorithm with $O((m+nM)k)$ total agent moves by agents having distinct IDs and with $O(nk)$ agent moves per message when $m = O(nM)$, where m is the number of links, n is the number of nodes and M is the number of messages created in the simulated execution of the message-passing algorithm. Another algorithm simulates a message-passing algorithm with $O((m+nk)M)$ total agent moves by anonymous agents (or agents without IDs).

1.4 Organization of This Dissertation

This dissertation consists of seven chapters. In Chapter 2, we describe definitions of our system model, agent model, and each problem. In Chapter 3, we propose algorithms to solve the g-partial gathering problem in ring networks. In Chapter 4, we propose algorithms to solve the g-partial gathering problem in tree networks. In Chapter 5, we propose algorithms to solve the uniform deployment problem in ring networks. In Chapter 6, we propose fault-tolerant agent algorithms simulating message-passing algorithms. We conclude this dissertation in Chapter 7.

Chapter 2

Preliminary

In this chapter, we describe a general definition of a network model, an agent model, and the exploration problem.

Network and Agent. A network is modeled as a undirected, connected, and simple graph $G = (V, E)$, where V is a set of nodes and E is a set of links (or edges). We use a link and an edge interchangeably in the following chapters. We denote the number of nodes by $n\,(= |V|)$ and the number of links by $m\,(= |E|)$. A link connecting v and u is denoted as e_{vu} or (v, u). Let $E(v)$ denote the set of links incident to node v, let δ_v (or deg_v) $= |E(v)|$ be the degree of node v, and let $\Delta = \max_v\{\delta_v\}$ be the maximum degree of G. Each link incident to node v is locally labeled by a bijection $\lambda_v : E(v) \to \{0, \ldots, \delta_v - 1\}$; no other assumption is made about the label. Using these labels, an agent in v distinguishes the neighbors of v.

A dynamic network is modeled as a time-varying graph (TVG), $\mathcal{G} = (V, E, \mathbb{T}, \rho)$, where $\mathbb{T}$ is the temporal domain, and $\rho : E \times \mathbb{T} \to \{0, 1\}$, called *presence function*, indicates whether a given link is available at a given time. In this dissertation, for dynamic networks, we always assume discrete time, that is, $\mathbb{T} = \mathbb{Z}^*$. When we consider $\mathcal{G}$, the graph $G = (V, E)$ is also called *underlying graph* (or *footprint*) of $\mathcal{G}$. When or which links are deleted is determined by an adversary. The adversary can see the contents of the memory of nodes and agents, knows the agent's algorithm and deletes links under the given restriction to prevent agents from completing their goal.

A set $A = \{a_0, a_1, \ldots, a_{k-1}\}$ of k *agents* operates in G, initially occupying arbitrary positions. Each agent $a \in A$ is a computational entity endowed with private memory (called notebook), and

capable of moving from a node to a neighboring node (in dynamic networks, provided that the link between the nodes exists at the time). A node v has buffer space for each link to store at most one agent, called a *port* or a *room*. An agent in a node v can write to and read from memory space of v when v has such memory space. Such memory space (writable and readable for agents) in a node is called a *whiteboard*.

In a dynamic network, an agent sometimes use a *view*. The view contains information of presence nearby links in near future, i.e., an agent with a view can see when and which links are present. How long ahead and/or how far an agent can see by the view depends on models and is specified in each chapter when we use it.

An agent iteratively makes the following actions, Look, Compute, Move in this order. The details of each action are as follows:

Look: Agent a_i observes the content of its notebook and, if any, the contents of the whiteboard and the ID of the node where it currently resides which is also called its *current node*. It also checks the node memory, its location and the ports of the node to determine if there are other agents at this node and where (e.g., which ports). If a view is given, it sees when and which links are present using its view.

Compute: On the basis of the information obtained in the Look phase, a_i decides whether to move or not, and it can update the whiteboard of the current node. If it decides to move, it places itself in correspondence of the selected port (if it is not occupied by another agent).

Move: If a_i is at a port, it tries to move; (in a dynamic network) if the corresponding link exists, a_i reaches the other side, otherwise it stays on the port. If a_i does not occupy a port, it does not move.

Look and Compute phases are executed as an atomic action. Atomic actions of agents in the same node are executed with mutual exclusion access to the whiteboard. The order of actions of agents is determined by an adversary.

If all the agents complete exactly one cycle of Look-Compute-Move in every specified time unit, we say that agents are activated *fully synchronously* and call the model Fsync. If a subset of agents completes exactly one cycle of Look-Compute-Move and agents not in the subset do

nothing in every specified time unit, we say that agents are activated *semi-synchronously* and call the model SSYNC. Otherwise, we say that agents are activated *asynchronously* and call the model ASYNC. In FSYNC and SSYNC model, we call a specified time unit a *round*. Note that in FSYNC and SSYNC model, at most one agent moves through each edge in every round from the definition.

Exploration. We say that a node v is *visited at* round t if v has an agent at the beginning of t and that a node v is *visited by* round t if v is visited at round t' for some t' ($0 \leq t' \leq t$). We say that the network is explored by round t if every node is visited by round t.

A *perpetual* exploration algorithm is one where, in every execution, every node is visited at an infinite number of rounds. An exploration *with termination* algorithm is one where, in every execution, all the agents terminate after all nodes have been visited at least once.

Chapter 3

Exploration of Dynamic Graphs with Arbitrary Footprints

3.1 Introduction

In this chapter, we consider the problem of exploring temporal graphs of *arbitrary unknown topology*. We study the feasibility of exploration, under both the fully synchronous (FSYNC) and semi-synchronous (SSYNC) activation schedulers, focusing on the number of agents necessary and sufficient to explore such graphs.

We first consider the minimal (i.e., less restrictive) assumption on the dynamics of the graph under which exploration is still feasible: *temporal connectivity*. Let $\mathcal{H}$ be the class of temporally connected graphs; we show that for any temporal graph $\mathcal{G} \in \mathcal{H}$ the number of anonymous agents sufficient to perform exploration is related to the number of its transient edges, a parameter $\eta(\mathcal{G})$ we call evanescence of the graph. More precisely, any $\mathcal{G} \in \mathcal{H}$ can be explored by $k \geq 2\eta(\mathcal{G}) + 1$ anonymous agents; this bound is tight as we prove there are $\mathcal{G} \in \mathcal{H}$ that cannot be explored by $2\eta(\mathcal{G})$ agents.

We then turn our attention to the well-known stronger assumption on the dynamics of the graph, called *1-interval connectivity*: the graph is connected at any time step. Let $\mathcal{W} \subset \mathcal{H}$ be the class of these always-connected temporal graphs. For this class, we prove the existence of a difference between FSYNC and SSYNC and between anonymous and non-anonymous agents when

there is a bound ℓ on the number of edges missing at each time. In fact, we first show a tight bound of $2\ell + 1$ on the number of anonymous agents necessary and sufficient in SSYNC, and a smaller tight bound of 2ℓ in FSYNC. We then turn to agents with a leader, where we provide a tight bound of 2ℓ on the number of agents necessary and sufficient in SSYNC, and a smaller tight bound of $2\ell - 1$ in FSYNC.

The results are summarized in Table 3.1

Table 3.1: Tight bounds on the number of agents.

		anonymous	with leader
Temporally connected: $\mathcal{H}$	FSYNC, SSYNC	$2\eta + 1$	$2\eta + 1$
ℓ-bounded 1-interval : $\mathcal{W}(\ell)$	SSYNC	$2\ell + 1$	2ℓ
	FSYNC	2ℓ	$2\ell - 1$

3.2 Preliminary

3.2.1 Network

In this chapter, we consider the perpetual exploration of dynamic networks with arbitrary footprints, i.e., $G = (V, E)$ is arbitrary. The nodes in V are anonymous (i.e., they have no IDs).

A *journey* is a temporal walk in $\mathcal{G}$ and it is defined as a sequence of couples $\mathcal{J} = \{(e_1, t_1), (e_2, t_2) \ldots, (e_k, t_k)\}$, such that $\{e_1, e_2, ..., e_k\}$ is a walk in G and $\forall i, 1 \leq i < k$, $\rho(e_i, t_i) = 1$ and $t_{i+1} > t_i$. Let $J(u, v, t)$ denote the set of journeys from u to v starting at time $t' \geq t$.

In this case, the TVG $\mathcal{G}$ of discrete time is usually called *temporal graph* (or *evolving graph*), and can be viewed as a sequence of static graphs: $\mathcal{S}_{\mathcal{G}} = G_0, G_1, \ldots, G_t, \ldots$, where $G_t = (V, E_t)$ is the graph induced by the edges present at time t (called *snapshot* of $\mathcal{G}$ at time t). We denote by $\bar{E}_t = E \setminus E_t$ ($\subseteq E$) the set of edges that do not appear in the snapshot at time t.

An edge $e \in E$ is said to be *recurrent* if $\forall t \in \mathbb{Z}^+, \exists t' > t : \rho(e, t') = 1$; in other words, a recurrent edge appears infinitely often. An edge $e \in E$ that is not recurrent is said to be *transient*; in other words, a transient edge appears only in a finite number of snapshots. Let E^* and E^- denote the set of recurrent and of transient edges, respectively; the number $\sigma(\mathcal{G}) = |E^*|$

of recurrent edges is called the *solidity* of $\mathcal{G}$; while the number $\eta(\mathcal{G}) = |E^-| = |E| - \sigma(\mathcal{G}))$ of transient edges is called the *evanescence* of $\mathcal{G}$.

3.2.2 Connectivity

Temporal graphs can be clasified in terms of the effect that the dynamic topological changes have on their connectivity. In this chapter, we consider the following connectivities.

Definition 3.2.1 (Temporally Connected)**.** *A temporal graph $\mathcal{G}$ is* temporally connected *(or* connected over time*) if $\forall t \in \mathbb{Z}^+$, $\forall u, v \in V$, $J(u, v, t) \neq \emptyset$.*

Note that temporal connectivity is the minimal condition to be able to perform any global task when an adversary determines the initial locations of agents; in particular, any problem requiring every node to be involved (e.g., exploration) is trivially unsolvable if $\mathcal{G}$ is not temporally connected. Let $\mathcal{H}$ denote the class of temporally connected TVGs.

A variety of stronger assumptions have been studied in the literature. In this chapter we are interested also in the well-known class of temporal graphs where connectivity is actually guaranteed at every time, and in particular when the number of missing edges at any given time is bounded.

Definition 3.2.2 (ℓ-Bounded 1-Interval Connected)**.** *A temporal graph $\mathcal{G}$ is* 1-interval connected *(or* always connected*) if $\forall G_i \in \mathcal{S}_{\mathcal{G}}$, G_i is connected. Moreover, $\mathcal{G}$ is* ℓ-bounded 1-interval connected *if it is always connected and $|\bar{E}_t| \leq \ell$.*

Let $\mathcal{W}(\ell) \subset \mathcal{H}$ denote the class of ℓ-bounded 1-interval connected temporal graphs.

3.2.3 Agents

A set A of k agents initially occupies arbitrary positions. When the agents are all undistinguishable, we say that they are *anonymous*; if one of them is different from all the others, we say that they have a *leader* (and are not-anonymous).

When at a node v, an agent has access to the node's ports and rotor-router mechanism. More precisely, in correspondence of each edge $e \in E(v)$, there is in v a *port* p_i where $i = \lambda_v(e)$, used by agents (at most one at a time) intending to leave v through e. Additionally, v provides

a rotor-router mechanism, which indicates one of the ports; this indication, called a pointer, can be read and modified by the agents; access to this mechanism is in fair mutual exclusion. This mechanism is implemented by a whiteboard of $O(\log \delta_v)$ bits in the following.

In SSYNC, the scheduler is an adversary which knows the algorithm of the agents, has infinite computing capacity, and tries to prevent agents from completing their task; however, it must activate every agent infinitely often. An agent which is not activated at round t is said to be *sleeping* at that round. The length of the sleeping time is finite but unbounded.

Under the semi-synchronous scheduler, it is necessary to specify the behavior of the agents that fall asleep on a port when the corresponding edge is missing. We consider the weakest condition, *eventual transport*, according to which the agent sleeping at a port will eventually be activated at a time when the edge corresponding to the port is present; this prevents the adversary from using semi-synchronicity to block an agent forever on a recurrent edge.

In this chapter, every agent has no view, i.e., an agent cannot use the information about when and which links are deleted at all.

3.2.4 Configuration and Execution

A *configuration* C_t is defined by: the contents of the whiteboards, the local memory of the agents, and the locations of the agents at the start of round t.

An *execution* $\mathcal{E}(\mathcal{A}) = C_0C_1\ldots$ of an algorithm $\mathcal{A}$ is an infinite sequence of configurations such that C_0 is an initial configuration (i.e., a configuration at round 0) and C_{t+1} is obtained from C_t by executing one round of algorithm $\mathcal{A}$. This execution is subject to two types of adversarial actions: those by the activation scheduler deciding which agents are activated in that round, and those of the topological scheduler deciding which edges are missing in that round. When no ambiguity arises, we use $\mathcal{E}$ instead of $\mathcal{E}(\mathcal{A})$.

3.2.5 Augmented Configuration and Execution

We use an *augmented configuration* and an *augmented execution* in Sections 3.4.2 and 3.5.2. To define an augmented configuration, we introduce variable $\mathsf{visited}_v$ for all $v \in V$ which is written and read only by an external observer. The initial value of $\mathsf{visited}_v$ is 0. When v is visited, $\mathsf{visited}_v$

is set to 1 by the external observer.

Then, an augmented configuration C_t^{aug} is defined by: configuration C_t and the value of $\mathsf{visited}_v$ of every node v at round t. We say that an augmented configuration is *terminal* when $\mathsf{visited}_v = 1$ for any node v.

An augmented execution $\mathcal{E}^{\mathsf{aug}}(\mathcal{A}) = C_0^{\mathsf{aug}} C_1^{\mathsf{aug}} \dots C_r^{\mathsf{aug}}$ is a sequence of augmented configurations such that C_0^{aug} is an initial augmented configuration; C_{t+1}^{aug} is obtained from C_t^{aug} by executing one round of algorithm $\mathcal{A}$; C_r^{aug} is a unique terminal configuration in $\mathcal{E}^{\mathsf{aug}}$. An augmented execution is also subject to the two types of adversarial actions. Note that the agents may keep executing $\mathcal{A}$ after round r, but augmented configurations after round r are ignored in $\mathcal{E}^{\mathsf{aug}}$. When no ambiguity arises, we use $\mathcal{E}^{\mathsf{aug}}$ instead of $\mathcal{E}^{\mathsf{aug}}(\mathcal{A})$ and an "execution" instead of an "augmented execution".

3.3 Exploration of Temporally Connected TVGs

In this section, we consider the minimal class of explorable temporal graphs: *temporally connected TVGs*, and we show that the feasibility of the exploration of $\mathcal{G}$ is related to its evanescence η, providing a tight bound of $2\eta(\mathcal{G}) + 1$ agents.

3.3.1 Impossibility

Let $\mathcal{H}(\ell) = \{\mathcal{G} \in \mathcal{H} : \eta(\mathcal{G}) \leq \ell\}$ be the class of temporally connected TVGs with evanescence at most ℓ. In this section we show that it is impossible to perform perpetual exploration of all $\mathcal{G} \in \mathcal{H}(\ell)$ with 2ℓ agents. The result is quite strong as it applies also to TVGs that are connected at every time step, with uniquely labeled nodes and agents, under a fully-synchronous scheduler, and in presence of topological knowledge.

Theorem 3.3.1. *There exist temporally connected time-varying graphs $\mathcal{G} \in \mathcal{H}(\ell)$ that cannot be explored by $k = 2\ell$ agents. The result holds even if nodes and/or agents have distinct IDs, the network is always connected, the agents n, m or k, and the scheduler is fully-synchronous.*

Proof. We show the theorem by constructing a graph $\mathcal{G} \in \mathcal{H}(\ell)$ that cannot be explored by 2ℓ agents by any algorithm. The main point of this proof is that an agent can eventually have only

one of these two behaviors when wishing to traverse an edge that is missing: (i) the agent stays permanently on the chosen port, waiting for the appearance of the continuously missing edge; (ii) the agent eventually chooses a different edge. The agents of the former type are called (with respect to the number of changes of a selected edge) *finite agents* and those of the latter are *infinite agents*.

The components for constructing the graph are as follows. For $0 \leq i \leq 2\ell - 1$ $(= k - 1)$, let S_i be a star with center node c_i and 3 leaf nodes $\{b_{(i,0)}, b_{(i,1)}, b_{(i,2)}\}$. We construct the graph using S_i for $0 \leq i \leq 2\ell - 1$ and an additional node u.

Each component is connected as follows. For $0 \leq i \leq 2\ell - 1$ and $j \in \{0, 1\}$, each $b_{(i,j)}$ is connected with u by edge $(b_{(i,j)}, u)$; and for $0 \leq i \leq \ell - 1$, each $b_{(2i,2)}$ connected with $b_{(2i+1,2)}$ by $(b_{(2i,2)}, b_{(2i+1,2)})$. A graph for $\ell = 2$ $(k = 4)$ is depicted in Figure 3.1.

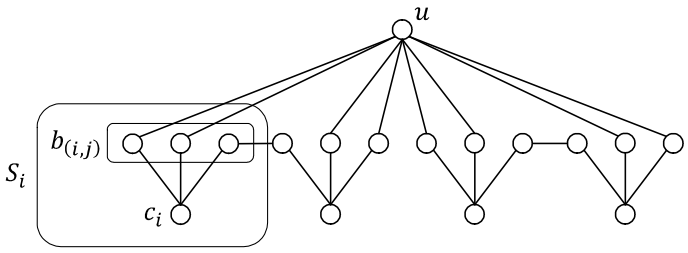

Figure 3.1: Example of a graph for $\ell = 2$ and $k = 2\ell = 4$ that cannot be explored by 2ℓ agents. There are four stars S_i for $0 \leq i \leq 3$ in the figure. Each star S_i has one center node c_i and three leaf nodes $\{b_{(i,0)}, b_{(i,1)}, b_{(i,2)}\}$.

For the constructed graph, we first show that, given any exploration algorithm using 2ℓ agents, the adversary can construct an execution for the algorithm such that in the execution $\mathcal{G}$ cannot be explored while the adversary may violate the restriction of $\mathcal{H}(\ell)$, i.e., $\eta(\mathcal{G})$ may be more than ℓ. Then, we give a way to convert the execution into another execution such that $\eta(\mathcal{G})$ is at most ℓ in the new execution and the agents cannot distinguish these two executions and thus cannot explore $\mathcal{G}$ also in the new execution.

We start by showing that, given any exploration algorithm, say $\mathcal{A}$, using 2ℓ agents, the adversary can construct an execution $\mathcal{E}_1$ of $\mathcal{A}$ in which the agents cannot explore $\mathcal{G}$. The adversary puts agent a_i on c_i for $0 \leq i \leq 2\ell - 1$ in the initial configuration of $\mathcal{E}_1$. During

execution $\mathcal{E}_1$ of $\mathcal{A}$, the adversary deletes the edge leading to u or the other star whenever a_i is on $b_{(i,j)}$. Clearly, this prevents any agent executing $\mathcal{A}$ from visiting u and thus $\mathcal{G}$ is not explored permanently while the adversary violates the restriction for the number of transient edges (it is at most 2ℓ in $\mathcal{E}_1$).

We now show how the adversary converts $\mathcal{E}_1$ into another execution, say $\mathcal{E}_2$, so that the agents cannot distinguish $\mathcal{E}_1$ and $\mathcal{E}_2$ and $\eta(\mathcal{G})$ is at most ℓ in $\mathcal{E}_2$. The adversary first separates the agents into two groups: *finite agents* and *infinite agents* depending on their behavior when faced with a missing edge during $\mathcal{E}_1$. Let f $(0 \leq f \leq k)$ be the number of *finite agents*. In the following, *finite agents* are denoted by $a_0^{\mathsf{fin}}, \ldots, a_{f-1}^{\mathsf{fin}}$. In the initial configuration of $\mathcal{E}_2$, each agent (a_i) is put on the same node (c_i) as in $\mathcal{E}_1$.

Then, the adversary constructs a new assignment of the port labels and the node ID (if any) of nodes so that every agent cannot distinguish $\mathcal{E}_1$ and $\mathcal{E}_2$ as follows. For *infinite agents*, the adversary does nothing. For *finite agents*, let $a_i^{\mathsf{fin}} = a_{i'}$ and $b_{(i',x_i)}$ be the node where a_i^{fin} finally waits for a missing edge permanently in $\mathcal{E}_1$. For $0 \leq i \leq f-1$, the adversary does the following: if $x_i = 2$, the adversary does nothing; and otherwise, the adversary swaps the assignment of the port labels and the node ID of $b_{(i',2)}$ and $b_{(i',x_i)}$ and accordingly permutes the port labeling of $c_{i'}$.

Execution $\mathcal{E}_2$ with the initial configuration, the node ID, and the assignment of port labels is constructed similarly to $\mathcal{E}_1$: the adversary deletes the edge leading to u or the other star when a_i exists on $b_{(i,j)}$. Obviously, every agent cannot distinguish $\mathcal{E}_1$ and $\mathcal{E}_2$: for all the agents, the node IDs and the port labeling observed in $\mathcal{E}_2$ is the same as $\mathcal{E}_1$. Thus, $\mathcal{G}$ cannot be explored since u is not visited by any agent also in $\mathcal{E}_2$.

Since the edges waited permanently by an agent are only $(b_{(2i,2)}, b_{(2i+1,2)})$ for $0 \leq i \leq \ell - 1$, $\eta(\mathcal{G})$ is at most ℓ in $\mathcal{E}_2$. □

3.3.2 Semi Synchronous Exploration by $2\eta(\mathcal{G}) + 1$ Agents

In this section, we show that every temporally connected time-varying graph $\mathcal{G} \in \mathcal{H}$ can be explored by $2\eta(\mathcal{G}) + 1$ anonymous agents that do not know the topology. In fact, we propose an exploration algorithm for $2\eta(\mathcal{G}) + 1$ anonymous agents in an anonymous network, which works under the semi-synchronous scheduler with eventual transport.

The strategy is simple and it is based on the classical *rotor router* mechanism, which was

introduced as a deterministic alternative to random walk and was studied in a variety of contexts, including static graph exploration (e.g., [4, 5, 6, 7, 8]).

In rotor router, each node v has a variable written on its whiteboard, $\mathsf{pointer}_v$, indicating one of its incident ports. When an agent a visits node v, a checks each port in ascending order from the port pointed by $\mathsf{pointer}_v$. If a finds some unoccupied port p, a moves to that port and sets $\mathsf{pointer}_v$ to $p + 1$. If a finishes to check all the ports and they all are occupied, a does nothing.

Algorithm 3.1 Computation at node v

```
1: if not on a port then
2:     i ← 0
3:     p ← pointer_v
4:     while i < δ_v ∧ port p is occupied do
5:         p ← (p + 1) mod δ_v
6:         i ← i + 1
7:     if i < δ_v then
8:         pointer_v ← (p + 1) mod δ_v
9:         move to port p
```

We first show that, in any round, there exists at least one agent succeeding to move within finite time (Lemma 3.3.1). We then show that, $2\ell + 1$ agents achieve perpetual exploration using Algorithm 3.1 (Theorem 3.3.2).

Lemma 3.3.1. *For any round t, if $2\eta(\mathcal{G}) + 1$ agents execute Algorithm 3.1 in a temporally connected temporal graph $\mathcal{G}$, at least one of them eventually moves after t.*

Proof. By contradiction, assume that there exists a round t such that every agent never succeeds to move after t. We consider two cases: (i) there exists a node v containing more than $\delta_v - 1$ agents, and (ii) there does not exist such a node.

In the first case, every agent on v is activated within finite time after t because of the fairness of the scheduler, which means that every port of v is eventually occupied by an agent. Since at least one of the edges incident to v is a recurrent edge, say e, the agent sleeping on the corresponding port of e eventually succeeds to move because of the eventual transport rule. This is a contradiction.

Also in the second case, every agent on v is activated within finite time after round t because of the fairness of the scheduler. Since there is no node containing more agents than its degree, every agent eventually stays on a port. When this happens, at least one of the agents is sleeping at the port of a recurrent edge since the number of agents is $2\eta(\mathcal{G}) + 1$ and there exist at most $2\eta(\mathcal{G})$ ports corresponding to transient edges. This means that, by the eventual transport rule, the agent sleeping at the port of a recurrent edge eventually succeeds to move after t; a contradiction. □

Then, the following theorem holds.

Theorem 3.3.2. *Any $\mathcal{G} \in \mathcal{H}$ can be explored by $2\eta(\mathcal{G}) + 1$ anonymous agents under the semi-synchronous scheduler.*

Proof. Consider Algorithm 3.1. By the definition of transient edges, there exists a time step t_e for any transient edge e such that $\rho(e, t) = 0$ for all $t > t_e$. Let t_E be $\max_{e \in E^-} t_e$, i.e., a time when all the transient edges have ceased to exist and all the edges that appear from this moment are recurrent. Let $x(t)$ be the sum of the number of agent moves from a node to another node over all the agents from the beginning of the execution up to time t.

We now show that, from an arbitrary initial configuration, $2\eta(\mathcal{G}) + 1$ agents following Algorithm 3.1 visit all the nodes infinitely often.

First, note that there exists a node, say v, that is visited infinitely often (for $t \to \infty$) because $x(t)$ goes to infinity (for $t \to \infty$) by Lemma 3.3.1.

We now show that every neighbor of v connected by a recurrent edge is also visited at an infinite number of rounds. We prove it by contradiction. Suppose that a neighbor u of v connected by a recurrent edge is visited at only a finite number of times and let t' be the last round when u is visited at. Since v is visited at an infinite number of rounds and the agents execute Algorithm 3.1 perpetually, some agent a visiting v eventually chooses (v, u) as the edge from which a moves out of v after time t'. Recall that (v, u) is a recurrent edge and the agents are activated by the eventual transport rule. It follows that a eventually visits u after round t'; a contradiction.

Since G_r is temporally connected, we can apply inductively the claim (e.g., the neighbors of a neighbor of v are also visited infinitely often) to all the nodes, proving the theorem. □

From Theorems 3.3.1 and 3.3.2, the following Theorem holds.

Theorem 3.3.3. *Exploration of all temporal graphs in $\mathcal{H}(\ell)$ by k agents is possible iff*

$$k \geq 2\ell + 1$$

Note that, if a graph is temporally connected, then its solidity $\sigma(\mathcal{G}) \geq n-1$; as a consequence, we have:

Theorem 3.3.4. *Every temporally connected temporal graph with n nodes and whose footprint has m edges can be explored by $2(m-n)+3$ agents.*

3.4 Exploration of 1-Interval Connected TVGs by Anonymous Agents

In this Section, we turn our attention to the class $\mathcal{W}(\ell)$ of 1-interval connected temporal graphs where the number of missing edges is bounded in each round by a constant ℓ. In other words, at any time t the TVG is connected, and no more than ℓ edges are missing. We establish tight bounds for the exploration of this class of temporal graphs by *anonymous* agents, in SSYNC and in FSYNC.

3.4.1 Semi-synchronous model

We first consider ℓ-bounded, 1-interval connected TVGs operating under a semi-synchronous scheduler and we show that there exist TVGs that cannot be explored by 2ℓ anonymous agents.

Theorem 3.4.1. *There exist 1-interval connected time-varying graphs $\mathcal{G} \in \mathcal{W}(\ell)$ that cannot be explored by $k = 2\ell$ anonymous agents. The result holds even if the agents know n, m and k and whiteboards are of unbounded size.*

Proof. We use the same graph $\mathcal{G}$ constructed for the proof of Theorem 3.3.1. The construction is omitted in this proof.

We first show that, given any exploration algorithm, say $\mathcal{A}$, using 2ℓ agents, the adversary can construct an execution $\mathcal{E}_1$ of $\mathcal{A}$, possibly violating the eventual transport rule, in which the agents cannot explore $\mathcal{G}$. We then show that it is always possible to convert this execution into another execution $\mathcal{E}_2$ that does not violate the eventual transport rule, and where the agents cannot explore $\mathcal{G}$.

In execution $\mathcal{E}_1$, the adversary puts agent a_i on c_i for $0 \leq i \leq k-1 = 2\ell - 1$ in the initial configuration of $\mathcal{E}_1$. During $\mathcal{E}_1$, exactly one agent is activated at each round: a_i is activated at round t when $t \equiv i \pmod{k}$. When the adversary activates a_i and a_i exists on $b_{(i,j)}$, the adversary deletes the edge leading to u or the other star whereas all the other edges are present. Note that the agents and the nodes are anonymous and thus either they are all *finite* (i.e., every agent permanently waits for appearance of its selected edge if the edge is permanently missing) or they are all *infinite* (i.e., every agent eventually changes its selected edge if the edge remains missing) in $\mathcal{E}_1$.

If the agents are *infinite*, the eventual transport rule is not violated even in $\mathcal{E}_1$ and thus the adversary can prevent the agents from completing the exploration in $\mathcal{E}_1$.

If the agents are *finite*, the adversary converts $\mathcal{E}_1$ into another execution, say $\mathcal{E}_2$, as follows. The adversary first puts a_i $(0 \leq i \leq k-1)$ on c_i in the initial configuration of $\mathcal{E}_2$. Then, the adversary changes the assignment of the port labels and the node ID (if any) of each node in S_i in the same way explained in the proof of Theorem 3.3.1 (also omitted in this proof). In $\mathcal{E}_2$, the adversary activates each agent in the same order as in $\mathcal{E}_1$ and deletes an edge leading to u or the other star whenever a_i is on $b_{(i,j)}$. After some round t from which every agent a_i does not change its selected edge, i.e., $b_{(i,2)}$, and waits at a port of $b_{(i,2)}$ forever for $0 \leq i \leq 2\ell$, the adversary deletes $(b_{(2j,2)}, b_{(2j+1,2)})$ for $0 \leq j \leq l-1$ at every round. Obviously, every agent cannot distinguish $\mathcal{E}_2$ from $\mathcal{E}_1$ and $\mathcal{G}$ cannot be explored since u is not visited by any agent in $\mathcal{E}_2$. It is also clear that the eventual transport rule is not violated in $\mathcal{E}_2$. □

Clearly, $\mathcal{W}(\ell) \subset H(\ell)$, thus any $\mathcal{G} \in \mathcal{W}(\ell)$ can be explored by Algorithm 3.1; that is:

Theorem 3.4.2. *Any $\mathcal{G} \in \mathcal{W}(\ell)$ can be explored by $2\ell + 1$ anonymous agents under the semi-synchronous scheduler with eventual transport.*

From Theorems 3.4.1 and 3.4.2 it follows that:

Theorem 3.4.3. *Under a semi-synchronous scheduler, exploration of all ℓ-bounded 1-interval connected TVG by k anonymous agents is possible iff $k \geq 2\ell + 1$.*

3.4.2 Fully-Synchronous Model

In this section, we show that, if the network size and the number of agents are known, there exists a difference between FSYNC and SSYNC in the exploration of ℓ-bounded 1-interval TVGs. In fact, we show that, $\mathcal{G} \in \mathcal{W}(\ell)$ can be explored if $k \geq 2\ell$, while there exist graphs that cannot be explored with $2\ell - 1$ agents.

Impossibility

We now consider ℓ-bounded, 1-interval connected TVGs operating under a fully-synchronous scheduler and we show that there exist TVGs that cannot be explored by $2\ell - 1$ agents, even if the agents know n, m, and k.

Theorem 3.4.4. *There exist ℓ-bounded 1-interval time-varying graphs $\mathcal{G} \in \mathcal{W}(\ell)$ that cannot be explored by $k = 2\ell - 1$ anonymous agents in FSYNC. The result holds even if the agents know n, m, and k, and whiteboards are of unbounded size.*

Proof. Let $K_{2\ell} = (V_{2\ell}, E_{2\ell})$ be the complete graph with 2ℓ nodes where $V_{2\ell} = \{v_0, v_1, \ldots, v_{2\ell-1}\}$. It is well known that the edges of $K_{2\ell}$ can be colored with $2\ell-1$ colors, that is, $E_{2\ell}$ can be partitioned into $2\ell - 1$ disjoint independent edge sets (or complete matchings): $E^{(0)}_{2\ell}, E^{(1)}_{2\ell}, \ldots, E^{(2\ell-2)}_{2\ell}$. For example, the following separation leads to disjoint independent edge sets: each $E^{(i)}_{2\ell}$ has ℓ edges, $(v_i, v_{2\ell-1})$, (v_{i-1}, v_{i+1}), (v_{i-2}, v_{i+2}), $\ldots$, $(v_{i-\ell+1}, v_{i+\ell-1})$, see Figure 3.2 (for simplicity, mod 2ℓ is omitted).

The execution where $v_{2\ell-1}$ remains unvisited is constructed as follows. For $0 \leq i \leq 2\ell - 1$, the adversary places each agent a_i on v_i and for $0 \leq j \leq 2\ell - 2$ assigns a label j to the port of v_i corresponding to e, if $e \in E^{(j)}_{2\ell}$. Note that, since agents and nodes are anonymous, all the agents make the same action and select the port with the same label to move at each round. Thus, the adversary can prevent any agent from moving by deleting all the edges of $E^{(i)}_{2\ell}$ when the agent selects port i; as a consequence, none of the agents can move out of their current nodes. This means that $v_{2\ell-1}$ remains unvisited forever.

In this execution, the number of missing edges is always ℓ and the network is obviously kept connected. Thus, the theorem holds. □

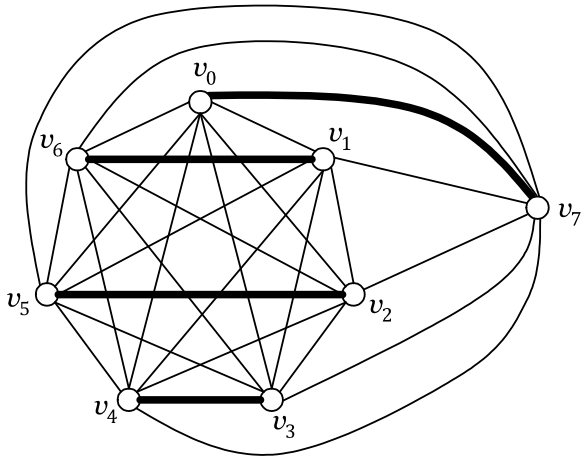

Figure 3.2: Example of a graph for $\ell = 4$ and $k = 2\ell - 1 = 7$ that cannot be explored by $2\ell - 1$ agents and its coloring. The bold lines are the edges of $E_8^{(0)}$.

Bound on Exploration Time

Let $\mathcal{G} \in \mathcal{W}(\ell)$. Since $\mathcal{W}(\ell) \subset H(\ell)$, $2\ell + 1$ agents can clearly completes the exploration by Algorithm 3.1 in graph $\mathcal{G}$. Interestingly, when executed on $\mathcal{G} \in \mathcal{W}(\ell)$, it can be shown that the time complexity of exploration can be bounded under the fully-synchronous scheduler. More specifically, we show that within $\Delta^n(\Delta + 1)^k(n - 1)^k$ rounds, all nodes of the graph have been visited at least once by a team of $k = 2\ell + 1$ agents.

We prove the theorem by a sequence of lemmas. First of all, we can easily show that $2\ell + 1$ agents executing Algorithm 3.1 cannot be all prevented from moving at any given round.

Lemma 3.4.1. *If* $2\ell + 1$ *agents activated fully-synchronously execute Algorithm 3.1 in* ℓ*-bounded 1-interval TVGs, at least one of them succeeds to move at every round.*

Proof. There exist two cases as in the proof of Lemma 3.3.1: at round t, (i) there exists a node v containing more than $\delta_v - 1$ agents, and (ii) there does not exist such a node.

In the first case, since there are more than $\delta_v - 1$ agents at v, every port is occupied by one agent at t since every agent is activated. In addition to that, v has at least one adjacent edge present at t by the connectivity of the TVG. This implies that at least one agent succeeds to move at round t.

In the second case, each agent occupies one port by assumption and by fully-synchronous activation, which means that $2\ell + 1$ ports are occupied. Moreover, at most ℓ edges are missing at each round, which means that at most 2ℓ ports are blocked at each round. It follows that at least one agent can move at round t also in this case.

□

For $\mathcal{E}^{\mathsf{aug}}$ of Algorithm 3.1, the following lemma holds.

Lemma 3.4.2. *In an augmented execution of Algorithm 3.1 by $2\ell + 1$ agents, any two augmented configurations are different.*

Proof. First note that Lemma 3.4.1 precludes the same two consecutive augmented configurations C_t^{aug} and C_{t+1}^{aug} in an augmented execution $\mathcal{E}^{\mathsf{aug}}$ of Algorithm 3.1 where no agents move between C_t^{aug} and C_{t+1}^{aug}.

Suppose that there exist two augmented configurations C_t^{aug} and $C_{t'}^{\mathsf{aug}}$ for $t < t'$ in $\mathcal{E}^{\mathsf{aug}}$. Let $\mathcal{E}_{t,t'}^{\mathsf{aug}} = C_t^{\mathsf{aug}} C_{t+1}^{\mathsf{aug}} \cdots C_{t'-1}^{\mathsf{aug}}$ be a subsequence of $\mathcal{E}^{\mathsf{aug}}$. In this case, the adversary can create an infinite augmented execution from $\mathcal{E}^{\mathsf{aug}}$ by repeating $\mathcal{E}_{t,t'}^{\mathsf{aug}}$, which means that the adversary can create an (augmented) execution where $2\ell + 1$ agents cannot complete the exploration forever. This contradicts Theorem 3.3.2. Thus, the lemma holds. □

We are now ready to show an upper bound on the exploration time of Algorithm 3.1, which is obtained by calculating the maximum length among all the augmented executions.

Lemma 3.4.3. *The length of any possible augmented execution by $k = 2\ell + 1$ agents is bounded by $\Delta^n(\Delta + 1)^k(n - 1)^k$.*

Proof. Let α be the maximum length among all the possible augmented executions. By Lemma 3.4.2, α is bounded by the number of possible augmented configurations in an execution.

The number of possible configurations on a fixed node set $V' \subseteq V$ is bounded by $\Delta^{|V'|}(|V'|(\Delta + 1))^k$, which corresponds to all the combinations of the possible values of $\mathsf{pointer}_v$ (i.e., $\Delta^{|V'|}$) and all of the the agents' locations (i.e., $(|V'|(\Delta + 1))^k$). Notice that only $\mathsf{pointer}_v$ of each node v is used as a variable in Algorithm 3.1. Since the number of nodes visited by an agent is not decreasing during the exploration, the exploration time is smaller than or equal to the sum of

$\Delta^{|V'|}(|V'|(\Delta+1))^k$ for $1 \leq |V'| \leq n-1$, i.e., $\alpha \leq \sum_{|V'|=1}^{n-1} \Delta^{|V'|}(|V'|(\Delta+1))^k \leq \Delta^n(\Delta+1)^k(n-1)^k$ rounds. □

It then follows that:

Theorem 3.4.5. *In* Fsync*, Algorithm 3.1 executed by $k = 2\ell + 1$ anonymous agents explores any ℓ-bounded 1-interval connected TVG within $\Delta^n(\Delta+1)^k(n-1)^k$ rounds.*

Note that, as a consequence, we obtain a *terminating exploration* algorithm for ℓ-bounded 1-interval connected TVGs.

Theorem 3.4.6. *In* Fsync*, with knowledge of n and k,* exploration with termination *of an arbitrary ℓ-bounded 1-interval connected temporal graph $\mathcal{W}(\ell)$ can be achieved in n^{n+2k} rounds by $k = 2\ell + 1$ agents.*

Exploration by 2ℓ Agents

The result of the previous section can be used to obtain a perpetual exploration algorithm of ℓ-bounded 1-interval connected graphs by 2ℓ agents (which know n and k). The solution (Algorithm 3.2 below) is obtained by applying to Algorithm 3.1 bounding the waiting time of an agent blocked on a missing edge.

In fact, while an agent keeps waiting for a missing edge forever in Algorithm 3.1, in Algorithm 3.2 an agent waits for a missing edge up to kT rounds where T is calculated on the basis of the results of Section 3.4.2.

Apart from the waiting time, the rest of the algorithm is the same as in Algorithm 3.1: each node has $\mathsf{pointer}_v$ pointing at a port. When agent a visits v, a checks each port in ascending order from the port pointed by $\mathsf{pointer}_v$. If a finds some unoccupied port p, a moves to the port and sets $\mathsf{pointer}_v$ to $p+1$. If a finishes to check all the ports and they all are occupied, a does nothing.

Variable Waiting of an agent represents the elapsed time since the last round when the agent moved to the current port.

Lemma 3.4.4. *Let 2ℓ agents execute Algorithm 3.2. If an agent waits at u for a missing edge $e = (u, v)$ for kT rounds, during this time either another agent starts to wait for e at v, or every node is visited by an agent at least once.*

Algorithm 3.2 Computation at node v

```
1: if on a port then
2:     Waiting ← Waiting + 1
3:     if Waiting > kT then
4:         exit the current port
5: if not on a port then
6:     Waiting ← 0
7:     i ← 0
8:     p ← pointer_v
9:     while i < δ_v ∧ port p is occupied do
10:        p ← (p + 1) mod δ_v
11:        i ← i + 1
12:    if i < δ_v then
13:        pointer_v ← (p + 1) mod δ_v
14:        move to the port p
```

Proof. Suppose that an agent a at u starts to wait for a missing edge (u, v) at round t and (u, v) is kept missing for the next kT rounds (including t).

We first show that there exist T successive rounds in $[t, t + kT)$ during which all the agents but a do not satisfy predicate Waiting $> kT$ even if their selected edge remains missing.

We show the claim by contradiction. We assume that in any interval of T successive rounds in $[t, t + kT)$, there is an agent that satisfies Waiting $> kT$.

By assumption, at least k agents other than a must satisfy Waiting $> kT$, since $kT/T = k$. This means that at least one agent (different from a) satisfies the predicate twice since the number of the agents (excluding a) is $k - 1$. However, once an agent satisfies Waiting $> kT$ at round $t' \in [t, t + kT)$, the agent never satisfies the predicate again in $[t, t + kT)$ since the length of the interval is kT. This is a contradiction. Thus, there exist T successive rounds in $[t, t + kT)$ during which all the agents (except for a) do not satisfy Waiting $> kT$ even if their chosen edge is kept missing.

Now, we show the lemma, i.e., show that another agent at v starts to wait for $e = (u, v)$ or the

exploration is completed. Suppose that no agent at v starts to wait for e in these T rounds. Since e is missing during these T rounds, during that time the network (without e) can be considered as a $(\ell - 1)$-bounded 1-interval connected TVG. By Theorem 3.4.5, every node of the network without e is visited at least once by an agent during these T rounds, because none of them starts to wait for e at v during that time by assumption. Thus, the lemma holds. □

Theorem 3.4.7. *In* Fsync*, any ℓ-bounded 1-interval connected temporal graph $\mathcal{G} \in \mathcal{W}(\ell)$ can be explored by $k = 2\ell$ anonymous agents with knowledge of n and k.*

Proof. There clearly exists at least one node v which is visited at an infinite number of rounds. We then show that all the neighbors of v are also visited at an infinite number of rounds by agents. We prove it by contradiction. Suppose that a neighbor u of v is visited at only a finite number of rounds and let t' be the last round when u is visited. Since v is visited at an infinite number of rounds and the agents execute Algorithm 3.2, some agent a visiting v eventually chooses (v, u) as the edge from which a moves after t'. If (v, u) appears by the kT-th round since a chooses it, a visits u as soon as (v, u) appears. Otherwise, another agent visits u by Lemma 3.4.4. It follows that u is eventually visited after t', which is a contradiction.

By the connectivity assumption, we can apply inductively the claim (e.g., the neighbors of a neighbor of v are also visited at an infinite number of rounds) to all the nodes, proving the theorem. □

From Theorems 3.4.4 and 3.4.7, we have:

Theorem 3.4.8. *In* Fsync*, with knowledge of n and k, the exploration of all ℓ-bounded 1-interval connected TVGs is possible iff $k \geq 2\ell$.*

3.5 Exploration of 1-Interval Connected Graphs with a Leader

In this section, we continue to consider the class $\mathcal{W}(\ell)$ of 1-interval connected temporal graphs with bounded missing edges, but we turn our attention to the case when one agent, the *leader*, is distinguishable from the others (the *non-leaders*). Also in this setting, we establish tight bounds for the exploration of this class of temporal graphs in Ssync and in Fsync showing that the presence of the leader allows the exploration to be performed using one fewer agent.

3.5.1 Semi-Synchronous Model

In this section, we show that, if there exists a leader, the bounds decrease by one in the exploration of ℓ-bounded 1-interval TVGs. In fact, we show that, $\mathcal{G} \in \mathcal{W}(\ell)$ can be explored by 2ℓ agents with one leader, while there exist graphs $\mathcal{G} \in \mathcal{W}(\ell)$ that cannot be explored by $2\ell - 1$ agents with one leader.

Impossibility

We start by showing the impossibility result.

Theorem 3.5.1. *There exist 1-interval connected time-varying graphs $\mathcal{G} \in \mathcal{W}(\ell)$ that cannot be explored by $k = 2\ell - 1$ agents with a leader. The result holds even if the agents know n, m and k, and whiteboards are of unbounded size.*

Proof. We construct a graph $\mathcal{G}'$ using the graph $\mathcal{G}$ employed in the proof of Theorem 3.4.1, where two new nodes v and w are connected to u and we use $2\ell - 2$ copies of stars instead of 2ℓ copies (see Figure 3.3). The subgraph corresponding to $\mathcal{G}$ (including u) is denoted by $\mathcal{G}'_1$ and the subgraph induced by u, v and w is denoted by $\mathcal{G}'_2$.

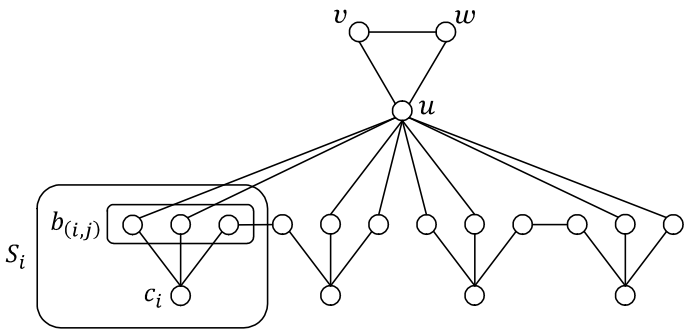

Figure 3.3: Example of a graph for $\ell = 3$ and $k = 2\ell - 1 = 5$ that cannot be explored by $2\ell - 1$ agents with a leader.

Let each non-leader agent a_i be on one of the nodes c_i, and the leader agent $\hat{a}$ be on w.

Consider $\mathcal{G}'_2$ and the following behavior of the adversary: whenever $\hat{a}$ chooses the port corresponding to (v, w) the adversary deletes (v, w), otherwise it deletes (u, w). With these

dynamics, $\hat{a}$ never visits u; moreover $\hat{a}$ has no effect on the exploration of $\mathcal{G}'_1$.

Consider now $\mathcal{G}'_1$: we let the adversary delete at most $\ell - 1$ edges at each round. Then, by Theorem 3.4.1 and since $\hat{a}$ has no effect on the exploration of $\mathcal{G}'_1$, the $2\ell - 2 = 2(\ell - 1)$ non-leader agents are also prevented from visiting u. Clearly, the number of missing edges at each round is at most ℓ and the graph is always connected. □

Exploration by 2ℓ Agents with a Leader

We now describe a strategy for 2ℓ agents (one of which is a leader) to explore ℓ-bounded 1-interval connected graphs. The general idea is simple: the leader agent always changes its chosen edge whenever it is blocked by a missing edge, while a non-leader agent always waits for its chosen edge to appear.

However, if we implement this strategy using one pointer for each node, like we did in Sections 3.3 and 3.4, two problems can occur: (*i*) a *broken rotor* and (*ii*) a *skipped port*.

A broken rotor is a pointer that can be changed by the adversary freely. Since the leader changes a pointer whenever it is blocked, the adversary can make the leader choose pointers in such a way that the leader is repeatedly blocked. To avoid this situation, we use an additional pointer $\mathsf{pointerL}_v$ for each node v that only the leader can change.

A skipped port is a port that remains unused. Suppose that a port p_v at node v is occupied by the leader. Since non-leaders skip an occupied port, they continue to skip p_v as long as the leader occupies p_v. However, the leader changes its port from p_v once it is blocked at p_v. In this way, the adversary can keep p_v unused, which can prevent a node from being explored. To avoid this situation, (*a*) $\mathsf{pointerL}_v$ is changed so that $\mathsf{pointerL}_v$ points to an occupied port p if and only if the agent occupying p is the leader, (*b*) a non-leader waits for an occupied port to be unoccupied when the port is pointed by $\mathsf{pointerL}_v$, and (*c*) the leader does not move to a port as long as there is a non-leader existing at the same node and is not on a port.

Algorithm 3.3 is the exploration algorithm of the leader and Algorithm 3.4 is the exploration algorithm of the non-leaders. In Algorithms 3.3 and 3.4, Setting $\mathsf{pointerL}_v$ to -1 is done to prevent $\mathsf{pointerL}_v$ from pointing to a port occupied by a non-leader. We assume that $\mathsf{pointerL}_v$ is initialized to -1.

First, we show that $\mathsf{pointerL}_v$ behaves correctly.

Algorithm 3.3 Computation of the leader at node v

```
1:  if on a port then
2:      exit the current port
3:  if (not on a port) ∧ (there is no non-leader not on a port) then
4:      i ← 0
5:      p ← pointerL_v + 1
6:      while i < δ_v ∧ port p is occupied do
7:          p ← (p + 1) mod δ_v
8:          i ← i + 1
9:      if i < δ_v then
10:         pointerL_v ← p
11:         move to the port p
12:     else
13:         pointerL_v ← −1
```

Lemma 3.5.1. *Variable $\mathsf{pointerL}_v$ points at an occupied port if and only if the agent occupying the port is the leader.*

Proof. ($\Leftarrow$) When the leader moves to a port p, it changes $\mathsf{pointerL}_v$ to p.

($\Rightarrow$) Let us show the contraposition of the claim: if the agent occupying port p is a non-leader, $\mathsf{pointerL}_v$ never points to p. At the beginning, the value of each $\mathsf{pointerL}_v$ is -1 and thus the claim holds. If a non-leader, say a_i, decides to move to a port p and $\mathsf{pointerL}_v$ points to p, then a_i changes $\mathsf{pointerL}_v$ to -1 before moving to p. By induction, $\mathsf{pointerL}_v$ never points to a port occupied by a non-leader. □

Theorem 3.5.2. *Any ℓ-bounded 1-interval connected temporal graph $\mathcal{G} \in \mathcal{W}(\ell)$ can be explored by $k = 2\ell$ agents with a leader under the semi-synchronous scheduler with eventual transport.*

Proof. Consider the leader executing Algorithm 3.3 and the $2\ell - 1$ non-leaders executing Algorithm 3.4. First, we show that unless $2\ell - 1$ non-leaders are all blocked forever, the $2\ell - 1$ non-leaders visit all the nodes infinitely often. Note that the adversary needs ℓ transient edges to block $2\ell - 1$ non-leaders.

Algorithm 3.4 Computation of a non-leader at node v

```
1:  if not on a port then
2:      i ← 0
3:      p ← pointer_v
4:      while i < δ_v ∧ port p is occupied do
5:          if p = pointerL_v then
6:              i ← δ_v, pointer_v ← p
7:              break from this loop
8:          p ← (p + 1) mod δ_v
9:          i ← i + 1
10:     if i < δ_v then
11:         pointer_v ← (p + 1) mod δ_v
12:         if p = pointerL_v then
13:             pointerL_v ← −1
14:         move to the port p
```

First assume that some non-leader agents can move from a node to another node infinitely often. Let A_m be the non empty set of such non-leaders, let t_e be the round such that after t_e, every agent $b \notin A_m$ is kept blocked forever, and let $x(t)$ be the total number of agent moves from a node to another node over all the agents in A_m from round t_e of the execution up to time t. Since $a \in A_m$ is never blocked by a transient edge, $x(t)$ goes to infinity (for $t \to \infty$). Thus, there exists a node, say v, which is visited at an infinite number of rounds by $a \in A_m$. Then, by an argument similar to the one used in the proof of Theorem 3.3.2, we can show that every neighbor of v connected with a recurrent edge is also visited at an infinite number of rounds by $a \in A_m$ and, inductively, that all the nodes are visited at an infinite number of rounds.

Suppose instead that every non-leader agent is blocked at some port forever after some round, and let $t'_e > t_e$ be a round when they are all blocked and all the 2ℓ transient edges have disappeared forever. In this case, we show that the leader completes the exploration. First observe that, since all the non-leaders are blocked at some port, after round t'_e the leader is never required to stop to wait for non-leaders to move to a port.

Moreover, since ℓ missing edges are transient and do not exist anymore after time t'_e, from this time, the network can be regarded as a static network with 2ℓ unusable ports: the $2\ell - 1$ occupied by non-leaders and one unoccupied. The leader, by construction, just skips the ports that are not available. In doing so, it executes the rotor-router algorithm on the static network induced by deleting all the transient edges from the footprint of the network. Hence, by the property of the rotor-router algorithm, the leader correctly performs the exploration. □

From Theorems 3.5.1 and 3.5.2, we have:

Theorem 3.5.3. *In SSYNC, with the existence of a leader, the exploration of all ℓ-bounded 1-interval connected TVGs is possible iff $k \geq 2\ell$.*

3.5.2 Fully-Synchronous Model

In this section, we show that, if there exists one leader and the agents are activated in FSYNC, the bounds on the number of agents for exploration in ℓ-bounded 1-interval TVGs decreases even further. In fact, we show that, with a leader, $\mathcal{G} \in \mathcal{W}(\ell)$ can be explored by $2\ell - 1$ agents if $\ell \geq 2$ (it is clear that when $\ell = 1$ and $k = 2\ell - 1 = 1$, the exploration is impossible), while there exist graphs $\mathcal{G} \in \mathcal{W}(\ell)$ that cannot be explored by $2\ell - 2$ agents.

Impossibility

We now consider ℓ-bounded, 1-interval connected TVGs operating under a fully-synchronous scheduler and we show that there exist TVGs that cannot be explored by $2\ell - 2$ agents with one leader ($2\ell - 3$ non-leaders and one leader agent), even if the agents know n, m, and k.

Theorem 3.5.4. *In FSYNC, there exist 1-interval connected time-varying graphs $\mathcal{G} \in \mathcal{W}(\ell)$ that cannot be explored by $k = 2\ell - 2$ agents with one leader. The result holds even if the agents know n, m and k, and whiteboards are of unbounded size.*

Proof. We construct a graph $K'_{2\ell-2}$ by adding two nodes u and w to the graph $K_{2\ell-2}$ used in the proof of Theorem 3.4.4 and connecting them to $v_{2\ell-3}$ (see Figure 3.4). The subgraph corresponding to $K_{2\ell-2}$ (including $v_{2\ell-3}$) is denoted by $K_{(1)}$ and the subgraph induced by $v_{2\ell-3}$, u, and w is denoted by $K_{(2)}$.

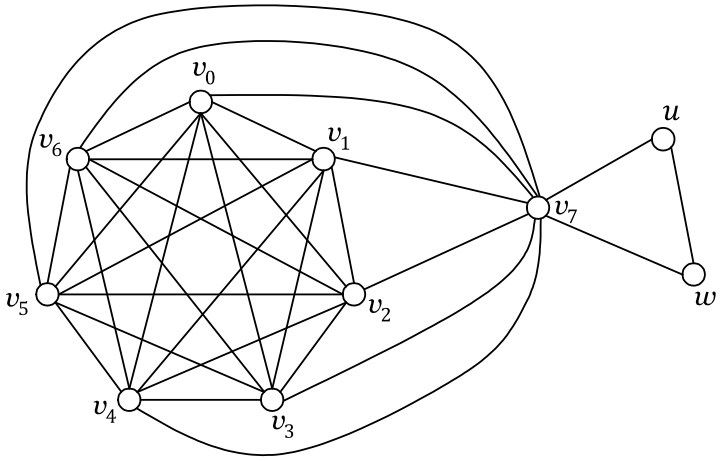

Figure 3.4: Example of a graph for $\ell = 5$ and $k = 2\ell - 2 = 8$ that cannot be explored by $2\ell - 2$ agents with one leader. It is constructed with the graph in Figure 3.2 and nodes u and w being connected to $v_{2\ell-3}$.

Let each non-leader a_i be on each v_i and let the leader agent $\hat{a}$ be on w. Consider $K_{(2)}$ and the following behavior of the adversary: whenever $\hat{a}$ chooses the port corresponding to $(v_{2\ell-3}, w)$ the adversary deletes $(v_{2\ell-3}, w)$, otherwise it deletes (u, w). With these dynamics, $\hat{a}$ never visits $v_{2\ell-3}$; moreover, $\hat{a}$ has no effect on the exploration of $K_{(1)}$.

For $K_{(1)}$, we let the adversary delete at most $\ell - 1$ edges at each round. Then, by Theorem 3.4.4 and since $\hat{a}$ has no effect on the exploration of $K_{(1)}$, the $2\ell - 3 = 2(\ell - 1) - 1$ non-leaders are also prevented from visiting $v_{2\ell-3}$. Clearly, the number of missing edges at each round is at most ℓ and the graph is always connected. □

Bound on Exploration Time

For the exploration algorithm using $2\ell - 1$ agents with one leader, we first consider the upper bound of exploration time of the exploration algorithm using 2ℓ agents with one leader, i.e., Algorithms 3.3 and 3.4. By a similar argument as the one used in Section 3.4.2, it can be shown that the time complexity of the exploration can be bounded under the fully-synchronous scheduler. More specifically, we show that within $\Delta^n(\Delta + 1)^{k+n}(n - 1)^k$ rounds, all nodes of the graph have been visited at least once by $k = 2\ell$ agents with one leader.

We prove the theorem by a sequence of lemmas. First of all, we show that the leader executing Algorithm 3.3 and $2\ell - 1$ non-leaders executing Algorithm 3.4 cannot be all prevented from moving or changing their port at any given round.

Lemma 3.5.2. *If 2ℓ agents activated fully-synchronously execute Algorithms 3.3 (for the leader) and 3.4 (for the non-leaders) in ℓ-bounded 1-interval TVGs, at least one of them succeeds to change its location at every round (i.e., moving to a port or a neighbor, or changing its port).*

Proof. We have two cases as in the proof of Lemma 3.3.1: at round t, (i) there exists a node v containing more than $\delta_v - 1$ agents, or (ii) there does not exist such a node.

In the first case, we can show the claim by the same argument used in the proof of Lemma 3.3.1.

We then consider the second case. If some agent is not on a port, this agent moves to a port or a neighbor. If every agent is on a port, the leader tries to change its port by construction. Note that since every node v is occupied by at most $\delta_v - 1$ agents, there is at least one unoccupied port at every node. Thus, the leader succeeds to change its port.

□

Using the same argument as the one of the proof of Lemma 3.4.2, we have:

Lemma 3.5.3. *In an augmented execution of Algorithm 3.3 executed by the leader and Algorithm 3.4 executed by the $2\ell - 1$ non-leaders, any two augmented configurations are different.*

Proof. We can show the lemma by the same argument used in the proof of Lemma 3.4.2.

□

We are now ready to show an upper bound on the exploration time of Algorithms 3.3 and 3.4, which is obtained by calculating the maximum length among all the augmented executions.

Lemma 3.5.4. *The length of any possible augmented execution of Algorithm 3.3 executed by the leader and Algorithm 3.4 executed by the $2\ell - 1$ non-leaders is bounded by $\Delta^n(\Delta+1)^{k+n}(n-1)^k$.*

Proof. Let α be the maximum length among all the possible augmented executions. By Lemma 3.5.3, α is bounded by the number of possible augmented configurations in an execution.

The number of possible configurations on a fixed node set $V' \subseteq V$ is bounded by $(\Delta(\Delta+1))^{|V'|}(|V'|(\Delta+1))^k$, which corresponds to all the combinations of the possible values of $\mathsf{pointer}_v$ and $\mathsf{pointerL}_v$ (i.e., $(\Delta(\Delta+1))^{|V'|}$) and all of the the agents' locations (i.e., $(|V'|(\Delta+1))^k$). Notice that only $\mathsf{pointer}_v$ and $\mathsf{pointerL}_v$ of each node v are used as variables in Algorithms 3.3 and 3.4. Since the number of visited nodes is not decreasing during the exploration, the exploration time is smaller than or equal to the sum of $(\Delta(\Delta+1))^{|V'|}(|V'|(\Delta+1))^k$ for $1 \leq |V'| \leq n-1$, i.e., $\alpha \leq \sum_{|V'|=1}^{n-1}(\Delta(\Delta+1))^{|V'|}(|V'|(\Delta+1))^k \leq \Delta^n(\Delta+1)^{k+n}(n-1)^k$ rounds. □

It then follows that:

Theorem 3.5.5. *In* Fsync*, the leader executing Algorithm 3.3 and* $2\ell-1$ *non-leaders executing Algorithm 3.4 explore any* ℓ*-bounded 1-interval connected TVG within* $\Delta^n(\Delta+1)^{k+n}(n-1)^k$ *rounds.*

Note that, as a consequence, we obtain a *terminating exploration* algorithm for ℓ-bounded 1-interval connected TVGs.

Theorem 3.5.6. *In* Fsync*, with knowledge of n and k, and the existence of a leader,* exploration with termination *of an arbitrary* ℓ*-bounded 1-interval connected temporal graph* $\mathcal{W}(\ell)$ *can be achieved in* $n^{2(n+k)}$ *rounds by* $k = 2\ell$ *agents.*

Exploration by $2\ell-1$ Agents with a Leader

The result of the previous section can be used to obtain a perpetual exploration algorithm of ℓ-bounded 1-interval connected graphs by $2\ell-1$ agents (which know n and k) one of which is a distinguishable leader. The solution (Algorithm 3.5 below) is obtained by applying Algorithm 3.4, appropriately bounding the waiting time of a non-leader blocked on a missing edge. The algorithm for the leader is the same as the one used in the previous section (i.e., Algorithm 3.3).

In fact, while a non-leader keeps waiting for a missing edge forever in Algorithm 3.4, in Algorithm 3.5 a non-leader waits for a missing edge up to $(k-1)T$ rounds where T is calculated on the basis of the results of Section 3.5.2.

Apart from the waiting time, Algorithm 3.5 is the same as Algorithm 3.4: each node has $\mathsf{pointer}_v$ pointing at a port and $\mathsf{pointerL}_v$ for the leader. When a non-leader a_i visits v, a_i checks

each port in ascending order from the port pointed by $\mathsf{pointer}_v$. If a_i finds the port occupied by the leader, a_i waits till the leader leaves the port. If a_i finds an unoccupied port p, a_i moves to the port and sets $\mathsf{pointer}_v$ to $p+1$ and additionally if p is equal to $\mathsf{pointerL}_v$, $\mathsf{pointerL}_v$ to -1. If a_i finishes to check all the ports and they all are occupied, a_i does nothing.

Variable Waiting of a non-leader represents the elapsed time since the last round when the non-leader moved to the current port.

Algorithm 3.5 Computation of a non-leader at node v

```
1:  if on a port then
2:      Waiting ← Waiting + 1
3:      if Waiting > (k − 1)T then
4:          exit the current port
5:  if not on a port then
6:      Waiting ← 0
7:      i ← 0
8:      p ← pointer_v
9:      while i < δ_v ∧ port p is occupied do
10:         if p = pointerL_v then
11:             i ← δ_v, pointer_v ← p
12:             break from this loop
13:         p ← (p + 1) mod δ_v
14:         i ← i + 1
15:     if i < δ_v then
16:         pointer_v ← (p + 1) mod δ_v
17:         if p = pointerL_v then
18:             pointerL_v ← −1
19:         move to the port p
```

Lemma 3.5.5. *Let the leader execute Algorithm 3.3 and the $2\ell-2$ non-leaders execute Algorithm 3.5. If a non-leader waits at u for a missing edge $e=(u,v)$ for $(k-1)T$ rounds starting from round t, then in $[t, t+(k-1)T)$ rounds there exist T successive rounds during which all the non-leaders*

do not satisfy predicate Waiting $> (k-1)T$ *even if their selected edge remains missing.*

Proof. Suppose that a non-leader a_i at u starts to wait for a missing edge (u, v) at round t and (u, v) is kept missing for the next $(k-1)T$ rounds (including t).

We show the lemma by contradiction. We assume that in any interval of T successive rounds in $[t, t+(k-1)T)$, there is a non-leader that satisfies Waiting $> (k-1)T$.

By assumption, at least $k-1$ non-leaders other than a_i must satisfy Waiting $> (k-1)T$ since $(k-1)T/T = k-1$. This means that at least one non-leader (different from a_i) satisfies the predicate twice since the number of non-leaders (excluding a_i) is $k-2$. However, once a non-leader satisfies Waiting $> (k-1)T$ at round $t' \in [t, t+(k-1)T)$, the non-leader never satisfies the predicate again in $[t, t+(k-1)T)$ since the length of the interval is $(k-1)T$. This is a contradiction. □

Lemma 3.5.6. *Let the leader execute Algorithm 3.3 and the* $2\ell-2$ *non-leaders execute Algorithm 3.5. If a non-leader waits at u for a missing edge* $e = (u, v)$ *for* $(k-1)T$ *rounds, during this time either another non-leader starts to wait for e at v, or every node is visited by an agent at least once.*

Proof. Suppose that a non-leader a_i at u starts to wait for a missing edge (u, v) at round t and (u, v) is kept missing for the next $(k-1)T$ rounds (including t).

By Lemma 3.5.5, in interval $[t, t+(k-1)T)$ there exist T successive rounds, say $\mathcal{I}$, during which all the non-leaders do not satisfy predicate Waiting $> (k-1)T$ even if their selected edge remains missing. Suppose that no non-leader starts to wait for e at v in $\mathcal{I}$. Since e is missing during $\mathcal{I}$, the network (without e) can be considered as a $(\ell-1)$-bounded 1-interval connected TVG during $\mathcal{I}$. By Theorem 3.5.5, $2\ell-2 = 2(\ell-1)$ agents with one leader complete the exploration of the $(\ell-1)$-bounded TVGs in the T successive rounds. This means that every node of the network (without e) is visited by an agent during $\mathcal{I}$ at least once because none of the non-leaders starts to wait for e at v during that time by assumption. Thus, the lemma holds.

□

Lemma 3.5.7. *Let the leader execute Algorithm 3.3 and the* $2\ell-2$ *non-leaders execute Algorithm 3.5. If there is a node visited at only a finite number of rounds (by the leader or non-leaders)*

and there is another node visited at only a finite number of rounds by non-leaders, every node is visited at only a finite number of rounds by non-leaders.

Proof. Suppose that a node v is visited at only a finite number of rounds by agents and another node u is visited at only a finite number of rounds by non-leaders. Let t_1 be the last round when v is visited by an agent or u is visited by non-leaders.

We first show that all the neighbors of u are visited at only a finite number of rounds by non-leaders. We prove this by contradiction, assuming that a neighbor w of u is visited at an infinite number of rounds by non-leaders. Eventually, an agent a_1 at w chooses (w, u) to move after t_1. Since u is never visited at after t_1, a_1 is kept blocked on w at the port of (w, u) for $(k-1)T$ rounds. By Lemma 3.5.6, however, another non-leader visits u or every node is visited by an agent at least once after t_1. Both cases contradict to the assumption and thus all the neighbors of u are visited at only a finite number of rounds by non-leaders. Since the network is connected, we can apply inductively the claim to all the nodes, proving the lemma. □

Theorem 3.5.7. *In Fsync, with knowledge of n and k, if $\ell \geq 2$, any ℓ-bounded 1-interval connected temporal graph $\mathcal{G} \in \mathcal{W}(\ell)$ can be explored by $k = 2\ell - 1$ agents with one leader.*

Proof. Consider the leader executing Algorithm 3.3 and non-leaders executing Algorithm 3.5. The proof follows the same lines of the one of Theorem 3.4.7. There clearly exists at least one node v which is visited at an infinite number of rounds. We then show that all the nodes are visited at an infinite number of rounds. Two cases are considered: *Case a)* v is visited at an infinite number of rounds by non-leaders and *Case b)* v is visited at only a finite number of rounds by non-leaders.

Case a) Suppose that v is visited at an infinite number of rounds by non-leaders. We show that all the neighbors of v are also visited at an infinite number of rounds by agents. We prove it by contradiction, assuming that a neighbor u of v is visited at only a finite number of rounds by agents and letting t_1 be the last round when u is visited by an agent. Since v is visited at an infinite number of rounds by the non-leaders executing Algorithm 3.5, some non-leader a_i visiting v eventually chooses (v, u) to move. If (v, u) appears within $(k-1)T$ rounds, a_i visits u in the period, which is a contradiction. Otherwise, another agent visits u by Lemma 3.5.6. It follows that u is eventually visited after t_1, which is a contradiction.

Case b) Suppose that v is visited at only a finite number of rounds by non-leaders. We show by contradiction that all the neighbor of v is visited at an infinite number of rounds. Assume that a neighbor of v is visited at only a finite number of rounds by agents. It follows by Lemma 3.5.7 that every node is visited at only a finite number of rounds by non-leaders. This means that no non-leader exists in the network by the definition. This is a contradiction since $\ell \geq 2$ and at least two non-leaders exist in the network.

In either case, all the neighbors of v is visited at an infinite number of rounds. Since the network is connected, we can apply inductively the claim to all the nodes, proving the theorem.

□

From Theorems 3.5.4 and 3.5.7, we have:

Theorem 3.5.8. *Under the fully-synchronous scheduler, with knowledge of n and k and the existence of a leader, if $\ell \geq 2$, the exploration of all ℓ-bounded 1-interval connected TVGs is possible iff $k \geq 2\ell - 1$.*

3.6 Conclusion

In this chapter, we considered perpetual exploration of temporal graphs with arbitrary and unknown topology, focusing on the number of agents that are necessary and sufficient to perform the task. We considered two common dynamic models: temporally connected networks, and 1-interval connected (or always connected) networks with a bounded number of missing edges at each round. We derived tight bounds for both models under fully synchronous and semi-synchronous settings, both when the agents are anonymous and when there is a leader.

Our algorithms use at each node v a rotor-router mechanism; this can be implemented with either a constant number of movable tokens that can be placed on the ports of v, or with a whiteboard of size $O(\log \delta_v)$ bits.

Chapter 4

Exploration of Dynamic Tori

4.1 Introduction

In this chapter, we consider exploration of a dynamic torus under some constraints on the dynamics (or topology changes).

An $\nu \times \mu$ torus ($3 \leq n \leq \mu$) is considered as a collection of ν row rings and μ column rings. The constraint on the dynamics is that each ring should be 1-interval connected, which allows at most one link to be missing at any time in each ring.

For the $\nu \times \mu$ dynamic torus, we propose exploration algorithms with and without a view. In this chapter, an agent with the view can detect which incident links are missing (if some link is missing) before determining its next move. On the other hand, without the view, an agent has to determine its next move without knowing which incident links are missing, which makes the agent stay on the same node when it tries to move through a missing link.

The main contribution of this chapter is to clarify the necessary and sufficient number of agents to explore an $\nu \times \mu$ dynamic torus with and without the view. Specifically, it is proven that, without the view, $\nu + 1$ agents are necessary and sufficient to explore the $\nu \times \mu$ dynamic torus. It also proven that, with the view, $\lceil \nu/2 \rceil + 1$ agents are necessary and sufficient when $\nu \neq 4$ and, $\lceil \nu/2 \rceil + 2$ (i.e., four) agents are necessary and sufficient when $\nu = 4$ to explore the $\nu \times \mu$ dynamic torus. With respect to the time complexity, we propose asymptotically time-optimal algorithms with and without the view.

The results are summarized in Table 4.1

Table 4.1: Exploration time on dynamic tori.

		#agents	#rounds
without the view	$3 \leq \nu \leq \mu$	less than $\nu + 1$	The exploration is **impossible**.
		$\nu + 1$	$O(\nu\mu(\mu - \nu + 1))$ (optimal when $\mu - \nu = O(1)$)
		$\nu + 2$	$O(\nu\mu)$ (optimal)
with the view		less than $\lceil \nu/2 \rceil + 1$	The exploration is **impossible**
	$5 \leq \nu \leq \mu$	$\lceil \nu/2 \rceil + 1$	$O(\nu\mu(\mu - \nu + 1))$ (optimal when $\mu - \nu = O(1)$)
	$3 \leq \nu \leq \mu$	$\lceil \nu/2 \rceil + 2$	$O(\nu\mu)$ (optimal)

4.2 Preliminary

4.2.1 Network

In this chapter, we consider the exploration with termination on dynamic networks whose foot print is a $\nu \times \mu$ torus where ν (resp., μ) is the number of rows (resp., columns) of the torus, V is $\{v_{i,j} \mid 0 \leq i \leq \nu - 1, 0 \leq j \leq \mu - 1\}$, and E is $\{(v_{i,j}, v_{i+1 \bmod \nu, j}), (v_{i,j}, v_{i,j+1 \bmod \mu}) \mid 0 \leq i \leq \nu - 1, 0 \leq j \leq \mu - 1\}$. We assume $3 \leq \nu \leq \mu$ to avoid self-loops or multiple edges. Hereafter, we omit the notations of mod ν or mod μ from modulo operations when it is clear from the context. Intuitively, G consists of ν row rings and μ column rings. A row ring R_i (resp, a column ring C_j) is a subgraph of G induced by $\{v_{i,j} \mid 0 \leq j \leq \mu - 1\}$ (resp, $\{v_{i,j} \mid 0 \leq i \leq \nu - 1\}$) (see Figure 4.1). Each row and column ring is a discrete time-varying graph and 1-interval connected (or connected at any round) where, at each round $t \in \mathbb{Z}^+$, one of its links may be missing. Since each of the row and column rings is 1-interval connected, the dynamic torus $\mathcal{G}$ is also 1-interval connected.

Each node $v_{i,j}$ is labeled by a pair (i, j) of its row number i and column number j. In this chapter, we assume four links incident to $v_{i,j}$ are locally labeled at $v_{i,j}$: $(v_{i,j}, v_{i,j+1})$, $(v_{i,j}, v_{i,j-1})$,

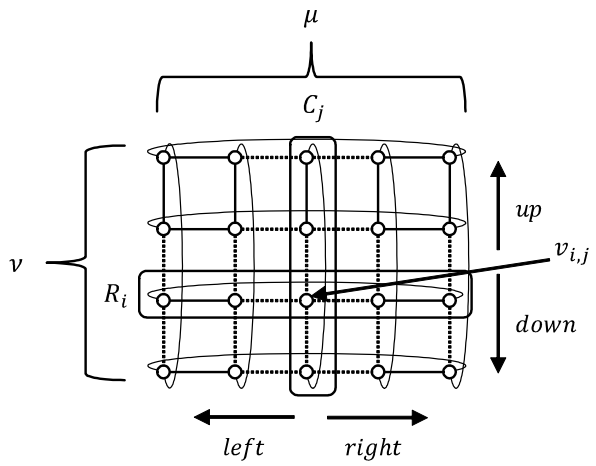

Figure 4.1: Example of an $\nu \times \mu$ torus and R_i and C_j of the torus.

$(v_{i,j}, v_{i+1,j})$, $(v_{i,j}, v_{i-1,j})$ are labeled by *right*, *left*, *down*, *up*, respectively.

Each node has four rooms, $right_v$, $left_v$, up_v and $down_v$, which correspond to the *right*, *left*, *up*, and *down* links, respectively.

4.2.2 Agents

A set A of k agents initially occupies arbitrary positions. An agent has knowledge of the torus size, that is, both ν and μ. An agent has no tool to communicate with other agents.

An agent may use a view giving the agent which incident links of its current node is present at the current round. When an agent use the view, we say that the model is *with the link presence detection*. Otherwise, we say that the model is *without the link presence detection*.

The actions of all agents is fully synchronized, i.e., we consider FSYNC model in this chapter.

4.2.3 Configuration

We call the locations of all the agents a *configuration*. That is, a configuration is a multiset (or bag) $W : V \to \mathbb{Z}^+$ with size k, i.e., $\sum_{v \in V} W(v) = k$. For any integer $t \geq 0$, we define the configuration in round t as W_t such that for any $v \in V$, $W_t(v) = x$ holds if and only if exactly x agents stay on node v at the beginning of round t. Note that W_t does not give the information

about the rooms where the k agents stay.

We call configuration W_0 an *initial* configuration. The initial configuration W_0 is arbitrary, i.e., an adversary can put each agent on any node in W_0. We assume that all the agents are not in rooms in round 0. This assumption does not lose generality: even when an agent is in any room in round 0, it can move out from a room by the beginning of round 1 and then all agents are not in any room in round 1.

4.3 Subroutines for 1-interval connected rings

In this section, we present several subroutines for 1-interval connected rings that are used as building blocks utilized in exploration algorithms of a dynamic torus. In the subroutines, we use the procedure Move($dir \mid p_1 : s_1 \ ; \ p_2 : s_2 \ ; \ \ldots \ ; \ p_\ell : s_\ell$) which is associated with each agent state, where $dir \in \{right, left, down, up, nil\}$, p_i is a predicate and s_i is a state that an agent enters when p_i is satisfied. At each round, an agent decides its state by repeating the following process until it enters state **Return** or a state where no predicate is satisfied: it tries to find the satisfied predicate with the smallest index among $p_1, p_2, \ldots, p_l$ in Move of the current state, and changes to the state corresponding to the predicate if it exists. If an agent decides its state as **Return**, it returns from the current operation. If an agent decides its state as a state other than **Return**, the agent moves to the room in the direction *dir* of the state. If *dir* is *nil*, an agent stays in the current node but not in a room. In Move, the following local variables are used.

- *time*: the number of rounds from the start of the subroutine called last. It is initialized to 0 when this subroutine is called.
- *current*: the node that the agent currently exists on.

The following predicate is used.

- *catches*: the predicate of an agent a holds if and only if the room corresponding to *dir* is occupied by an agent and the current room of a is different from the room in the current snapshot.

Exploration: ExplorationUp [15], which is defined by the pseudo code of Algorithm 4.1, is an algorithm to explore (1-interval connected) column rings of the torus. ExplorationUp guarantees that, if two or more agents staying in a column ring C_j start ExplorationUp simultaneously, then they succeed in exploring C_j within the following $3v$ rounds.

The execution of each agent is as follows. If an agent in C_j invokes ExplorationUp, then it starts to move *up* and tries to explore C_j until *time* reaches $3v$. If it catches another agent at node v during the exploration, that is, finds another agent in up_v of the current node, then it bounces back and starts to move *down* until *time* reaches $3v$.

ExplorationLeft for a row ring R_i is similarly defined by replacing *up*, *down*, and v in Algorithm 4.1 to *left*, *right*, and μ, respectively. ExplorationLeft finishes at the 3μ-th round. The following lemma holds.

Algorithm 4.1 ExplorationUp

1: In state **Init**:
2: Move(*up* | *time* $\geq 3v$: **Return**; *catches* : **Bounce**);
3: In state **Bounce**:
4: Move(*down* | *time* $\geq 3v$: **Return**);

Lemma 4.3.1. *If two or more agents are in a column (resp, row) ring and they start ExplorationUp (resp, ExplorationLeft) simultaneously, they explore the column (resp, row) ring by the* $3v$*-th (resp,* 3μ*-th) round from the start of ExplorationUp (resp, ExplorationLeft).*

Proof. In [15], authors proved that ExplorationUp explores a 1-interval connected ring when there are two agents in the ring. So we show that, when there are three or more agents in a (1-interval connected) column ring C_j, ExplorationUp explores C_j.

Hereinafter, we use *renaming* by which we can ignore the case where an agent cannot move because of mutual exclusion rule. Suppose that agent a on node v cannot move *dir* in a round because of mutual exclusion rule. At the round, there must exist another agent a' in a room on v corresponding to *dir*, and a' is renamed a. As a result, we can consider a is in the room.

Now, we show the lemma. There exists at least one agent b that always tries to move *up* during the execution of ExplorationUp since an agent bounces back only when there is another agent

moving *up*. By the 2ν-th round, b succeeds to move *up* in ν rounds or fails to move in ν rounds by missing links. In the former, exploration is completed. In the latter, there exists an (renamed) agent b' which succeeds to move *up* in $\nu + 1$ rounds or catches b during the 2ν rounds since there is at most one missing link in C_j in each round. If b' succeeds to move $\nu + 1$ rounds, exploration is completed. When b' catches b, b' starts to move *down*. Then, in ν additional rounds, b and (renamed) b' complete exploration of C_j since at most one link is missing from C_j and either b or b' succeeds to move at any round unless they are on adjacent nodes: b' is present in the node just above the node that b exists on. Thus, by the 3ν-th round, agents complete exploration of C_j. Similarly, the lemma holds for ExplorationLeft. □

Node arrangement: ArrangementUp(i) which is defined by the pseudo code of Algorithm 4.2 works on a 1-interval connected column ring. By ArrangementUp(i), from an arbitrary initial configuration, one of two agents in a column ring C_j reaches $v_{i,j}$ if the two agents exist, two of three agents in C_j reach $v_{i,j}$ if the three agent exist, and three of four or more agents in C_j reach $v_{i,j}$ if the agents exist.

Agent behavior is similar to that in ExplorationUp during the first 3ν rounds, except that an agent changes its state to **Wait** when it reaches $v_{i,j}$. Once an agent changes its state to **Wait**, it keeps its state and location until the end of ArrangementUp(i). At the 3ν-th (resp, 6ν-th) round from the start of ArrangementUp(i), unless an agent is on $v_{i,j}$, it starts the second (resp, third) exploration that is exactly same as the first part. Agents stop their actions at the 9ν-th round from the start of ArrangementUp(i).

ArrangementLeft(j) for a row ring is similarly defined by replacing *up*, *down*, i, R_i, and ν in Algorithm 4.2 to *left*, *right*, j, C_j, and μ, respectively. ArrangementLeft(j) finishes at the 6μ-th round. The following lemma holds.

Lemma 4.3.2. *If k agents ($k \geq 2$) are in a column (resp, row) ring, say C_j (resp, R_i), and they start ArrangementUp(i) (resp, ArrangementLeft(j)) simultaneously,* $\min(k-1, 3)$ *agents in C_j (resp, R_i) reach and stay on $v_{i,j}$ by the 9ν-th (resp, 9μ-th) round from the start of ArrangementUp(i) (resp, ArrangementLeft(j)).*

Proof. Suppose that there are two or more agents in C_j and they invoke ArrangementUp(i)

Algorithm 4.2 ArrangementUp(i)

1: In state **Init**:
2: Move(*up* | *time* $\geq 3\nu$: **Init'**; *current* $\in R_i$: **Wait**; *catches* : **Bounce**);
3: In state **Bounce**:
4: Move(*down* | *time* $\geq 3\nu$: **Init'**; *current* $\in R_i$: **Wait**);
5: In state **Init'**:
6: Move(*up* | *time* $\geq 6\nu$: **Init''**; *current* $\in R_i$: **Wait**; *catches* : **Bounce'**);
7: In state **Bounce'**:
8: Move(*down* | *time* $\geq 6\nu$: **Init''**; *current* $\in R_i$: **Wait**);
9: In state **Init''**:
10: Move(*up* | *time* $\geq 9\nu$: **Return**; *current* $\in R_i$: **Wait**; *catches* : **Bounce''**);
11: In state **Bounce''**:
12: Move(*down* | *time* $\geq 9\nu$: **Return**; *current* $\in R_i$: **Wait**);
13: In state **Wait**:
14: Move(*nil* | *time* $\geq 9\nu$: **Return**);

simultaneously. We first show by contradiction that, by the 3ν-th round from the start of ArrangementUp(i), one agent reaches $v_{i,j}$. We assume that no agent reaches $v_{i,j}$ by the 3ν-th round from the start of ArrangementUp(i). Then, the agents behave in the same way as if they invoke ExplorationUp during the first 3ν rounds. By Lemma 4.3.1, at least one agent reaches $v_{i,j}$, which shows that the assumption was incorrect. Hence, one of the agents reaches and stays on $v_{i,j}$. Similarly, by the 6ν-th round from the start of ArrangementUp(i), another agent must visit $v_{i,j}$ if there are three or more agents in the column ring and only one agent exists in $v_{i,j}$. Furthermore, by the 9ν-th round from the start of ArrangementUp(i), another agent must visit $v_{i,j}$ if there are four or more agents in the column ring and only two agents exist in $v_{i,j}$. Similarly, the lemma holds for ArrangementLeft(j). □

In the following sections, we use ExplorationUp, ExplorationLeft, ArrangementUp(i) and ArrangementLeft(j) as modules. Note that, as ExplorationUp, ExplorationLeft, ArrangementUp(i) and ArrangementLeft(j) finish in exactly 3ν, 3μ, 9ν and 9μ rounds,

respectively, agents can detect and share their termination at the same time. Notice that the subroutines work in both models with and without the link presence detection.

4.4 Exploration without the link presence detection in tori

In this section, we consider the exploration of the $\nu \times \mu$ dynamic torus *without* the link presence detection. We first prove in Section 4.4.1 that a group of ν (or fewer) agents cannot explore a dynamic torus. In Section 4.4.2, we give a time-optimal (but not optimal with respect to the number of agents) algorithm by which a group of $\nu + 2$ (or more) agents explores the dynamic torus in $O(\nu\mu)$ rounds, which is shown to be asymptotically optimal. In Section 4.4.3, we give an algorithm by which a group of $\nu+1$ (or more) agents explores the dynamic torus in $O(\nu\mu(\mu-\nu+1))$ rounds. We also show that the algorithm is asymptotically optimal in terms of the number of rounds when $\mu - \nu = O(1)$.

4.4.1 Impossibility of exploration

Lemma 4.4.1. *Without the link presence detection, a group of ν (or fewer) agents cannot explore the $\nu \times \mu$ dynamic torus.*

Proof. Suppose that there is an agent on each node $v_{i,i}$ of T ($i = 0, 1, \ldots, \nu - 1$) in the initial configuration W_0 (see Figure 4.2). In this case, an adversary can delete all the links through which agents try to move since there is only one agent in each row and column ring. Then, all the agents cannot leave the current nodes forever. □

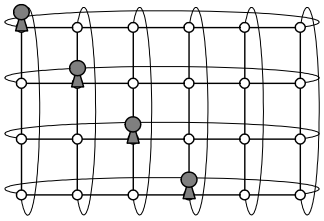

Figure 4.2: Impossible case without the link presence detection; $\nu = 4$ and $k = 4$.

4.4.2 Exploration by $\nu + 2$ agents

In this section, we propose an algorithm by which a group of $\nu + 2$ agents explores the $\nu \times \mu$ dynamic torus within $O(\nu\mu)$ rounds. The pseudo code is presented in Algorithm 4.3.

The algorithm makes two agents move to and explore R_i for each i ($0 \leq i \leq \nu - 1$) one by one. In order to explore R_i, agents first execute ArrangementLeft(0) at line 2, which locates two or three agents in C_0. Then, they execute ArrangementUp(i) at line 3 so that at least two agents are in R_i, and finally, two or more agents in R_i explore R_i by ExplorationLeft at line 4. By repeating this for $R_0, R_1, \ldots, R_{\nu-1}$, a group of $\nu + 2$ agents explores the $\nu \times \mu$ dynamic torus. The following theorems hold.

Algorithm 4.3 Exploration by $\nu + 2$ agents

1: **for** $i = 0$ to $\nu - 1$ **do**
2: ArrangementLeft(0);
3: ArrangementUp(i);
4: ExplorationLeft;
5: **end for**

Theorem 4.4.1. *Without the link presence detection, for $\nu \geq 3$, a group of $\nu + 2$ agents explores the $\nu \times \mu$ dynamic torus in $O(\nu\mu)$ rounds by Algorithm 4.3.*

Proof. Correctness: We show that at least two agents are in R_i at the start of ExplorationLeft at line 4 because two or more agents in R_i explore R_i by ExplorationLeft at line 4 by Lemma 4.1.

We consider two cases, one is that there is only one agent in R_i and the other is that there is no agent in R_i at the start of ArrangementLeft(0) at line 2.

Suppose that only one agent is in R_i at the start of ArrangementLeft(0) at line 2. At this moment, $\nu + 1$ agents are in $\nu - 1$ row rings. Therefore, 1) there are two row rings with two or more agents or 2) there is one row ring with three or more agents. Hence, at the end of ArrangementLeft(0) at line 2, at least two agents are in C_0 and one of them reaches R_i by ArrangementUp(i) at line 3.

Suppose that no agent is in R_i at the start of ArrangementLeft(0) at line 2. At this moment,

$\nu + 2$ agents are in $\nu - 1$ row rings. Therefore, 1) there are three row rings with two or more agents, 2) there is one row ring with two or more agents and there is another row ring with three or more agents, or 3) there is one row ring with four or more agents. Hence, at the end of ArrangementLeft(0) at line 2, at least three agents are in C_0. Therefore, two of the agents in C_0 reach R_i by ArrangementUp(i) at line 3.

Thus, at least two agents are in R_i at the start of ExplorationLeft at line 4.

Time complexity: It takes 9μ, 9ν and 3μ rounds to execute ArrangementLeft(0), ArrangementUp(i) and ExplorationLeft, respectively and the number of repetitions of the for-loop at lines 1–5 is ν. Hence, it takes $(12\mu + 9\nu)\nu = O(\nu\mu)$ ($\nu \leq \mu$) rounds that a group of $\nu + 2$ agents explore the $\nu \times \mu$ dynamic torus by Algorithm 4.3. □

Theorem 4.4.2. *Without the link presence detection, the required number of rounds to explore the $\nu \times \mu$ dynamic torus is $\Omega(\nu\mu)$ when $k = \nu + c$ for any constant $c \geq 1$.*

Proof: Suppose that there are $\nu + c$ agents in the $\nu \times \mu$ dynamic torus where $c \geq 1$. The adversary can make ν agents stay on their current nodes by the claim in the proof of Lemma 4.4.1. Thus, the remaining nodes should be visited by other c agents. Since the number of the remaining nodes is $\nu\mu - \nu$, it takes $(\nu\mu - \nu)/c = \Omega(\nu\mu)$ rounds to visit all the nodes with $\nu + c$ agents. □

From Theorems 4.4.1 and 4.4.2, Algorithm 4.3 is asymptotically optimal in terms of the number of rounds when $k = \nu + O(1)$.

4.4.3 Exploration by $\nu + 1$ agents

In this section, we present an algorithm by which a group of $\nu + 1$ (or more) agents explore the $\nu \times \mu$ dynamic torus within $O(\nu\mu(\mu - \nu + 1))$ rounds. The pseudo code is presented in Algorithm 4.4.

The algorithm makes two agents move to and explore each R_i ($0 \leq i \leq \nu - 1$) one by one. An agent executes ArrangementLeft(0) and ArrangementUp(i) at lines 2 and 3 where R_i is the row ring to be explored in the i-th iteration of the for-loop at lines 1–9. At the end of these executions, there exists at least one agent in R_i, which is shown in the proof of Theorem 4.4.3.

Then, an agent executes the for-loop at lines 4–7 $2(\mu - \nu) + 5$ times to locate another agent in R_i; in each of the first $\mu - \nu + 3$ iterations, letting $j' = j \bmod (\mu - \nu + 3)$, an agent tries to move to the destination in $C_{j'}$ by ArrangementLeft(j') at line 5 and executes ArrangementUp(i) at line 6 from $j = 0$ to $\mu - \nu + 2$, and in the following $\mu - \nu + 2$ iterations, an agent tries to move to the destination in $C_{j'}$ by ArrangementLeft(j') at line 5 and executes ArrangementUp(i) at line 6 from $j = 0$ to $\mu - \nu + 1$ again. Note that an agent which reaches R_i stays in R_i during the for-loop at lines 4–7.

When the for-loop at lines 4–7 finishes, at least two agents are in R_i, which is shown in the proof of Theorem 4.4.3, and these agents explore R_i by ExplorationLeft at line 8. By repeating this for $R_0, R_1, \ldots, R_{\nu-1}$, a group of $\nu + 1$ agents explores the $\nu \times \mu$ dynamic torus. The following theorem holds.

Algorithm 4.4 Exploration by $\nu + 1$ agents

```
1: for i = 0 to ν − 1 do
2:    ArrangementLeft(0);
3:    ArrangementUp(i);
4:    for j = 0 to 2(μ − ν) + 4 do
5:       ArrangementLeft(j mod (μ − ν + 3));
6:       ArrangementUp(i);
7:    end for
8:    ExplorationLeft;
9: end for
```

Theorem 4.4.3. *Without the link presence detection, for $\nu \geq 3$, a group of $\nu + 1$ (or more) agents explores the $\nu \times \mu$ dynamic torus in $O(\nu\mu(\mu - \nu + 1))$ rounds by Algorithm 4.4.*

Proof. **Correctness:** Without loss of generality, it suffices to show that the agents explore ring R_0 during the first execution (when $i = 0$) of the for-loop at lines 1–9. To prove that, it suffices to show that two agents exist in R_0 at the beginning of the first execution (when $i = 0$) of ExplorationLeft at line 8 because, by Lemma 4.3.1, two or more agents can explore a single row ring by ExplorationLeft at line 8.

First, we show that at least one agent is in R_0 at the end of ArrangementUp(0) at line 3. If one or more agents are already in R_0 at the beginning of ArrangementLeft(0) at line 2, they are still in R_0 at the end of ArrangementUp(0) at line 3 by the definitions of ArrangementUp(i) and ArrangementLeft(j). If no agent is in R_0 at the beginning of ArrangementLeft(0) at line 2, $\nu + 1$ agents are in $\nu - 1$ row rings other than R_0. Thus, there are two or more row rings with two agents or there is one or more row rings with three or more agents. Hence, Lemma 4.3.2 implies that at least two agents reach C_0 by ArrangementLeft(0) at line 2 and one of them reaches R_0 by ArrangementUp(0) at line 3. This means that at least one agent exists in R_0 at the end of ArrangementUp(0) at line 3.

In the following, we show that at least two agents are in R_0 at the end of the for-loop at lines 4–7, which directly gives the lemma. This proposition trivially holds if two or more agents exist in R_0 at the start of the for-loop. Hence, we consider only the case where exactly one agent, say a_0, exists in R_0 at the beginning of the for-loop. In what follows, we show by contradiction that another agent reaches R_0 during the execution of the for-loop. We assume that no agent other than a_0 reaches R_0 during the execution of the for-loop at lines 4–7. For simplicity, we ignore agent a_0 in the following discussion. For example, "there is at most one agent in each C_j" means "there is at most one agent *other than* a_0 in each C_j" in the following discussion.

We first consider global configurations during the for-loop at lines 4–7. At most one agent is in each C_j at the beginning of ArrangementUp(0) at line 6 since no agent reaches R_0 by ArrangementUp(0) at line 6. Therefore, for each $j' = j \bmod (\mu - \nu + 3)$ where $0 \leq j \leq 2(\mu - \nu) + 4$, two agents do not reach $C_{j'}$ by ArrangementLeft(j') at line 5. This means that there must not be two or more row rings with two agents and there must not be one or more row rings with three or more agents at the start of ArrangementLeft(j') at line 5. Otherwise, at least two agents reach $C_{j'}$ by ArrangementLeft(j') at line 5, which gives a contradiction. Note that ν agents exist in the $\nu - 1$ row rings $R_1, R_2, \ldots, R_{\nu-1}$ during the for-loop at lines 4–7. Hence, at the start of ArrangementLeft(j') at line 5, two agents are in some row ring $R_{i''}$ ($i'' > 0$) and, for each $i' \neq i''$, only one agent is in $R_{i'}$ (see Figure 4.3). Notice that i'' may differ for different iterations of the for-loop at lines 4–7.

Next, we consider the movements of agents. Agents do not move *down* since there is at most one agent in each C_j at the start of ArrangementUp(0) at line 6 (agents move *down* by

ArrangementUp(i) only when they catch another agent). It means that i'' does not increase throughout an execution. Moreover, since there must exist at least one agent in each R_i, an agent in R_i for $i > i''$ do not move *up*. Additionally, since any two of the agents which are not in $R_{i''}$ are not in the same row ring by the definition of i'', they do not move *right*.

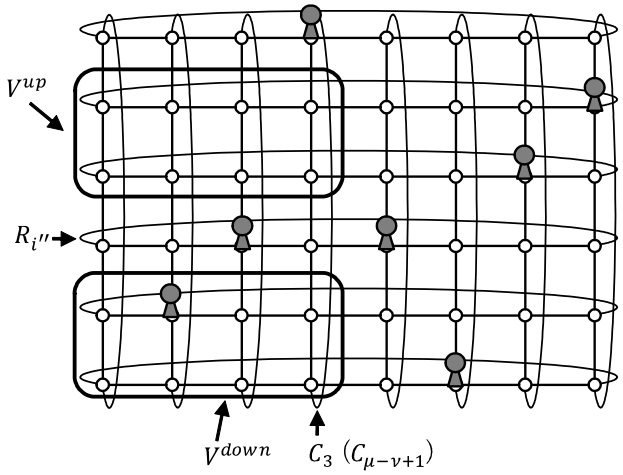

Figure 4.3: The configuration at the end of ArrangementLeft(j') at line 5 in the $(\mu - \nu + 3)$-th iteration of the for-loop at lines 4–7 for $\nu = 6$, $\mu = 8$, and $k = 7$: there are two agents in $R_{i''}$ and there are no agents in V^{up}.

Let $\tilde{i}$ denote the value of i'' at the end of ArrangementLeft(j') at line 5 in the $(\mu - \nu + 3)$-th iteration of the for-loop at lines 4–7. We show that, at the end of ArrangementLeft(j') at line 5 in the $(\mu - \nu + 3)$-th iteration of the for-loop at lines 4–7, there exists no agent in $V^{up} = \{v_{i,j} | 1 \leq i \leq \tilde{i} - 1, 0 \leq j \leq \mu - \nu + 1\}$. By the above claims, no agent enters into V^{up} from left (i.e., through C_0), up (i.e., through R_0), or down (i.e., through $R_{i''}$). Besides, no agent is in C_j other than $v_{i'',j}$ for each j where $0 \leq j \leq \mu - \nu + 2$ during ArrangementLeft(j') at line 5. Otherwise two agents exist in C_j for some j because one of the two agents in $R_{i''}$ always reaches C_j by ArrangementLeft(j') at line 5, which gives a contradiction. It means that no agent enters into V^{up} from right since agents cannot move *left* in the period from the end of ArrangementLeft(j') at line 5 to the start of ArrangementLeft($j' + 1$) at line 5. Therefore, at the end of ArrangementLeft(j') at line 5 in the $(\mu - \nu + 3)$-th iteration of the for-loop at lines

4–7, there exists no agent in V^{up}.

Now, we show that there is at least one agent in $V^{down} = \{v_{i,j} \mid \tilde{i}+1 \leq i \leq \nu-1, 0 \leq j \leq \mu-\nu+1\}$ at the end of ArrangementLeft(j') at line 5 in the $(\mu-\nu+3)$-th iteration of the for-loop at lines 4–7. There are at least two agents in any $\mu-\nu+2$ column rings since there is at most one agent in each C_j and there are ν agents (except for a_0). Thus, there are at least two agents in $V^{up} \cup V^{down} \cup \{v_{\tilde{i},j} \mid 0 \leq j \leq \mu-\nu+1\}$. There is no agent in V^{up} from the above claim. There is at most one agent in $\{v_{\tilde{i},j} \mid 0 \leq j \leq \mu-\nu+1\}$ because there must exist exactly one agent on $v_{i'',\mu-\nu+2}$ at the end of ArrangementLeft(j') at line 5 in the $(\mu-\nu+3)$-th iteration of the for-loop at lines 4–7. Hence, there is at least one agent in V^{down} (see Figure 4.3).

By the above claims, an agent in V^{down} does not move *up* and *right*, and one of the agents in $R_{i''}$ is always in $C_{j'}$ at the end of ArrangementLeft(j') at line 5. Thus, during the following $\mu-\nu+2$ iterations, there is some iteration where two agents are in $C_{j'}$ at the end of ArrangementLeft(j') at line 5. One of the agents in $C_{j'}$ reaches R_0 by ArrangementUp(0) at line 6. This is a contradiction. Hence, a group of $\nu+1$ (or more) agents explores the $\nu \times \mu$ dynamic torus by Algorithm 4.4.

Time complexity: As each execution of lines 5 and 6 takes $9(\nu+\mu)$ rounds and the number of repetitions of the for-loop in lines 4–7 is $2(\mu-\nu)+5$, each execution of the for-loop at lines 1–9 takes $9(\nu+\mu)(2(\mu-\nu)+5)+9\nu+12\mu = O(\mu(\mu-\nu+1))$ rounds since $\nu \leq \mu$. As the number of repetitions of the for-loop at lines 1–9 is ν, the total number of rounds required by Algorithm 4.4 is $O(\nu\mu(\mu-\nu+1))$ rounds. □

From Lemma 4.4.1 and Theorem 4.4.3, the following theorem and corollary hold.

Theorem 4.4.4. *Without the link presence detection, for $\nu \geq 3$, $\nu+1$ agents are necessary and sufficient to explore the $\nu \times \mu$ dynamic torus.*

Corollary 4.4.1. *Without the link presence detection, a group of $\nu+1$ agents explores the $\nu \times \mu$ dynamic torus in $O(\nu^2)$ rounds by Algorithm 4.4 when $\mu-\nu = O(1)$.*

In other words, Algorithm 4.4 is optimal in terms of the number of agents. It is also asymptotically optimal in terms of the number of rounds when $\mu-\nu = O(1)$.

4.5 Exploration with the link presence detection in tori

In this section, we consider the exploration of the $\nu \times \mu$ dynamic torus *with* the link presence detection. We first prove in Section 4.5.1 that a group of $\lceil \nu/2 \rceil$ (or fewer) agents cannot explore a dynamic torus and, for $\nu = 4$, a group of three (i.e., $\lceil \nu/2 \rceil + 1$) agents cannot explore the dynamic torus. In Section 4.5.2, we give a time-optimal (but not optimal with respect to the number of agents) algorithm by which a group of $\lceil \nu/2 \rceil + 2$ (or more) agents explores the dynamic torus in $O(\nu\mu)$ rounds for any $\nu \geq 3$. In Section 4.5.3, we give an algorithm by which a group of $\lceil \nu/2 \rceil + 1$ (or more) agents explores the dynamic torus in $O(\nu\mu(\mu - \nu + 1))$ rounds for $\nu \geq 5$. We also show that the algorithm is asymptotically optimal in terms of the number of rounds when $\mu - \nu = O(1)$. Finally, in Section 4.5.4, we propose an algorithm for $\nu = 3$ and $k = 3$ ($= \lceil \nu/2 \rceil + 1$). Note that, for $\nu \geq 3$, we present optimal results with respect to the number of agents.

4.5.1 Impossibility of exploration

First, we show a building block of the proof of the impossibility result: an adversary can forever contain an agent in a 2×2 subgrid consisting of four nodes, $v_{i,j}$, $v_{i+1,j}$, $v_{i,j+1}$ and $v_{i+1,j+1}$. Suppose that an agent, say a, is located at node $v_{i,j}$. By removing two links connecting $v_{i,j}$ to the nodes outside the subgrid (i.e., $v_{i-1,j}$ and $v_{i,j-1}$), the adversary can prevent a from going out from the subgrid. So a can move only in the subgrid. By repeating the argument, we can show that a cannot go out of the subgrid forever. Notice that even when another agent, say a', comes into the subgrid, the adversary can still prevent a from going out from the subgrid while a' may go out from the subgrid: the adversary always removes the two links by which a cannot go out from the subgrid, ignoring the location of a'.

Lemma 4.5.1. *With the link presence detection, for $\nu \geq 3$, a group of $\lceil \nu/2 \rceil$ or fewer agents cannot explore the $\nu \times \mu$ dynamic torus.*

Proof. We consider two cases, one is that ν is even and the other is that ν is odd. First, suppose that ν is even. Let $a_0, a_1, \ldots, a_{\nu/2-1}$ be $\nu/2$ agents in the torus and suppose a_i is located at $v_{2i,2i}$ in W_0 (see Figure 4.4). In this case, by the above building block, the adversary can contain each a_i in the subgrid consisting of $v_{2i,2i}$, $v_{2i+1,2i}$, $v_{2i,2i+1}$ and $v_{2i+1,2i+1}$ forever. Thus, when ν is even, a group of $\lceil \nu/2 \rceil$ agents cannot explore the dynamic torus.

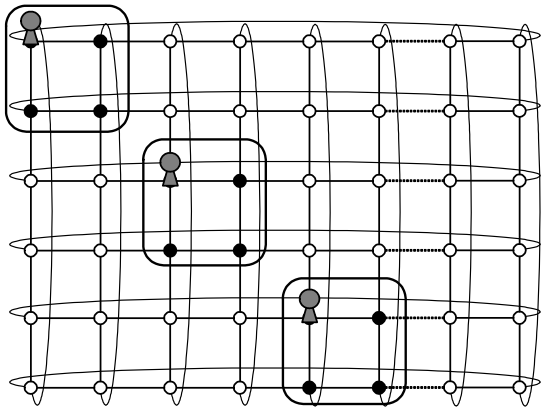

Figure 4.4: Impossible case with the link presence detection; $\nu = 6$ and $k = 3$.

Next, we consider the case where ν is odd. Let $a_0, a_1, \ldots, a_{(\nu-1)/2}$ be $(\nu + 1)/2$ agents in the torus and suppose that a_j $(0 \leq j \leq (\nu - 3)/2)$ are located at $v_{2j,2j}$ and $a_{(\nu-1)/2}$ is located at any node other than $v_{\nu-1,\nu-1}$. By the building block, the adversary can contain a_j in the subgrid consisting of $v_{2j,2j}$, $v_{2j+1,2j}$, $v_{2j,2j+1}$ and $v_{2j+1,2j+1}$ forever. Furthermore, the adversary can prevent $a_{(\nu-1)/2}$ from visiting $v_{\nu-1,\nu-1}$ forever by removing the link to $v_{\nu-1,\nu-1}$ every time it comes to a neighboring node of $v_{\nu-1,\nu-1}$. Consequently, no agent can visit $v_{\nu-1,\nu-1}$ and thus a group of $\lceil \nu/2 \rceil$ agents fails to explore the dynamic torus. □

The following slightly stronger impossibility holds for $\nu = 4$.

Lemma 4.5.2. *With the link presence detection, for $\nu = 4$, a group of three ($\lceil \nu/2 \rceil + 1$) or fewer agents cannot explore the $4 \times \mu$ dynamic torus.*

Proof. Let $V_{0,0} = \{v_{0,0}, v_{0,1}, v_{1,0}, v_{1,1}\}$, $V_{1,0} = \{v_{2,0}, v_{2,1}, v_{3,0}, v_{3,1}\}$, $V_{0,1} = \{v_{i,j} | 0 \leq i \leq 1, 2 \leq j \leq \mu - 1\}$ and $V_{1,1} = \{v_{i,j} | 2 \leq i \leq 3, 2 \leq j \leq \mu - 1\}$, and suppose that each $V_{0,0}$, $V_{1,0}$ and $V_{0,1}$ contains one agent. We show that no algorithm can change the situation where each of $V_{0,0}$, $V_{1,0}$ and $V_{0,1}$ contains one agent and, thus, all the nodes of $V_{1,1}$ remain unexplored. Let a_0 be the agent in $V_{0,0}$, a_1 be the agent in $V_{1,0}$ and a_2 be the agent in $V_{0,1}$.

If a_0 and a_1 are in different column rings (e.g., a_0 is on $v_{0,0}$ and a_1 on $v_{2,1}$) or they are on neighboring nodes (e.g., a_0 is on $v_{1,0}$ and a_1 is on $v_{2,0}$), the adversary can easily prevent a_0 and a_1

from moving to $V_{1,0}$ and $V_{0,0}$, respectively. Moreover, we do not consider the possibility that a_1 moves *left* or *right* direction. This is because, when a_1 tries to move *left* or *right*, the adversary can easily prevent a_1 from leaving from $V_{1,0}$. For a_0 and a_2, we do not consider the possibility that a_0 and a_2 are in different row rings, they are on neighboring nodes, or a_2 move *up* and *down*. So it suffices to consider the case where a_0 and a_1 are in the same column ring and a_0 and a_2 are in the same row ring, but no pair of agents is on neighboring nodes. Without loss of generality, we assume that a_0 is on $v_{0,0}$, a_1 is on $v_{2,0}$ and a_2 is on $v_{0,2}$ (see Figure 4.5).

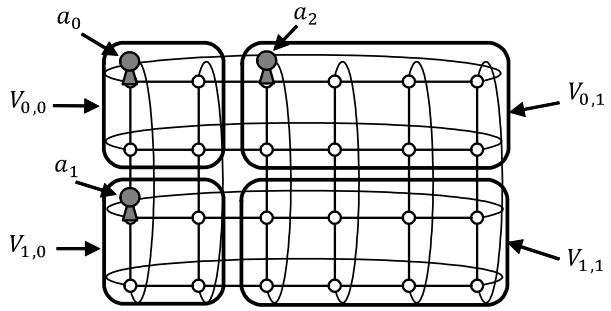

Figure 4.5: Impossible case with the link presence detection; $\nu = 4$ and $k = 3$.

Then, we consider behavior of agents and prove that, in any execution, the adversary can keep the situation where each of $V_{0,0}$, $V_{0,1}$ and $V_{1,0}$ contains one agent unchanged. At first, the adversary simulates the behavior of agents when no link is deleted. Then, depending on that simulation, the adversary decides whether it deletes a link or not and, if so, which link it deletes.

If a_0 and a_1 move in the same column direction, the adversary lets a_0 and a_1 move and deletes $(v_{0,1}, v_{0,2})$ and $(v_{0,2}, v_{3,2})$ so that a_2 cannot move out from $V_{0,1}$. If a_0 moves *down* and a_1 moves *up*, the adversary deletes $(v_{1,0}, v_{2,0})$, $(v_{2,0}, v_{2,m-1})$, $(v_{0,1}, v_{0,2})$ and $(v_{0,2}, v_{3,2})$ so that a_1 and a_2 cannot move out from $V_{1,0}$ and $V_{0,1}$, respectively. In this case, a_0 moves *down* and remains in $V_{0,0}$

If a_0 moves *up* and a_1 moves *down*, the adversary deletes $(v_{0,0}, v_{3,0})$. In this case, as a_0 can decide the direction using the knowledge that $(v_{0,0}, v_{3,0})$ is missing, the adversary simulates the movement of a_0 with the knowledge. If a_0 moves *down*, the adversary deletes $(v_{0,1}, v_{0,2})$ and $(v_{0,2}, v_{3,2})$ additionally. If a_0 and a_2 move in the same direction, the adversary lets them move. If a_0 moves *left* and a_2 moves *right*, the adversary deletes $(v_{0,0}, v_{0,\mu-1})$ additionally. If a_0 moves *right* and a_2 moves *left*, the adversary deletes $(v_{0,1}, v_{0,2})$ and $(v_{0,2}, v_{3,2})$ additionally.

From the above claim, the situation where each of $V_{0,0}$, $V_{0,1}$ and $V_{1,0}$ contains one agent does not change by any execution and, thus, all the nodes of $V_{1,1}$ remain unexplored. Thus, for $\nu = 4$ and $k = 3$ ($\lceil \nu/2 \rceil + 1$), the dynamic torus cannot be explored. □

4.5.2 Exploration by $\lceil \nu/2 \rceil + 2$ agents

With the link presence detection, at each round t, an agent can move *down* or *up* (if it wants to do) since each column ring has at most one missing link. This also holds for the direction *right* or *left*.

A main idea of exploration is that each agent moves to a neighboring node (if necessary) so that all agents should be in row rings with even row numbers or row rings with odd row numbers, and executes an algorithm similar to that in Section 4.4. Hereinafter, we call row rings with odd (resp, even) row numbers *odd (resp, even) row rings*, and call column rings with odd (resp, even) column numbers *odd (resp, even) column rings*.

ArrangementV(i) (Algorithm 4.5) makes every agent move one hop vertically (if necessary) to an even (resp, odd) row ring when i is even (resp, odd). The only exception is the case that both ν and i are odd: the algorithm allows agents in $R_{\nu-1}$ to move to R_0 in addition to odd row ring $R_{\nu-2}$. Remind that at most one agent can move in each direction of each link at each round by the mutual exclusion rule. Thus, ArrangementV(i) is executed for five rounds so that one of the conditions in Lemma 4.5.3 holds. In the following pseudo codes, we use the following notation and predicate.

- $v_{p,q}$: It denotes the current node of the agent which is executing the pseudo code.
- *stay* (predicate): It is true if i and p are even, i and p are odd, or ν and i are odd and $p = 0$.

ArrangementH(j) locating agents in even column rings or odd column rings (with exception that C_0 is included) is similarly defined by replacing i, *up*, *down*, $v_{p,q}$, and $v_{p-1,q}$ in Algorithm 4.5.3 to j, *left*, *right*, $v_{q,p}$, and $v_{q,p-1}$, and i and ν in the definition *stay* to j and μ, respectively. ArrangementH(j) also finishes at the fifth round. The following lemma holds.

Lemma 4.5.3. *At the end of ArrangementV(i) (resp, ArrangementH(j)), one of the following conditions holds.*

Algorithm 4.5 ArrangementV(i)

1: In state **Init:**
2: Move(nil | $time \geq 5$: **Return**; $\neg stay$: **Leave_Up**);
3: In state **Leave_Up:**
4: Move(up | $time \geq 5$: **Return**; $stay$: **Init**; $(v_{p,q}, v_{p-1,q})$ is missing : **Leave_Down**);
5: In state **Leave_Down:**
6: Move($down$ | $time \geq 5$: **Return**; $stay$: **Init**; $(v_{p,q}, v_{p-1,q})$ is not missing : **Leave_Up**);

1. *Predicate stay is true for every agent.*
2. *There is one row (resp, column) ring with two or more agents and there is another row (resp, column) ring with three or more agents.*
3. *There is one row (resp, column) ring with four or more agents.*

Proof. We consider the case where i is even because the following discussion also holds, with slight modification, for the case where i is odd. We assume that at least one agent stays on a node v of an odd row ring at the end of ArrangementV(i) (i.e., claim 1 does not hold), and show that, at this moment, there is one row ring with two or more agents and there is another row ring with three or more agents (i.e., claim 2 holds), or there is one row ring with four or more agents (i.e., claim 3 holds).

ArrangementV(i) continues for five rounds and at least one agent on v succeeds to move in each round. Since one agent stays on v at the end of ArrangementV(i), there must be at least six agents on v at the start of ArrangementV(i). Since five of them move *up* or *down* from v and they never move again by ArrangementV(i), there is one row ring with two or more agents and there is another row ring with three or more agents, or there is one row ring with four or more agents, that is, Condition 2 or 3 holds. Similarly, the lemma holds for ArrangementH(j). □

In the following, we present an algorithm by which a group of $\lceil \nu/2 \rceil + 2$ agents explores an $\nu \times \mu$ dynamic torus within $O(\nu\mu)$ rounds. The pseudo code is presented in Algorithm 4.6.

The algorithm makes two agents move to and explore R_i for each i ($0 \leq i \leq \nu - 1$) one by one. In order to explore each R_i, agents first execute ArrangementV(i) and ArrangementLeft(0)

at lines 2 and 3, which locates two or more agents in C_0 by Lemmas 4.3.2 and 4.5.3. Then, they execute ArrangementUp(i) at line 4 so that at least two agents are in R_i, and finally, these agents explore R_i by ExplorationLeft at line 5. By repeating this for $R_0, R_1, \ldots, R_{\nu-1}$, a group of $\lceil \nu/2 \rceil + 2$ agents explores the $\nu \times \mu$ dynamic torus. The following theorems and corollary hold.

Algorithm 4.6 Exploration by $\lceil \nu/2 \rceil + 2$ agents

1: **for** $i = 0$ to $\nu - 1$ **do**
2: ArrangementV(i);
3: ArrangementLeft(0);
4: ArrangementUp(i);
5: ExplorationLeft;
6: **end for**

Theorem 4.5.1. *With the link presence detection, for $\nu \geq 3$, a group of $\lceil \nu/2 \rceil + 2$ agents explores the $\nu \times \mu$ dynamic torus in $O(\nu\mu)$ rounds by Algorithm 4.6.*

Proof. **Correctness:** It suffices to show that at least two agents are in R_i at the start of ExplorationLeft at line 5 because two or more agents in R_i explore R_i by ExplorationLeft at line 5 by lemma 4.1.

We consider two cases, one is that there is only one agent in R_i and the other is that there is no agent in R_i at the end of ArrangementV(i) at line 2.

Suppose that only one agent is in R_i at the end of ArrangementV(i) at line 2. At the time, by Lemma 4.5.3, the other $\lceil \nu/2 \rceil + 1$ agents are in $\lceil \nu/2 \rceil - 1$ row rings. Therefore, there are two row rings with two or more agents or there is one row ring with three or more agents. Thus, by Lemma 4.3.2, at least two agents are in C_0 at the end of ArrangementLeft(0) at line 3 and one of them reaches R_i by ArrangementUp(i) at line 4.

Suppose that no agent is in R_i at the end of ArrangementV(i) at line 2. At the time, by Lemma 4.5.3, $\lceil \nu/2 \rceil + 2$ agents are in $\lceil \nu/2 \rceil - 1$ row rings. Hence, there are three row rings with two or more agents, there is one row ring with two or more agents and there is another row ring with three or more agents, or there is one row ring with four or more agents. Thus, by Lemma 4.3.2, at least three agents are in C_0 at the end of ArrangementLeft(0) at line 3 and two of them reach R_i by ArrangementUp(i) at line 4.

Thus, at least two agents are in R_i at the start of ExplorationLeft at line 5.

Time complexity: It takes 5, 9μ, 9ν and 3μ rounds to execute ArrangementV(i), ArrangementLeft(0), ArrangementUp(i) and ExplorationLeft, respectively, and the number of repetitions of the for-loop at lines 1–6 is ν. From these, by Algorithm 4.6, $\nu + 2$ (or more) agents explore the $\nu \times \mu$ dynamic torus in $(12\mu + 9\nu + 5)\nu = O(\nu\mu)$ rounds. (Note that $\nu \leq \mu$.) □

Corollary 4.5.1. *With the link presence detection, four agents are necessary and sufficient to explore the* $4 \times \mu$ *dynamic torus for any* $\mu \geq 4$.

Theorem 4.5.2. *With the link presence detection, the required number of rounds to explore the* $\nu \times \mu$ *dynamic torus with* k *agents is* $\Omega(\nu\mu)$ *when* $k = \lceil \nu/2 \rceil + c$ *for any constant* $c \geq 1$.

Proof. Suppose that there are $\lceil \nu/2 \rceil + c$ agents in the dynamic torus where $c \geq 1$.

The adversary can make each of $\lfloor \nu/2 \rfloor$ agents stay in their current 2×2 subgrids as shown in the proof of Lemma 4.5.1. Thus, the remaining nodes must be visited by at most $c + 1$ agents (the number of agents other than ones kept in subgrids is at most c when ν is even and $c + 1$ when ν is odd). Since the number of the remaining nodes is $\nu\mu - 4(\lfloor \nu/2 \rfloor)$, it takes $(\nu\mu - 4(\lfloor \nu/2 \rfloor))/(c + (\nu \bmod 2)) = \Omega(\nu\mu)$ rounds to visit all the nodes with $\lceil \nu/2 \rceil + c$ agents. □

From Theorem 4.5.1 and 4.5.2, Algorithm 4.6 is asymptotically optimal in terms of the number of rounds when $k = \lceil \nu/2 \rceil + O(1)$.

4.5.3 Exploration by $\lceil \nu/2 \rceil + 1$ agents

In this section, we present an algorithm with the link presence detection by which a group of $\lceil \nu/2 \rceil + 1$ (or more) agents explores the $\nu \times \mu$ dynamic torus within $O(\nu\mu(\mu - \nu + 1))$ rounds. The pseudo code is presented in Algorithm 4.7.

The algorithm makes two agents move to and explore R_i for each i $(0 \leq i \leq \nu - 1)$ one by one. At line 2, an agent executes ArrangementV(i) to move to an even row ring or an odd row ring (with the exception that R_0 is included). An agent executes ArrangementLeft(0) and ArrangementUp(i) at lines 3 and 4 where R_i is the row ring to be explored in the i-th iteration

of the for-loop at lines 1–12. At the end of these executions, there exists at least one agent in R_i (we explain this in the proof of Theorem 4.5.3).

Then, an agent executes the for-loop at lines 5–10 to locate another agent in R_i. Let j' denote $2(j \bmod (\lceil \mu/2 \rceil - \lceil \nu/2 \rceil + 3))$. A set of executions of ArrangementV(i) and ArrangementLeft(j') at lines 6 and 7 works like ArrangementLeft($j \bmod (\mu - \nu + 3)$) at line 5 in Algorithm 4.4. An agent executes the for-loop at lines 5–10 for $2(\lceil \mu/2 \rceil - \lceil \nu/2 \rceil) + 5$ times; in each of the first $\lceil \mu/2 \rceil - \lceil \nu/2 \rceil + 3$ iterations, an agent tries to move to the destination in C_{2j} from $j = 0$ to $\lceil \mu/2 \rceil - \lceil \nu/2 \rceil + 2$, and in the following $\lceil \mu/2 \rceil - \lceil \nu/2 \rceil + 2$ iterations, an agent tries to move to C_{2j} from $j = 0$ to $\lceil \mu/2 \rceil - \lceil \nu/2 \rceil + 1$ again. Note that an agent which reaches R_i stays in R_i until the for-loop at lines 5–10 finishes.

At the end of the for-loop at lines 5–10, at least two agents are in R_i, which we prove in the proof of Theorem 4.5.3 and these agents explore R_i by ExplorationLeft at line 11. By repeating this for $R_0, R_1, \ldots, R_{\nu-1}$, a group of $\lceil \nu/2 \rceil + 1$ agents explores the $\nu \times \mu$ dynamic torus. The following theorem holds.

Algorithm 4.7 Exploration by $\lceil \nu/2 \rceil + 1$ agents

```
1:  for i = 0 to ν − 1 do
2:      ArrangementV(i);
3:      ArrangementLeft(0);
4:      ArrangementUp(i);
5:      for j = 0 to 2(⌈μ/2⌉ − ⌈ν/2⌉) + 4 do
6:          ArrangementV(i);
7:          ArrangementLeft(2(j mod (⌈μ/2⌉ − ⌈ν/2⌉ + 3)));
8:          ArrangementH(0);
9:          ArrangementUp(i);
10:     end for
11:     ExplorationLeft;
12: end for
```

Theorem 4.5.3. *With the link presence detection, for $\nu \geq 5$, a group of $\lceil \nu/2 \rceil + 1$ agents explores the $\nu \times \mu$ dynamic torus in $O(\nu\mu(\mu - \nu + 1))$ rounds by Algorithm 4.7.*

Proof. **Correctness:** As we showed in Lemma 4.4.3, it is sufficient to show that two agents exist in R_0 at the beginning of the first execution (when $i = 0$) of ExplorationLeft at line 11. Thus, we fix $i = 0$ throughout this proof.

First, we show that at least one agent is in R_0 at the end of ArrangementUp(0) at line 4. If one or more agents are already in R_0 at the start of ArrangementV(0) at line 2, they are still in R_0 at the end of ArrangementUp(0) at line 4 by the definitions of ArrangementUp(i), ArrangementLeft(j) and ArrangementV(i). Thus, we consider the case where there is no agent in R_0 at the start of ArrangementV(0) at line 2. At the end of ArrangementV(0) at line 2, by Lemma 4.5.3, at least one of the followings holds: $\lceil \nu/2 \rceil + 1$ agents are in $\lceil \nu/2 \rceil - 1$ even row rings other than R_0, there is one row ring with two or more agents and there is another row ring with three or more agents, or there is one row ring with four or more agents. In the first case, there are two row rings with two or more agents or there is one row ring with three or more agents. Thus, Lemma 4.2 implies that at least two agents reach C_0 by ArrangementLeft(0) at line 3. In the second and third cases, Lemma 4.2 implies that at least three agents reach C_0 by ArrangementLeft(0) at line 3. This means that at least two agents exist in C_0 at the start of ArrangementUp(0) at line 4 and at least one of them reaches R_0 by ArrangementUp(0) at line 4. Hence, at least one agent exists in R_0 at the end of ArrangementUp(0) at line 4.

In the following, we show that at least two agents are in R_0 at the end of the for-loop at lines 5–10, which directly gives the lemma. This proposition trivially holds if two or more agents exist in R_0 at the start of the for-loop. Hence, we consider only the case where exactly one agent, say a_0, exists in R_0 at the start of the for-loop. In what follows, we show by contradiction that another agent reaches R_0 during the execution of the for-loop. We assume that no agent other than a_0 reaches R_0 during the execution of the for-loop at lines 5–10. For simplicity, we ignore a_0 in the following discussion. For example, "there is at most one agent in each C_j" means "there is at most one agent *other than* a_0 in each C_j" in the following discussion.

We first consider global configurations during the for-loop at lines 5–10. At most one agent is in each C_j at the start of ArrangementUp(0) at line 9 since no agent reaches R_0 by ArrangementUp(0) at line 9. Therefore, for each $j' = 2(j \bmod (\lceil \mu/2 \rceil - \lceil \nu/2 \rceil + 3))$ such that $0 \leq j \leq 2(\lceil \mu/2 \rceil - \lceil \nu/2 \rceil) + 4$, two agents do not reach $C_{j'}$ by ArrangementLeft(j') at line 7. This means that there must not be two row rings with two or more agents or there must

not be one row ring with three or more agents at the start of ArrangementLeft(j') at line 7. Otherwise, at least two agents reach $C_{j'}$ by ArrangementLeft(j') at line 7 and one of them reaches R_0 by ArrangementUp(0) at line 9, which gives a contradiction. Hence, at the end of ArrangementV(0) at line 6, neither Conditions 2 nor 3 of Lemma 4.5.3 hold and, thus, Condition 1 of Lemma 4.5.3 holds, that is, agents are in even row rings. Since $\lceil \nu/2 \rceil$ agents are in $\lceil \nu/2 \rceil - 1$ row rings, two agents are in some even row ring $R_{i''}$ ($i'' > 0$), and for each even number $i' \neq i''$, only one agent is in $R_{i'}$ and there is no agent in odd row rings at the end of ArrangementV(0) at line 6 (see Figure 4.6). Similarly, there is no agent in odd column rings at the end of ArrangementH(0) at line 8.

Next, we consider the movements of agents. Agents do not move *down* by ArrangementUp(0) at line 9 since there is at most one agent in each C_j during the execution of ArrangementUp(0) at line 9 (agents move *down* by ArrangementUp(i) only when they catch another agent). Moreover, since there must be at least one agent in each even row ring at the end of ArrangementV(0), an agent in even row ring R_i such that $i > i''$ exists in the same row ring R_i after executing ArrangementUp(0) and ArrangementV(0) at lines 9 and 6, that is, the agent moves *up* at most one hop by ArrangementUp(0) at line 9 and does not move *up* by ArrangementV(0) at line 6. Hence, the value of i'' at the start of ArrangementLeft($j' + 1$) at line 7 is no more than the value of i'' at the start of ArrangementLeft(j') at line 7. Additionally, since any two of the agents which are not in $R_{i''}$ are not in the same row ring by the definition of i'', they do not move *right*.

Let $\tilde{i}$ denote the value of i'' at the end of ArrangementLeft(j') at line 5 in the $(\mu - \nu + 3)$-th iteration of the for-loop at lines 5–10. We show that, at the end of ArrangementLeft(j') at line 7 in the $(\lceil \mu/2 \rceil - \lceil \nu/2 \rceil + 3)$-th iteration of the for-loop at lines 5-10, there exists no agent in $V^{up} = \{v_{i,j} | 1 \leq i \leq \tilde{i} - 1, 0 \leq j \leq 2(\lceil \mu/2 \rceil - \lceil \nu/2 \rceil + 1)\}$. By above claims, no agent enters into V^{up} from left (i.e., through C_0), up (i.e., through R_0), or down (i.e., through $R_{i''}$). Besides, no agent is in C_{2j} other than $v_{i'',2j}$ for each j where $0 \leq j \leq \lceil \mu/2 \rceil - \lceil \nu/2 \rceil + 2$ during ArrangementLeft(j') at line 7. Otherwise two agents exist in C_{2j} for some j because one of the two agents in $R_{i''}$ always reaches $v_{i'',2j}$ by ArrangementLeft(j') at line 7, which gives a contradiction. It means that no agent enters into V^{up} from right since agents cannot move *left* in the period from the end of ArrangementH(0) at line 7 (at the time, agents are in even collumn rings) to the start of

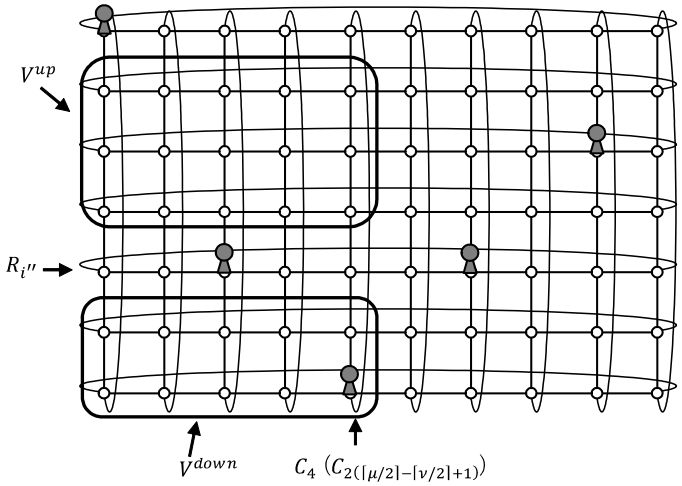

Figure 4.6: At the end of ArrangementLeft(j') at line 7 in the ($\lceil\mu/2\rceil - \lceil\nu/2\rceil + 3$)-th iteration of the for-loop at lines 5–10 for $\nu = 7$, $\mu = 10$, and $k = 5$: there are two agents in $R_{i''}$ and there are no agents in V^{up}.

ArrangementLeft($j' + 1$) at line 7. Therefore, at the end of ArrangementLeft(j') at line 7 in the ($\lceil\mu/2\rceil - \lceil\nu/2\rceil + 3$)-th iteration of the for-loop at lines 5–10, there exists no agent in V^{up}.

Now, we show that there is at least one agent in $V^{down} = \{v_{i,j} | \ \tilde{i} + 1 \leq i \leq n - 1, 0 \leq j \leq 2(\lceil\mu/2\rceil - \lceil\nu/2\rceil + 1)\}$ at the end of ArrangementLeft(j') at line 7 in the ($\lceil\mu/2\rceil - \lceil\nu/2\rceil + 3$)-th iteration of the for-loop at lines 5–10. There are at least two agents in any $\lceil\mu/2\rceil - \lceil\nu/2\rceil + 2$ even column rings since there is at most one agent in each C_j and there are $\lceil\nu/2\rceil$ agents except for R_0. Thus, there are at least two agents in $V^{up} \cup V^{down} \cup \{v_{i'',j} \mid 0 \leq j \leq 2(\lceil\mu/2\rceil - \lceil\nu/2\rceil + 1)\}$ at the end of ArrangementLeft(j') at line 5 in the ($\lceil\mu/2\rceil - \lceil\nu/2\rceil + 3$)-th iteration of the for-loop at lines 5–10. There is no agent in V^{up} from the above claim. There is at most one agent in $\{v_{i'',j} \mid 0 \leq j \leq 2(\lceil\mu/2\rceil - \lceil\nu/2\rceil + 1)\}$ because there must exist exactly one agent on $v_{i'',2(\lceil\mu/2\rceil-\lceil\nu/2\rceil+2)}$ at the end of ArrangementLeft(j') at line 7 in the ($\lceil\mu/2\rceil - \lceil\nu/2\rceil + 3$)-th iteration of the for-loop at lines 5–10. Hence, there is at least one agent in V^{down} (see Figure 4.6).

By the above claims, an agent in V^{down} does not move *up* or *right*, and one of the agents in $R_{i''}$ is always in $C_{j'}$ at the end of ArrangementLeft(j') at line 7. Thus, during the following $\lceil\mu/2\rceil - \lceil\nu/2\rceil + 2$ iterations, there is some iteration where two agents are in $C_{j'}$ at the end of

ArrangementLeft(j') at line 7. One of the agents in $C_{j'}$ reaches R_0 by ArrangementUp(0) at line 9. This is a contradiction. Hence, a group of $\lceil \nu/2 \rceil + 1$ (or more) agents explores the $\nu \times \mu$ dynamic torus by Algorithm 4.7.

Time complexity: As each execution of the for-loop at lines 5–10 takes $9(\nu + \mu) + 10$ rounds and the number of repetitions of the for-loop at lines 5–10 is $2(\lceil \mu/2 \rceil - \lceil \nu/2 \rceil) + 5$, each execution of the for-loop at lines 1–12 takes no more than $(9(\nu+\mu)+10)((\mu+1)-(\nu+1)+5)+9\nu+12\mu+5 = O(\mu(\mu - \nu + 1))$ rounds. As the number of repetitions of the for-loop at lines 1–12 is ν, the total number of rounds required by Algorithm 4.7 is $O(\nu\mu(\mu - \nu + 1))$ rounds. □

Additionally, the following corollary holds.

Corollary 4.5.2. *With the link presence detection, a group of* $\lceil \nu/2 \rceil + 1$ *agents explores the* $\nu \times \nu$ *dynamic torus in* $O(\nu^2)$ *rounds by Algorithm 4.7 when* $\mu - \nu = O(1)$.

In other words, Algorithm 4.7 is optimal in terms of the number of agents. It is also asymptotically optimal in terms of the number of rounds when $\mu - \nu = O(1)$.

4.5.4 Exploration by three agents for $\nu = 3$

In this section, we propose an algorithm by which a group of three agents explores the $3 \times \mu$ dynamic torus. This means that $\lceil \nu/2 \rceil + 1$ agents is sufficient to explore the $\nu \times \mu$ dynamic torus for $\nu = 3$. Note that, as explained below, there is a case where agents fail to explore the dynamic torus by Algorithm 4.7 when $\nu = 3$ and $k = 3$ $(= \lceil \nu/2 \rceil + 1)$.

Suppose that $\nu = 3$, $k = 3$ and one agent, say a, is on $v_{0,0}$ and two agents are in R_2 at the initial configuration (see Figure 4.7). Let us consider the first iteration of the for-loop at lines 1–12 (when $i = 0$). By ArrangementV(0) at line 2, agents do not move. An adversary lets only one of the two agent in R_2 move and reach C_0 by ArrangementLeft(0) at line 3 and prevents the agent from moving to R_0 by ArrangementUp(0) at line 4. Then, only a is in R_0 (on $v_{0,0}$) at the start of the for-loop at lines 5–10. In the for-loop at lines 5–10, the adversary can prevent agents from exploring R_0 by the following strategy. The adversary makes agents other than a move to R_2 by ArrangementV(0) at line 6, prevents a from moving and two agents in R_2 from meeting by ArrangementLeft(j') and ArrangementH(0) at lines 7 and 8 and prevents agents other

than a from moving to R_0 by ArrangementUp(0) at line 9. By repeating these, the adversary can prevent agents from exploring the nodes in R_0 other than $v_{0,0}$ during the first iteration of the for-loop at lines 5–10.

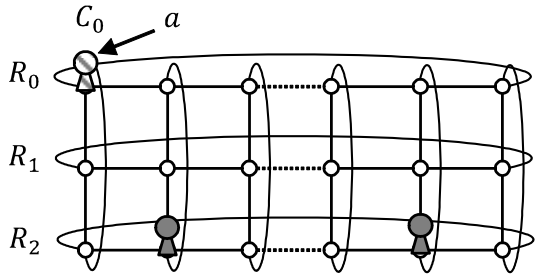

Figure 4.7: An initial configuration where one agent a is in $v_{0,0}$ and two agents are in R_2.

In the following two iterations of the for-loop at lines 1–12, the adversary makes a stay in $v_{0,0}$ and lets the other agents explore R_1 and R_2. Consequently, agents terminate Algorithm 4.7 without visiting the nodes in R_0 other than $v_{0,0}$.

Subroutines for $\nu = 3$

Before we propose the exploration algorithm, we propose subroutines ExplorationUp(j), ArrangementV_3(i) and MeetingV(i, j) which are presented in Algorithms 4.8, 4.9 and 4.10, respectively.

ExplorationUp(j) is the same as ExplorationUp defined in Algorithm 4.1, but with the addition of state **Wait**. Agents in C_j act as if they executed ExplorationUp. The other agents become **Wait** and stay at their current node during ExplorationUp(j). The following lemma holds.

Lemma 4.5.4. *By ExplorationUp(j), if two or more agents are in C_j and they start ExplorationUp(j) simultaneously, they explore C_j by the 3ν-th round from the start of ExplorationUp(j) while the other agents stay at their current nodes.*

Proof. Agents executing ExplorationUp(j) in C_j act as if they were executing ExplorationUp since *current* $\notin C_j$ does not hold for them. Thus, C_j is explored if two or more agents are in C_j

Algorithm 4.8 ExplorationUp(j)

1: In state **Init**:
2: Move($up \mid time \geq 3v$: **Return**; $current \notin C_j$: **Wait**; $catches$: **Bounce**);
3: In state **Bounce**:
4: Move($down \mid time \geq 3v$: **Return**);
5: In state **Wait**:
6: Move($nil \mid time \geq 3v$: **Return**);

at the start of ExplorationUp(j) by Lemma 4.3.1. Agents which are not in C_j change their state to **Wait** by satisfying $current \notin C_j$ as soon as they start ExplorationUp(j). Once they become **Wait**, they do not move till the end of ExplorationUp(j). □

ArrangementV_3(i) is the algorithm by which the three agents move to R_i or R_{i-1}. Agents which exist in R_{i+1} at the start of ArrangementV_3(i) succeed to move to R_i or R_{i-1}. Agents which exist in R_i or R_{i-1} stay at their current nodes. The following lemma holds.

Algorithm 4.9 ArrangementV_3(i)

1: In state **Init:**
2: Move($nil \mid time \geq 3$: **Return**; $p = i + 1$: **Leave_Up**);
3: In state **Leave_Up:**
4: Move($up \mid time \geq 3$: **Return**; $p = i \vee p = i - 1$: **Init**; $(v_{p,q}, v_{p-1,q})$ is missing : **Leave_Down**);
5: In state **Leave_Down:**
6: Move($down \mid time \geq 3$: **Return**; $p = i \vee p = i - 1$: **Init**; $(v_{p,q}, v_{p-1,q})$ is not missing : **Leave_Up**);

Lemma 4.5.5. *When $v = 3$ and $k = 3$, at the end of ArrangementV_3(i), the three agents are in R_i or R_{i-1}.*

Proof. Agents in R_i or R_{i-1} do not move during ArrangementV_3(i) since $p = i + 1$ does not hold for them and their states remain **Init**. Now, we consider agents in R_{i+1} at the start of ArrangementV_3(i). First, suppose that there are three agents in the same node, say $v_{i+1,j}$

($0 \leq j \leq \mu$), in R_{i+1} at the start of ArrangementV_3(i). They first change their states to **Leave_Up**. After that, they try to move to R_i if $(v_{p,q}, v_{p-1,q})$ exists; otherwise, they change their states to **Leave_Down** and try to move to R_{i-1}. Note that by the mutual exclusion rule, one of the agents succeeds to move for both cases. At the start of the second round of ArrangementV_3(i), the states of the two agents staying in $v_{i+1,j}$ is **Leave_Up** or **Leave_Down**. Starting from either state, they change their states to **Leave_Up** (resp., **Leave_Down**) and try to move to R_i (resp., R_{i-1}) if $(v_{p,q}, v_{p-1,q})$ exists (resp., does not exist) and one of them succeeds to move. At the third round, the remaining agent surely succeeds to move to R_i or R_{i-1}. For the cases where there is one agent or are two agents in $v_{i+1,j}$ at the start of ArrangementV_3(i), we can show by the similar way that the agent(s) is in R_i or R_{i-1} at the end of ArrangementV_3(i). □

MeetingV(i, j) makes an agent on $v_{i,j}$ and another agent on $v_{i-1,j}$ meet at $v_{i,j}$ or $v_{i-1,j}$ if they exist and makes the other agents wait two rounds at their current nodes. Algorithm 4.10 is a pseudo code of MeetingV(i, j). For simplicity, let us suppose that exactly one agent (say a) is on $v_{i,j}$, exactly one agent (say b) is on $v_{i-1,j}$, and no agent is on $v_{i+1,j}$ as in Figure 4.8a. The movement of a and b is as follows.

In the first round, if $(v_{i,j}, v_{i-1,j})$ exists, a stays at its current node (line 6) and b moves to $v_{i,j}$ (line 15) as in Figure 4.8b and, in the second round, both the agents stay at $v_{i,j}$ (lines 6 and 16). Otherwise, a stays at $v_{i,j}$ (line 8) and b moves to $v_{i+1,j}$ in the first round (line 18) as in Figure 4.8c. In this case, a and b can share the node to meet at the second round depending on whether $(v_{i,j}, v_{i+1,j})$ exists or not in the second round: a stays at $v_{i,j}$ (line 10) while b moves to $v_{i,j}$ (line 22) if $(v_{i,j}, v_{i+1,j})$ exists (see Figure 4.8d) or both of them move to $v_{i-1,j}$ (lines 12 and 24) otherwise (see Figure 4.8e).

For agents in column rings other than C_j, they stay their current nodes (lines 1 and 2). When there are several agents on $v_{i,j}$ (resp, $v_{i-1,j}$) at the start of MeetingV(i, j), exactly one of them behaves as a (resp, b) due to the mutual exclusion rule and other agents stay at $v_{i,j}$ (resp, $v_{i-1,j}$ at line 20).

Lemma 4.5.6. *The agents existing on $v_{i,j}$ or $v_{i-1,j}$ at the start of MeetingV(i, j) exist on $v_{i,j}$ or $v_{i-1,j}$ at the end of MeetingV(i, j). Moreover, if an agent (say a) exists on $v_{i,j}$ and another agent (say b) exists on $v_{i-1,j}$ at the start of MeetingV(i, j), these agents exist on the same node $v_{i,j}$ or*

Algorithm 4.10 MEETINGV(i, j)

```
1: if (q ≠ j) then
2:     wait two rounds;
3: else
4:     if (p = i) then
5:         if ((v_{i,j}, v_{i−1,j}) exists) then
6:             wait two rounds;
7:         else
8:             wait one round;
9:             if ((v_{i,j}, v_{i+1,j}) exists) then
10:                wait one round;
11:            else
12:                move to v_{i−1,j};
13:    else
14:        if ((v_{i,j}, v_{i−1,j}) exists) then
15:            move to v_{i,j};
16:            wait one round;
17:        else
18:            move to v_{i+1,j};
19:            if the current node is v_{i−1,j} then
20:                wait one round;
21:            else if ((v_{i,j}, v_{i+1,j}) exists) then
22:                move to v_{i,j};
23:            else
24:                move to v_{i−1,j};
```

$v_{i-1,j}$ at the end of MEETINGV(i, j).

Proof. Since we can see that the second part holds as explained above, we only show the first part. The only possibility of exiting from $v_{i,j}$ and $v_{i-1,j}$ is at line 18 (in other parts, agents only move through $(v_{i,j}, v_{i-1,j})$). If an agent moves to $v_{i+1,j}$ at line 18, it always succeeds to move $v_{i,j}$

or $v_{i-1,j}$ at line 22 or 24. Thus, the first part holds. □

Exploration algorithm

An exploration algorithm by three agents for the $3 \times \mu$ dynamic torus is presented in Algorithm 4.11.

Algorithm 4.11 aims to explore all the row rings or all the column rings. During the for-loop at lines 4–12, the three agents succeed to explore R_i or C_j (we will prove this in the proof of Theorem 4.5.4). Hence, the agents explore R_i or all the column rings during the for-loop at lines 3–13. Therefore, the agents explore all the row rings or all the column rings during the for-loop at lines 1–14.

First, agents execute ArrangementV_3(i) to move to R_i or R_{i-1}. In the for-loop at lines 4–12, agents execute ExplorationLeft at line 5 to explore R_i. Then, agents execute ArrangementLeft(j) and ExplorationUp(j) at lines 6 and 7 to explore C_j and execute ArrangementV_3(i) and ExplorationLeft at lines 8 and 9 to explore R_i. Finally, agents execute ArrangementLeft(ℓ) and MeetingV(i, ℓ) at lines 10 and 11 to meet another agent.

By repeating this for $0 \leq \ell \leq \mu^2 - \mu$, agents succeed to explore C_j or R_i and, by repeating this for $0 \leq j \leq \mu - 1$ and $0 \leq i \leq 2$, agents complete exploration of the dynamic torus. The following theorem holds.

Theorem 4.5.4. *With the link presence detection, a group of three agents explores the* $3 \times \mu$ *dynamic torus by Algorithm 4.11.*

Proof. It suffices to show that C_j or R_i is explored during the for-loop at lines 4–12.

First, it holds that, at the start of each iteration of the for-loop at lines 4–12, agents are in R_i or R_{i-1}. This is because agents execute ArrangementV_3(i) at line 2 before the for-loop and execute ArrangementV_3(i) at line 8 just after ExplorationUp(j) at line 7 which may make agents exit from R_i or R_{i-1} (MeetingV(i, ℓ) does not make agents in R_i or R_{i-1} exit from R_i and R_{i-1} by Lemma 4.5.6).

In the following, we show the theorem by contradiction. We assume that neither C_j nor R_i are explored during the for-loop at lines 4–12.

Algorithm 4.11 Exploration by three agents for $\nu = 3$ and $\mu \geq 5$

```
1:  for i = 0 to 2 do
2:      ArrangementV_3(i);
3:      for j = 0 to μ − 1 do
4:          for ℓ = 0 to μ^2 − μ do
5:              ExplorationLeft;
6:              ArrangementLeft(j);
7:              ExplorationUp(j);
8:              ArrangementV_3(i);
9:              ExplorationLeft;
10:             ArrangementLeft(ℓ);
11:             MeetingV(i, ℓ);
12:         end for
13:     end for
14: end for
```

First, it holds that one agent, say a, is in R_i and two agents, are in R_{i-1} at the start of each iteration of the for-loop at lines 4–12 ($*$). The reason is that, if there are two or three agents in R_i at this moment, R_i is explored by ExplorationLeft at line 5 and, if there are three agents in R_{i-1} at this moment, C_j is explored by ArrangementLeft(j) and ExplorationUp(j) at lines 6 and 7.

Next, we show that a does not exist at $v_{i,j}$ at the start of ExplorationUp(j) at line 7, which means a does not move by ExplorationUp(j) at line 7. At the start of ExplorationUp(j) at line 7, one of the two agents in R_{i-1} must exist at $v_{i-1,j}$ by ArrangementLeft(j) at line 6. If a is on $v_{i,j}$ at the start of ExplorationUp(j) at line 7, two agents exist in C_j and C_j is explored by ExplorationUp(j) at line 7, which leads to a contradiction.

Additionally, at the start of MeetingV(i, ℓ) at line 11, a is in R_i and the other two agents are in R_{i-1}. This is because agents move to R_i or R_{i-1} (note that a is already in R_i and does not move) by ArrangementV_3(i) at line 8 and, since R_i is not explored by ExplorationLeft at line 9, there exists only one agent in R_i at the end of ExplorationLeft at line 9.

Then, we show that agents do not meet by MeetingV(i, ℓ) at line 11. By the above claims, a is in R_i and two agents are in R_{i-1} at the start of MeetingV(i, ℓ) at line 11, and by Lemma 4.5.6, meeting of agents leads to a configuration where two agents are in R_i or three agents are in R_{i-1}. This contradicts to the above claim (labeled $(*)$).

Finally, we show that a moves *left* at least once during μ iterations of the for-loop at lines 4–12, which meas that a moves at least $(\mu - 1)$ times during $\mu(\mu - 1)$ iterations of the for-loop at lines 4–12 and, thus, by the claim that a does not move *right*, a explores R_i during the for-loop at lines 4–12. We show that by contradiction. We assume that a does not move *left* during μ iterations of the for-loop at lines 4–12. From the above claims, a does not move from R_i and a does not move *right*. Thus, a stays at a node v_{i,j_a} during μ iterations of the for-loop at lines 4–12. However, one of the two agents in R_{i-1} reaches C_{j_a} by ArrangementLeft(j_a) at line 10 and a meets the agent. It leads to a contradiction. Therefore, a moves *left* at least once during μ iterations of the for-loop at lines 4–12, which means that a explores R_i during the for-loop at lines 4–12. This is a contradiction. □

By Lemma 4.5.1, Theorems 4.5.3 and 4.5.4, the following theorem holds.

Theorem 4.5.5. *With the link presence detection, for $\nu \geq 5$ and $\nu = 3$, $\lceil \nu/2 \rceil + 1$ agents are necessary and sufficient to explore the $\nu \times \mu$ dynamic torus.*

4.6 Concluding Remarks

We considered group exploration of the dynamic torus consisting 1-interval connected rings. We proposed exploration algorithms with termination and we showed that the link presence detection has a considerable influence on the number of agents required to explore the dynamic torus. Specifically, we showed that, without the link presence detection, $\nu + 1$ agents are necessary and sufficient to explore and, with the link presence detection, $\lceil \nu/2 \rceil + 1$ agents are necessary and sufficient when $\nu \neq 4$ and $\lceil \nu/2 \rceil + 2$ agents are necessary and sufficient when $\nu = 4$ to explore the dynamic torus.

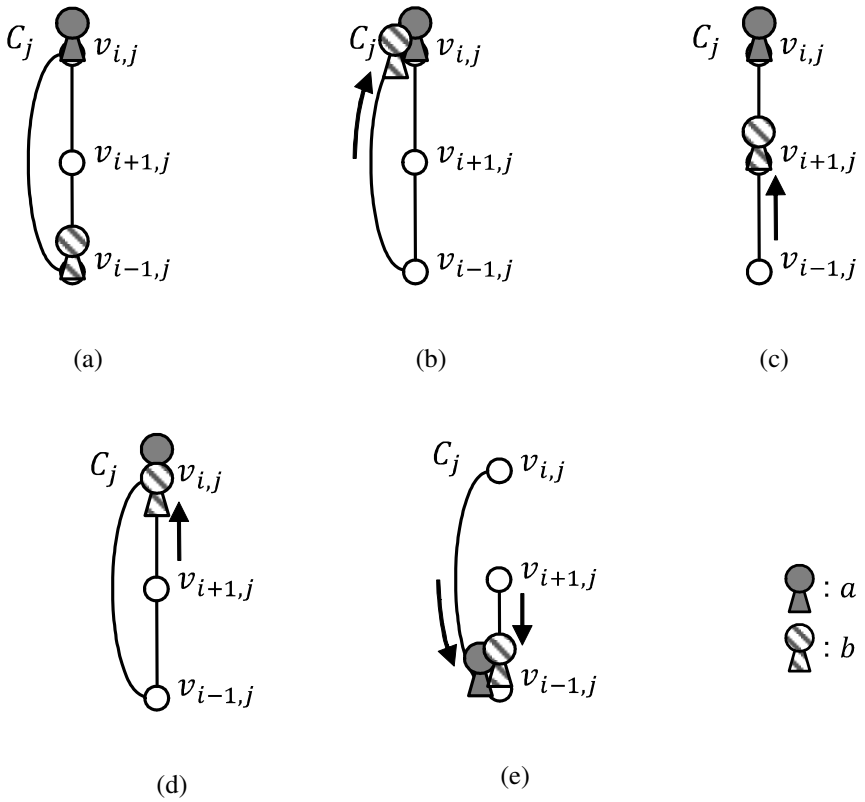

Figure 4.8: (a) Configuration at the start of MEETINGV(i, j). (b) Move of agents when $(v_{i,j}, v_{i-1,j})$ exists in the first round. (c) Move of agents when $(v_{i,j}, v_{i-1,j})$ does not exist in the first round. (d) The move following the move of (c) when $(v_{i,j}, v_{i+1,j})$ exists in the second round. (e) The move following the move of (c) when $(v_{i,j}, v_{i+1,j})$ does not exists in the second round.

Chapter 5

Exploration of Dynamic Rings with (H, S) view

5.1 Introduction

In this chapter, we consider the exploration of dynamic networks by a single agent with partial information about network changes, i.e., a *view*. We focus on 1-interval connected rings as dynamic networks in this chapter.

We assume that the single agent has partial information called the (H, S) view by which it always knows whether or not each of the links within H hops is available in each of the next S time steps. In this setting, we show that $H + S \geq n$ and $S \geq \lceil n/2 \rceil$ (n is the size of the network) are necessary and sufficient conditions to explore 1-interval connected rings. Moreover, we investigate the upper and lower bounds of the exploration time. It is proven that the exploration time is $O(n^2)$ for $\lceil n/2 \rceil \leq S < 2H' - 1$, $O(n^2/H + nH)$ for $S \geq \max(\lceil n/2 \rceil, 2H' - 1)$, $O(n^2/H + n \log H)$ for $S \geq n - 1$, and $\Omega(n^2/H)$ for any S where $H' = \min(H, \lfloor n/2 \rfloor)$.

The results are summarized in Table 5.1

Table 5.1: Upper and lower bounds of the exploration time on 1-interval connected rings.

H and S	Upper bound	Lower bound
$H + S < n$ or $S < \lceil n/2 \rceil$	The exploration is **impossible**.	
$H + S \geq n$ and $\lceil n/2 \rceil \leq S < 2H' - 1$	$O(n^2)$	$\Omega(n^2/H)$
$H + S \geq n$ and $\max(\lceil n/2 \rceil, 2H' - 1) \leq S < n - 1$	$O(n^2/H + nH)$	
$n - 1 \leq S$	$O(n^2/H + n \log H)$	

5.2 Preliminary

5.2.1 Network

In this chapter, we consider the exploration with termination on dynamic networks whose foot print is a ring i.e., $V = \{v_0, v_1, \ldots, v_{n-1}\}$ is a set of n nodes and $E = \{e_0, e_1, \ldots, e_{n-1}\}$ is a set of n links such that $e_i = (v_i, v_{i+1 \bmod n})$. The nodes of the network are anonymous. For simplicity, we omit mod n in the following. In this chapter, a time unit is called a *step*. We assume that $\mathcal{G}$ is 1-*interval connected*, i.e., in each step t, a network is connected. In other words, in each step t, at most one link is missing.

We say the ascending (resp., descending) order of node indices is the right (resp., left) direction. Each port of e_i has a globally consistent label at v_i and v_{i+1} which gives an agent on the ring a global direction (the right direction at v_i and the left direction at v_{i+1}) of the ring. Given a connected component $V' \subsetneq V$, the *right* (resp., *left*) *extremity* of V' is the node $v_i \in V'$ such that $v_{i+1} \notin V'$ (resp., $v_{i-1} \notin V'$). If $|V'| = 1$, the unique node in V' is both the right extremity and the left extremity of V'.

5.2.2 Agents

In the network, a single agent A is operational (i.e., $k = 1$). Agent A knows the network size n. Agent A have a view showing which link is missing within H hops from the current node and within S steps in the future including the current step. The view is called the (H, S) view. Formally speaking, for $\lceil n/2 \rceil \geq H \geq 1$ and $S \geq 1$, A gets the (H, S) view $\beta_{H,S}(i, s) = \{(e_j, t, \rho(e_j, t)) \mid i - H \leq j \leq i + H - 1, s \leq t \leq s + S - 1\}$ when A exists on v_i at step s. For example, when $H = 2$, $S = 2$, and A exists on v_0 at step 5, A can see $\beta_{2,2}(0, 5) = \{(e_1, 5, 0), (e_0, 5, 1), (e_{n-1}, 5, 1), (e_{n-2}, 5, 1), (e_1, 6, 1), (e_0, 6, 0), (e_{n-1}, 6, 1), (e_{n-2}, 6, 1)\}$.

We say that A reaches a node at the t-th step when A visits the node at the end of the $(t-1)$-th step and that A *explores* a node v at the t-th step if v is visited by the $(t-1)$-th step and A reaches v at the t-th step. The set of explored (resp., unexplored) nodes at the start of the t-th step is denoted by V^t (resp., $\overline{V^t}$). Without loss of generality, we assume A starts the exploration from v_0.

In the following, we use "to move to right (resp., left)" instead of "to move in the right (resp., left) direction" for simplicity.

5.3 Impossibility Result

We show an impossibility result in this section. Specifically, we show that the exploration is impossible when $H + S < n$ or $S < \lceil n/2 \rceil$ holds.

Lemma 5.3.1. *If $H + S < n$ or $S < \lceil n/2 \rceil$, a deterministic single agent with the (H, S) view cannot explore 1-interval connected rings.*

Proof. We first consider the condition $S < \lceil n/2 \rceil$. We assume $H = \lceil n/2 \rceil$. It suffices to show that the exploration is impossible when $S = \lceil n/2 \rceil - 1$. We assume for contradiction, that there is an algorithm by which A can explore any ring under any link scheduling when $S = \lceil n/2 \rceil - 1$. Since A can explore the ring, A starting from v_0 eventually reaches v_{n-1} (no matter whether the exploration is completed or not).

The adversary decides a link scheduling so that e_{n-1} (resp., e_{n-2}) is missing when A exists on v_0 (resp., v_{n-2}). The adversary first keeps showing a link scheduling where e_{n-1} is kept deleted for S steps from the current step until A moves to $v_{n-\lceil n/2 \rceil}$. If A does not move to $v_{n-\lceil n/2 \rceil}$ and

stays v_i for $0 \leq i < n - \lceil n/2 \rceil$, e_{n-1} is kept deleted and A cannot reach v_{n-1} (A must pass through e_{n-1} or $e_{n-\lceil n/2 \rceil - 1}$ to reach v_{n-1} from v_0), which is a contradiction. Thus, A eventually reaches $v_{n-\lceil n/2 \rceil}$ at some step, say t.

Then, the adversary deletes e_{n-2} from the $(t+S-1)$-th step (the $(t+\lceil n/2 \rceil - 2)$-th step) until A moves to $v_{n-\lceil n/2 \rceil - 1}$. By the scheduling, since A reaches v_{n-2} at earliest at the $(t+n-2-(n-\lceil n/2 \rceil))$-th step (the $(t + \lceil n/2 \rceil - 2)$-th step) from $v_{n-\lceil n/2 \rceil}$, e_{n-2} starts to disappear when (or before) A reaches v_{n-2} and keeps disappearing unless A moves to $v_{n-\lceil n/2 \rceil - 1}$. Thus, if A does not move to $v_{n-\lceil n/2 \rceil - 1}$, A cannot reaches v_{n-1}. This is a contradiction.

This means that A moves to $v_{n-\lceil n/2 \rceil - 1}$ after the t-th step. However, by the similar way, the adversary can prevent A from reaching v_{n-1}. This is a contradiction. Hence, when $S < \lceil n/2 \rceil$, a single agent cannot explore 1-interval connected rings.

Secondly, we consider the condition $H + S < n$ and $S \geq \lceil n/2 \rceil$. It is sufficient to show that A cannot explore the ring when $S = n - H - 1$ for $1 \leq H \leq \lfloor n/2 \rfloor - 1$ since $H < \lfloor n/2 \rfloor$ from the conditions. Again, we assume for contradiction, that there is an algorithm by which A can explore any ring under any link scheduling. Since A can explore the ring, A starting from v_0 eventually reaches v_{n-1} (no matter whether the exploration is completed or not).

The adversary first keeps showing a link scheduling where e_{n-1} is kept deleted for S steps from the current step until A moves to v_H. If A does not move to v_H and stays at v_i for $0 \leq i \leq H - 1$, e_{n-1} is kept deleted and A cannot reach v_{n-1}, which is a contradiction. Thus, A eventually reaches v_H at some step, say t. After step t, depending on whether A reaches v_{H-1} before v_{n-H-1} or not, the missing link is decided (Figure 5.1). Note that since $H \leq \lfloor n/2 \rfloor - 1$, $(n - H - 1) - (H - 1) \geq 2$ and there exists a node v_i such that $H \leq i \leq n - H - 2$. Moreover, A can see neither e_{n-1} nor e_{n-2} in its view when existing at v_i for $H \leq i \leq n - H - 2$.

If A reaches v_{H-1} before v_{n-H-1}, the adversary keeps deleting e_{n-1}. By the link scheduling, unless A decides to reach v_{n-H-1} from v_H, e_{n-1} is kept deleted and A cannot reach v_{n-1}, which is a contradiction. This means that A eventually reaches v_{n-H-1}. Let t' be the last step before A reaches v_{n-H-1} such that A exists at v_{H-1} at the start of t'.

When A leaves v_{H-1} at the t'-th step, the adversary makes a scheduling so that e_{n-2} starts and keeps disappearing from the $(t' + n - H - 1)$-th step until A comes back to v_{n-H-2}. This does not conflict with the link scheduling in the past view of A since at the t'-th step, e_{n-1} is scheduled to

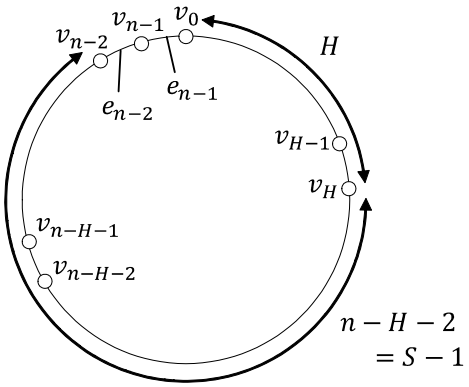

Figure 5.1: Illustrating the proof of Theorem 5.1 for the case of $H + S < n$ and $S \geq \lceil n/2 \rceil$.

be deleted for the next $S = n - H - 1$ steps and for the next $n - H - 1 - x$ steps at the $(t' + x)$-th step.

Since it takes at least $n - H - 2$ steps to reach v_{n-2} from v_H, A reaches v_{n-2} at earliest at the $(t' + n - H - 1)$-th step. However, at the $(t' + n - H - 1)$-th step, e_{n-2} is missing and the adversary keeps deleting e_{n-2} until A reaches v_{n-H-2}. Then, A cannot reach v_{n-1} unless moving to v_{n-H-2}. However, by the similar way, the adversary can prevent A from reaching v_{n-1}. This is a contradiction. Hence, when $H + S < n$ or $S < \lceil n/2 \rceil$, a single agent cannot explore 1-interval connected rings. □

5.4 Possibility Result and Upper Bounds of Exploration Time

In this section, we prove the exploration is possible when $H + S \geq n$ and $S \geq \lceil n/2 \rceil$ by giving an exploration algorithm by a single agent. The algorithm also gives upper bounds of the exploration time, $O(n^2/H + nH)$ if $2H' - 1 \leq S$ or otherwise $O(n^2)$. Note that $S \geq H$ since $S \geq \lceil n/2 \rceil$ and $H \leq \lceil n/2 \rceil$.

We first introduce two operations ExpH(t, v_i) and ExpOne(t, v_i) that are used as building blocks to construct the exploration algorithm.

In the algorithms, *Extremity*(t, v) is a function which returns *right* if v is the right extremity of V^t, *left* if v is the left extremity, or otherwise *nil*. Variable *dir* is used to store the direction and $\overline{dir}$ denotes the other direction (e.g., if *dir* is *right*, $\overline{dir}$ is *left*).

ExpH. ExpH(t, v_i) described in Algorithm 5.1 is an algorithm by which A explores H' nodes when A starts ExpH(t, v_i) from v_i at the t-th step under the assumption that v_i is the right or left extremity of V^t and $2H' + |V^t| - 1 \leq \min(S + 1, n)$. Note that in the following, when A executes ExpH(t, v_i), A is always on the right or left extremity of V^t.

Algorithm 5.1 ExpH(t, v_i)

1: $dir \leftarrow Extremity(t, v_i)$
2: **if** A can move H' hops to dir by the $(t+2H'+|V^t|-2)$-th step **then**
3: Move H' hops to dir
4: **else**
5: Move $|V^t| - 1 + H'$ hops to $\overline{dir}$
6: Wait until the $(t + 2H' + |V^t| - 2)$-th step

When starting the algorithm, A first sees if v_i is the right extremity or the left one and stores *right* if v_i is the right extremity or otherwise *left* in *dir*. If A can move H' hops to *dir* by the $(t + 2H' + |V^t| - 2)$-th step according to the view, A does so (Figure 5.2b). Otherwise, A moves $|V^t| - 1 + H'$ hops to $\overline{dir}$ (Figure 5.2c). Notice that A can decide this condition because $H' \leq H$ and $2H' + |V^t| - 2 \leq S$.

Lemma 5.4.1. *Suppose that at the t-th step, A exists at the right or left extremity, say v_i, of V^t and starts ExpH(t, v_i). If $2H' + |V^t| - 1 \leq \min(S + 1, n)$, A explores H' nodes by the t'-th step (the end of ExpH(t, v_i)) and exists on the right or left extremity of $V^{t'}$ at the t'-th step where $t' = t + 2H' + |V^t| - 2$.*

Proof. Without loss of generality, we assume v_i is the right extremity of V^t. Let $m = |V^t|$, $E_r = \{e_i, e_{i+1}, \ldots, e_{i+H'-2}, e_{i+H'-1}\}$, and $E_l = \{e_{i-H'-m+1}, e_{i-H'-m+2}, \ldots, e_{i-2}, e_{i-1}\}$. Note that since $|E_r| + |E_l| = 2H' + m - 1$ and $2H' + m - 1 \leq n$, $E_r \cap E_l = \emptyset$.

Now, consider the move of A. Since $2H' + m - 1 \leq S + 1$, i.e, $2H' + m - 2 \leq S$, A can see whether it can move H' hops to right by the $(t + 2H' + m - 2)$-th step or not.

If A can, A moves H' hops to right and thus the lemma holds.

Otherwise, A can move at most $H' - 1$ hops to right by the $(t + 2H' + m - 2)$-th step, which means during the $2H' + m - 2$ steps, there exists at least $2H' + m - 2 - (H' - 1) = H' + m - 1$

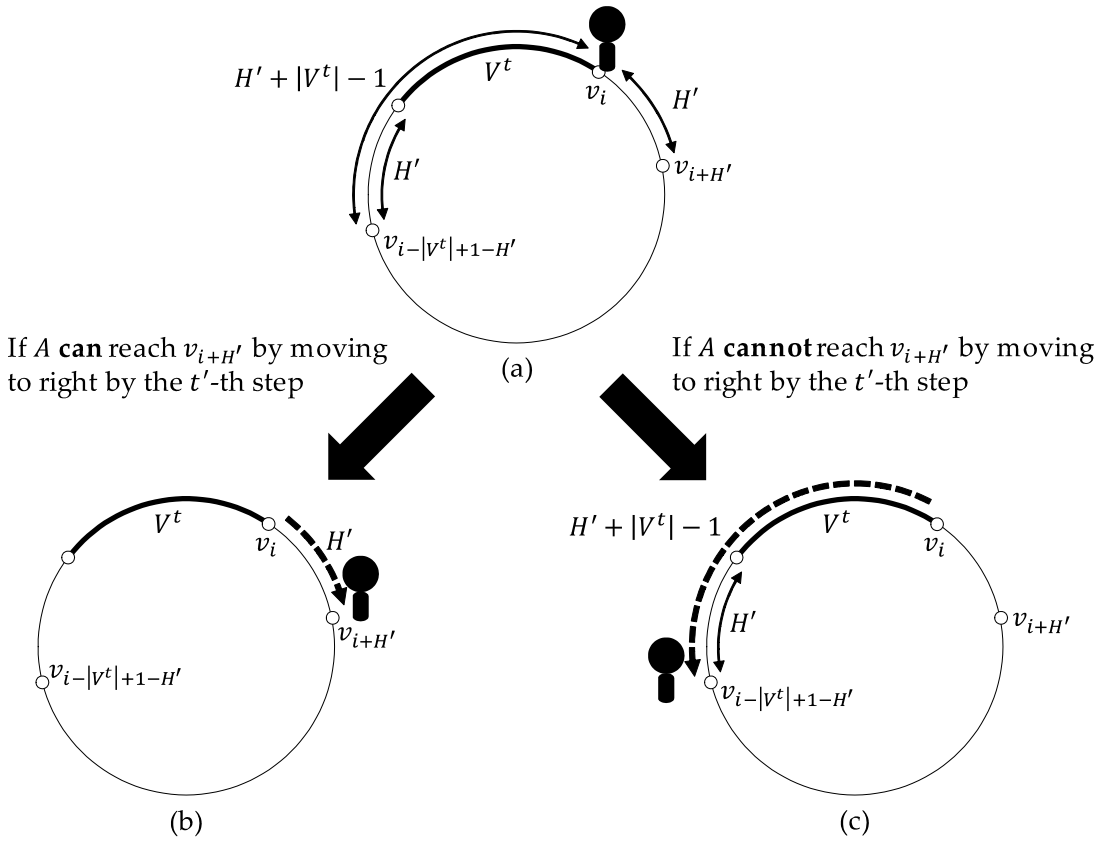

Figure 5.2: The moves of A by ExpH(t, v_i) where $t' = t + 2H' + |V^t| - 2$ in the case where v_i is the right extremity of V^t. (**a**) At the start of ExpH(t, v_i), A exists on v_i. (**b**) If A can reach $v_{i'+H}$ by moving to right by the t'-th step, A moves to right and reaches $v_{i+H'}$ by the t'-th step. (**c**) Otherwise, A moves to left and reaches $v_{i-|V^t|+1-H'}$ by the t'-th step.

steps at each of which one of the links in E_r is missing. Since at most one link is missing at each step and $E_r \cap E_l = \emptyset$, every link in E_l exists at each of the $H' + m - 1$ steps. Thus, A succeeds to move $H' + m - 1$ hops to left and the lemma holds. □

ExpOne. ExpOne(t, v_i) described in Algorithm 5.2 is an algorithm by which A explores at least one node or completes the exploration when A starts ExpOne(t, v_i) from v_i at the t-th step under the assumption that v_i is the right or left extremity of V^t. Note that in the following, when A executes ExpOne(t, v_i), A is always on the right or left extremity of V^t.

Algorithm 5.2 ExpOne(t, v_i)

1: $dir \leftarrow Extremity(t, v_i)$
2: **if** dir is *right* **then**
3: $\quad i' \leftarrow i+1, i'' \leftarrow i$
4: **else**
5: $\quad i' \leftarrow i-1, i'' \leftarrow i-1$
6: $d \leftarrow 0$
7: $S' \leftarrow \max(n-H, \lceil n/2 \rceil)$
8: **while** $(d < H)$ **do**
9: $\quad$ **if** $e_{i''}$ is always missing until the $(t+d+S'-1)$-th step **then**
10: $\quad\quad$ Move one hop to $\overline{dir}$
11: $\quad\quad$ $d \leftarrow d+1$
12: $\quad$ **else**
13: $\quad\quad$ Move d hops to dir (reach v_i)
14: $\quad\quad$ Wait for $e_{i''}$ to appear and pass through $e_{i''}$ as soon as it appears
15: $\quad\quad$ Exit from the while loop
16: **if** $(d \geq H)$ **then**
17: $\quad$ Move $n-H-1$ hops to $\overline{dir}$ (reach $v_{i'}$)
18: Wait until the $(t+n)$-th step

When starting the algorithm, A first sees if v_i is the right extremity or the left one and stores the direction in dir. Variables i' and i'' are used to remember the dir neighbor of v_i and the dir incident edge of v_i respectively, e.g., $i' = i+1$ (resp., $i' = i-1$) if dir is *right* (resp., *left*). Then, A stores $\max(n-H, \lceil n/2 \rceil)$ to S' which is not larger than S and is used instead of S in the algorithm.

After that, if $e_{i''}$ appears by the $(t+S'-1)$-th step, A waits at v_i until $e_{i''}$ appears and moves to $v_{i'}$ when $e_{i''}$ appears. Otherwise, for each $0 \leq d \leq H-1$, A moves one hop to $\overline{dir}$ at the $(t+d)$-th step if $e_{i''}$ is missing at the $(t+S'-1+d)$-th step in its view (Figure 5.3a). If A sees $e_{i''}$ appear at the $(t+S'-1+d)$-th step in its view at the $(t+d)$-th step, then A starts to move dir from the $(t+d)$-th step, returns to v_i, waits at v_i until $e_{i''}$ appears, and reaches $v_{i'}$ through $e_{i''}$ (Figure 5.3b). When d reaches H, i.e., A moves H hops to $\overline{dir}$ and $e_{i''}$ is no longer included in the view

of A, A starts to keep moving to $\overline{dir}$ until reaching $v_{i'}$ and the exploration finishes when reaching $v_{i'}$ (Figure 5.3c).

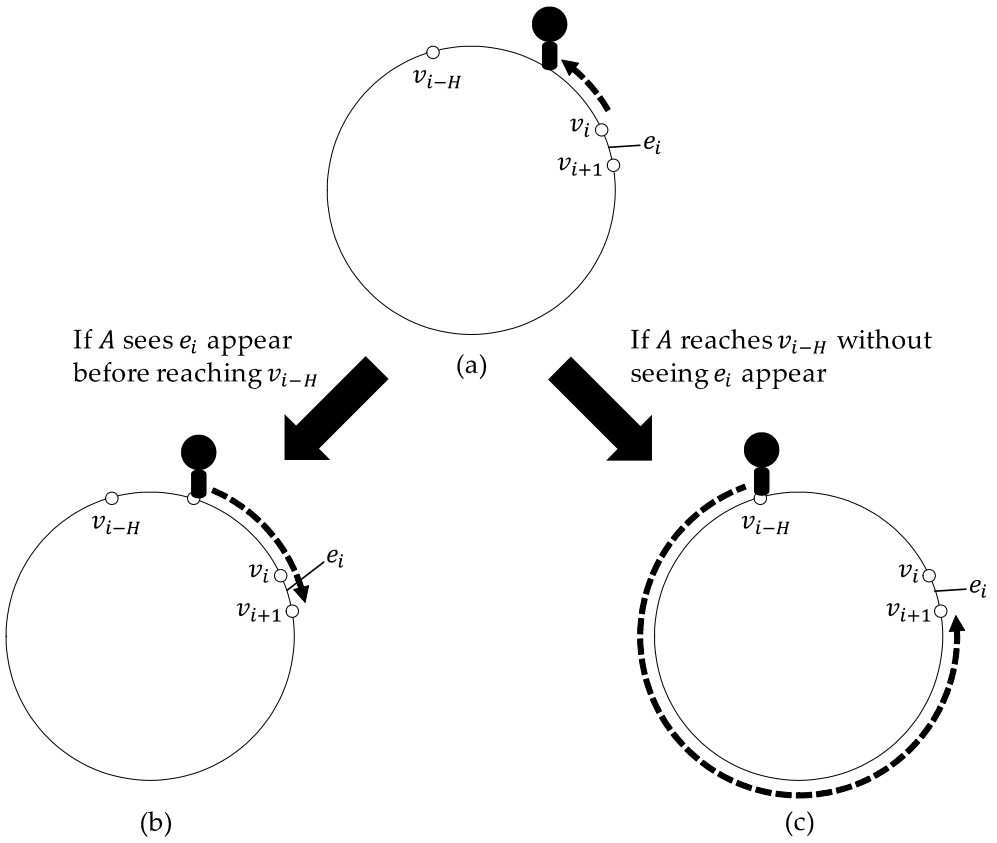

Figure 5.3: The moves of A by ExpOne(t, v_i) in the case where v_i is the right extremity. (**a**) Unless A sees e_i appear, A moves to left. (**b**) If A sees e_i appear before reaching v_{i-H}, A starts to move to right and reaches v_{i+1}. (**c**) If A reaches v_{i-H} without seeing e_i appear, A keeps moving to left until reaching v_{i+1} and finishes the exploration.

Lemma 5.4.2. *Suppose that at the t-th step, A exists at the right (resp., left) extremity, say v_i, of V^t and starts ExpOne(t, v_i). Then, A completes the exploration or reaches v_{i+1} (resp., v_{i-1}) by the $(t+n)$-th step (the end of ExpOne(t, v_i)). In addition to that, A exists on the right or left extremity of V^{t+n} at the $(t+n)$-th step when the exploration has not been completed.*

Proof. Without loss of generality, we assume v_i is the right extremity of V^t. As in Algorithm 5.2,

let $S' = \max(n-H, \lceil n/2 \rceil)$. We first show the lemma for the case e_i appears by the $(t+d+S'-1)$-th step in A's view at the $(t+d)$-th step for $0 \leq d \leq H-1$.

For $d = 0$, A can clearly reach v_{i+1} by the $(t+S'-1)$-th step.

For $1 \leq d \leq H-1$, when A sees e_i appear for the first time at the $(t+d+S'-1)$-th step in its view at the $(t+d)$-th step, e_i must appear at the $(t+d+S'-1)$-th step and be missing at the t'-th step for $t+d \leq t' \leq t+d+S'-2$ by the construction. This means that all the other links than e_i are present at the t'-th step ($t+d \leq t' \leq t+d+S'-2$), and thus A can move for $S'-1$ steps from v_{i-d} to right without interference by missing links until reaching v_i.

Since $d \leq H-1$ and $H \leq S'$, A always reaches v_i by the $(t+d+S'-1)$-th step at which e_i appears. Then, A reaches v_{i+1} as soon as e_i appears. Since A moves at most $H-1$ hops to left, e_i appears S' steps after A starts to move to right, and $H-1+S' \leq n$ from $S' = \max(n-H, \lceil n/2 \rceil)$, A reaches v_{i+1} through e_i by the $(t+n)$-th step.

We then show for the other case, i.e., A reaches v_{i-H} at the $(t+H)$-th step. When this happens, e_i must be deleted for at least $S'-1$ steps from the $(t+H)$-th step and all the other links than e_i are present in the $S'-1$ steps. Thus, A can move for $S'-1 \geq n-H-1$ steps from v_{i-H} to left without interference by missing links until reaching v_{i+1} since $S' = \max(n-H, \lceil n/2 \rceil)$. Since $H+n-H-1 = n-1$, A reaches v_{i+1} after $n-H-1$ steps, i.e., at the $(t+n-1)$-th step, and the exploration is completed at the same time. □

Exploration algorithm. Algorithm 5.3 describes the exploration algorithm. Let $S'' = \min(S, n-1)$. The algorithm repeats ExpH(t, v_i) for $\lfloor (S''+1-H')/H' \rfloor$ times (lines 2-6) and ExpOne(t, v_i) for $n - H'\lfloor (S''+1-H')/H' \rfloor - 1$ times (lines 7-13). We call the part repeating ExpH(t, v_i) (lines 2-6) *the first part* and the part repeating ExpOne(t, v_i) *the second part* (lines 7-13). In the first part, $H'\lfloor (S''+1-H')/H' \rfloor + 1$ nodes are explored and, in the second part, the remaining $n - H'\lfloor (S''+1-H')/H' \rfloor - 1$ nodes are explored.

Theorem 5.4.1. *For $H+S \geq n$ and $S \geq \lceil n/2 \rceil$, the exploration time of 1-interval connected rings by a single agent with the (H, S) view is upper-bounded by $O(n^2/H + nH)$ if $2H'-1 \leq S$ or otherwise it is upper-bounded by $O(n^2)$.*

Proof. It suffices to show that A with the (H, S) view completes exploration within $O(n^2/H+nH)$ steps if $2H'-1 \leq S$ or otherwise $O(n^2)$ steps by executing Algorithm 5.3 when $H+S \geq n$ and

Algorithm 5.3 Exploration algorithm for $H + S \geq n$

```
1: S'' ← min(S, n − 1)
2: p ← 1 //starting the first part
3: while (p ≤ ⌊(S'' + 1 − H')/H'⌋) do
4:    Let t be the current step and v_i be the current node
5:    ExpH(t, v_i)
6:    p ← p + 1
7: p ← 1 //starting the second part
8: while (p ≤ n−H'·⌊(S''+1−H')/H'⌋−1) do
9:    Let t be the current step and v_i be the current node
10:   ExpOne(t, v_i)
11:   if Exploration is completed then
12:      Exit from the while loop
13:   p ← p + 1
```

$S \geq \lceil n/2 \rceil$.

We first consider the case where $2H' - 1 \leq S$. In this case, since $\lfloor (S'' + 1 - H')/H' \rfloor \geq 1$, the first part is executed at least once. Consider the first part. Let t_p be the step when A starts the p-th ExpH(t, v_i).

We show by induction that for $1 \leq p \leq \lfloor (S'' + 1 - H')/H' \rfloor$, $|V^{t_p}| = (p-1)H' + 1$ and A explores H' nodes by ExpH(t_p, v_i).

For the base case, i.e., $p = 1$, $|V^{t_1}|$ is clearly $1 = (p-1)H' + 1$. This leads to that $2H' + |V^t| - 1 = 2H' \leq \min(n, S + 1)$. Then, by Lemma 5.4.1, A explores H' nodes by ExpH(t_1, v_i).

Now, for $k \leq \lfloor (S'' + 1 - H')/H' \rfloor - 1$, assume that $|V^{t_k}| = (k-1)H' + 1$ and A explores H' nodes by ExpH(t_k, v_i). Then, clearly $|V^{t_{k+1}}| = (k-1)H' + 1 + H' = kH' + 1$. Since $k \leq \lfloor (S'' + 1 - H')/H' \rfloor - 1$, $2H' + |V^{t_{k+1}}| - 1 < n$ and $2H' + |V^{t_{k+1}}| - 1 < S + 1$. Thus, A explores H' nodes by ExpH(t_{k+1}, v_i).

By Lemma 5.4.1, $S'' = O(n)$, and $H' = \Theta(H)$, the exploration time of the first part is

$$\sum_{p=1}^{\lfloor (S''+1-H')/H' \rfloor} (2H' + |V^{t_p}| - 2) = \sum_{p=1}^{\lfloor (S''+1-H')/H' \rfloor} ((p+1)H' - 1) = O(n^2/H).$$

We then consider the second part. By Lemma 5.4.1, A exists at the right or left extremity of V^t and $|\overline{V^t}| = n - H'\lfloor (S'' + 1 - H')/H' \rfloor - 1 = O(H)$ at the start of the second part. Thus, since A explores one node within n steps by Lemma 5.4.2, the exploration time of the second part is $O(nH)$.

As a result, the exploration time of Algorithm 5.3 is $O(n^2/H + nH)$ when $2H' - 1 \leq S$.

When $2H' - 1 > S$, the first part is never executed and then the number of remaining nodes at the start of the second part is $n - 1$. Thus, in this case, the exploration time of Algorithm 5.3 is $O(n^2)$. □

From Lemma 5.1 and Theorem 5.4.1, the following theorem holds.

Theorem 5.4.2. *If and only if $H + S \geq n$ and $S \geq \lceil n/2 \rceil$, a single agent with the (H, S) view can explore of 1-interval connected rings within finite time steps.*

5.5 Upper Bound of Exploration Time for $S \geq N - 1$

In this section, we consider the upper bound of the exploration time when $S \geq n - 1$. We show that the upper bound of the exploration time is reduced to $O(n^2/H + n \log H)$ in this case by giving an exploration algorithm.

We first introduce a new operation ExpHalf(t, v_i) that is used as a building block to construct the exploration algorithm.

ExpHalf. ExpHalf(t, v_i) described in Algorithm 5.4 is an algorithm by which A explores $\lceil |\overline{V^t}|/2 \rceil$ nodes when A starts ExpHalf(t, v_i) from v_i at the t-th step under the assumption that v_i is the right or left extremity of V^t, $|\overline{V^t}| \leq 2H$, and $S \geq n - 1$. Note that in the following, when A executes ExpHalf(t, v_i), A is always on the right or left extremity of V^t.

When starting the algorithm, A first sees if v_i is the right extremity or the left one and stores *right* if v_i is the right extremity or otherwise *left* in *dir*. If A can move $\lceil |\overline{V^t}|/2 \rceil$ hops to *dir* by the

Algorithm 5.4 ExpHalf(t, v_i)

1: $dir \leftarrow Extremity(t, v_i)$
2: **if** A can move $\lceil|\overline{V^t}|/2\rceil$ hops to dir by the $(t+n-1)$-th step **then**
3: Move $\lceil|\overline{V^t}|/2\rceil$ hops to dir
4: **else**
5: Move $n - \lceil|\overline{V^t}|/2\rceil$ hops to $\overline{dir}$
6: Wait until the $(t+n-1)$-th step

$(t+n-1)$-th step according to the view, A does so (Figure 5.4b). Otherwise, A moves $n - |\overline{V^t}|/2$ hops to $\overline{dir}$ (Figure 5.4c).

Lemma 5.5.1. *Suppose that at the t-th step, A exists at the right or left extremity, say v_i, of V^t and starts ExpHalf(t, v_i). If $|\overline{V^t}| \leq 2H$ and $S \geq n-1$, A can explore at least $\lceil\overline{V^t}/2\rceil$ nodes by the t'-th step (the end of ExpHalf(t, v_i)) and exists on the right or left extremity of $V^{t'}$ at the t'-th step where $t' = t+n-1$.*

Proof. Without loss of generality, we assume v_i is the right extremity of V^t. Let $m = |\overline{V^t}|$, $E_r = \{e_i, e_{i+1}, \ldots, e_{i+\lceil m/2\rceil-1}, e_{i+\lceil m/2\rceil}\}$, and $E_l = \{e_{i+\lceil m/2\rceil+1}, e_{i+\lceil m/2\rceil+2}, \ldots, e_{i-1}\}$.

Now, consider the move of A. Since $S \geq n-1$ and $m \leq 2H$, A can see whether it can move $\lceil m/2\rceil$ hops to right by the $(t+n-1)$-th step or not.

If A can move $\lceil m/2\rceil$ hops, A moves $\lceil m/2\rceil$ hops to right and thus the lemma holds.

Otherwise, A can move at most $\lceil m/2\rceil - 1$ hops to right by the $(t+n-1)$-th step, which means during the $n-1$ steps, there exist at least $n-1-(\lceil m/2\rceil - 1) = n - \lceil m/2\rceil$ steps at each of which one of the links in E_r is missing. Since at most one link is missing at each step and $E_r \cap E_l = \emptyset$, every link in E_l exists at each of the $n - \lceil m/2\rceil$ steps. By this and $|E_l| = n - \lceil m/2\rceil$, A succeeds to reach $v_{i+\lceil m/2\rceil}$ by moving to left, which means at least $\lceil m/2\rceil$ nodes are explored. □

Exploration algorithm. Algorithm 5.5 describes the exploration algorithm. The algorithm repeats ExpH(t, v_i) for $\lfloor(n-H')/H'\rfloor$ times (lines 1–5) and ExpHalf(t, v_i) for $\lceil\log(n - H'\lfloor(n-H')/H'\rfloor - 1)\rceil$ times (lines 6–10). We call the part repeating ExpH(t, v_i) (lines 1-5) *the first part* and the part repeating ExpHalf(t, v_i) *the second part* (lines 6–10). In the first part, $H'\lfloor(n-H')/H'\rfloor + 1$

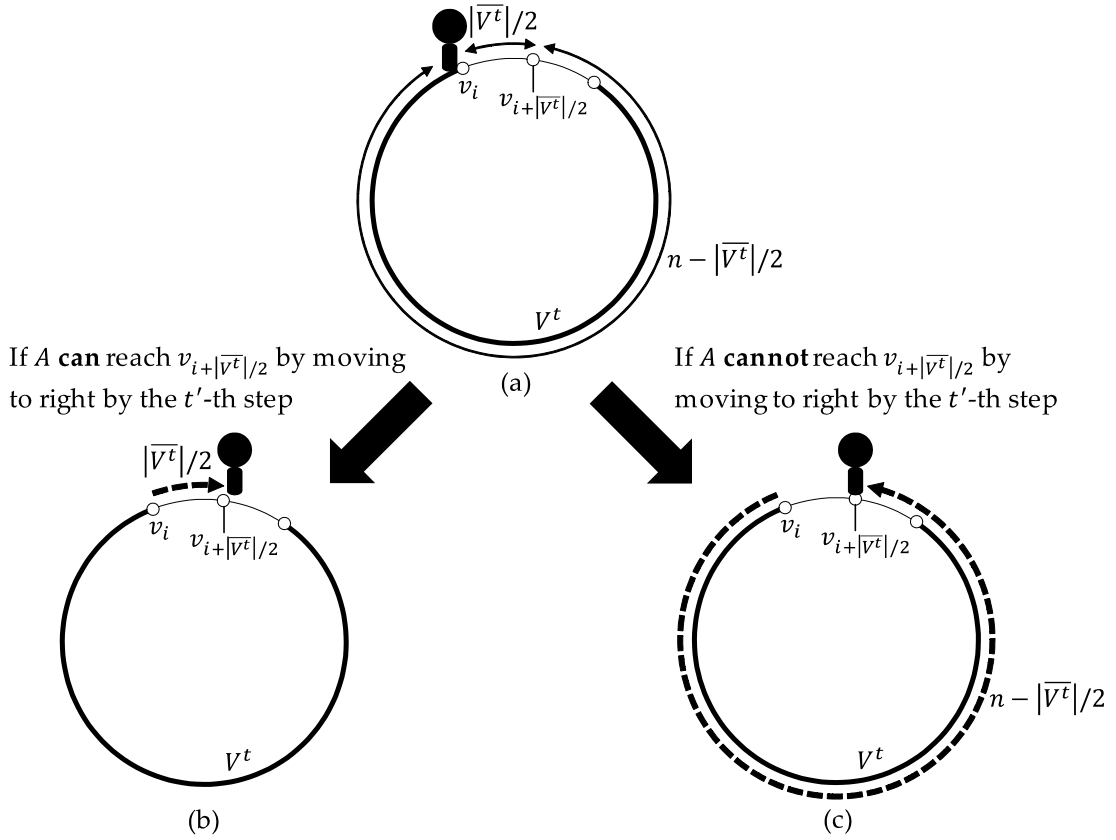

Figure 5.4: The moves of A by ExpHalf(t, v_i) where $t' = t + n - 1$ in the case where v_i is the right extremity of V^t. (**a**) At the start of ExpHalf(t, v_i), A exists on v_i. (**b**) If A can reach $v_{i+|\overline{V^t}|/2}$ by moving to right by the t'-th step, A moves to right and reaches $v_{i+|\overline{V^t}|/2}$ by the t'-th step. (**c**) Otherwise, A moves to left and reaches $v_{i+|\overline{V^t}|/2}$ by the t'-th step.

nodes are explored and, in the second part, the remaining $n - H'\lfloor (n - H')/H' \rfloor - 1$ nodes are explored.

Theorem 5.5.1. *For $S \geq n - 1$, the exploration time of* 1*-interval connected rings by a single agent with the (H, S) view is upper-bounded by $O(n^2/H + n \log H)$.*

Proof. It suffices to show that A completes exploration within $O(n^2/H + n \log H)$ steps by Algorithm 5.5 when $S \geq n - 1$. It is proven that the total exploration time of the first part is $O(n^2/H)$ and that of the second part is $O(n \log H)$.

Algorithm 5.5 Exploration algorithm for $S \geq n - 1$

1: $p \leftarrow 1$ //starting the first part
2: **while** $(p \leq \lfloor (n - H')/H' \rfloor)$ **do**
3: Let t be the current step and v_i be the current node
4: ExpH(t, v_i)
5: $p \leftarrow p + 1$
6: $p \leftarrow 1$ //starting the second part
7: **while** $(p \leq \lceil \log(n - H' \lfloor (n - H')/H' \rfloor - 1) \rceil)$ **do**
8: Let t be the current step and v_i be the current node
9: ExpHalf(t, v_i)
10: $p \leftarrow p + 1$

We first consider the first part. Note that, since $2H' \leq n$, $1 \leq \lfloor (n - H')/H' \rfloor$ and thus the first part is always executed at least once. Let t_p be the step when A starts the p-th ExpH(t, v_i). We can show that for $1 \leq p \leq \lfloor (n - H')/H' \rfloor$, A can explore H' nodes by ExpH(t_p, v_i) by induction and the exploration time of the first part is $O(n^2/H)$ as in the proof of Lemma 5.4.1.

We then consider the second part. By Lemma 5.4.1, A exists at the right or left extremity of V^t and $|\overline{V^t}| = n - H' \lfloor (n - H')/H' \rfloor - 1 \leq 2H'$ at the start of the second part. Thus, since A explores a half of $\overline{V^t}$ within $n - 1$ steps by Lemma 5.5.1, the exploration time of the second part is $O(n \log H)$. As a result, the exploration time of Algorithm 5.3 is $O(n^2/H + n \log H)$. □

5.6 Lower Bound of Exploration Time

A lower bound of the exploration time for any S is given in this section. The following theorem holds.

Theorem 5.6.1. *The exploration time of* 1*-interval connected rings by a single agent with the* (H, S) *view is lower-bounded by* $\Omega(n^2/H)$.

Proof. We first show that, provided that A is at the right or left extremity, say v_i, of V^t at the t-th step where $|V^t| \leq n - 2H + 1$, it takes at least $|V^t| + H - 1$ steps for A to explore H nodes from the circumstance under the following link scheduling: e_{i+H-1} (resp., e_{i-H}) is deleted until the $(t + |V^t| + H - 1)$-th step if v_i is the right (resp., left) extremity of V^t. Without loss of generality, we assume that v_i is the right extremity of V^t in the following. Figure 5.5 depicts the situation.

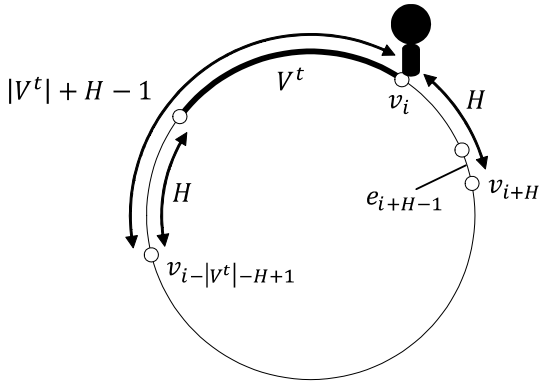

Figure 5.5: The situation where A exists on v_i at the t-th step (v_i is the right extremity of V^t). The adversary deletes e_{i+H-1} until the $(t + |V^t| + H - 1)$-th step in this situation.

Assume for contradiction that A explores the H nodes within $|V^t| + H - 1$ steps under the scheduling. Since e_{i+H-1} is missing until the $(t + |V^t| + H - 1)$-th step, A never reaches v_{i+H}. Therefore, A must explore at least one node on the left side of V^t. This and exploring H nodes take at least $|V^t| + H - 1$ steps; a contradiction.

Now, apply the above claim from the first step repeatedly. When applying the claim for the p-th time, $|V^t| = (p - 1)H + 1$ and then it takes $|V^t| + H - 1 = pH$ steps. Note that we can apply the claim while $(p - 1)H + 1 \leq n - 2H + 1$, i.e., $p \leq \lfloor (n - H)/H \rfloor$. We then derive the lower bound of the exploration time, $\sum_{p=1}^{\lfloor (n-H)/H \rfloor} pH = \Omega(n^2/H)$. □

5.7 Discussion

In this chapter, we studied the exploration problem on dynamic networks with its partial information, where we focused on 1-interval connected rings as a first step. In this section, we discuss what happens when we consider other connectivity and/or general graphs.

When considering 1-interval connected rings, we yields the restriction that at most one link is missing at each step. By this restriction, an agent gets to know that all the links outside its view exist when a link in its view is missing and can make the action plan to visit an unvisited node using the information. It is interesting to investigate such conditions on the space and the time of a view (H and S in this chapter) for more general graphs under some assumptions on the temporal connectivity and/or more general graphs. On the other hand, even under the assumption of 1-interval connectivity and/or the restriction on the number of missing links at each step, an agent cannot necessarily get the whole information of the temporal topology, which may prevent the agent from making the action plan to visit an unvisited node and makes the exploration problem more challenging.

We also conjecture that the space and the time of a view which are necessary and sufficient for an agent to explore depend on temporal diameter. Intuitively, temporal diameter is the maximum duration of the foremost path (the path with the least duration from a node to another node departing at specified time) in a dynamic network (see e.g., Section 4.6 of [23] for a formal definition). The fact that the temporal diameter of a 1-interval connected graph with n nodes is at most $n - 1$ fits a possibility result of this chapter, i.e., $H + S \leq n$. To investigate the relation of temporal distance and the power of a view is one of the intriguing research directions.

5.8 Concluding Remarks

In this chapter, we introduced the (H, S) view which can be used to model some situations where an agent (or robot) can partly see their nearby environment or can predict the near-future changes of the environment. For a single agent with the (H, S) view, we studied the exploration of 1-interval connected rings. We give some fundamental results, i.e., impossibility of the exploration for $H + S < n$ or $S < \lceil n/2 \rceil$, possibility of the exploration for $H + S \geq n$ and $S \geq \lceil n/2 \rceil$, and upper bounds and a lower bound of the exploration time for some cases.

Chapter 6

Fault-Tolerant Simulation of Message-Passing Algorithms by Mobile Agents

6.1 Introduction

In this chapter, we consider simulation of message-passing algorithms in a mobile agent model with k agents where f of them may crash for a given f ($\leq k-1$). Two fault-tolerant algorithms are proposed for the simulation.

Our first algorithm simulates a message-passing algorithm which eventually terminates (e.g. spanning tree construction and coloring), say Z, with $O((m+M)f)$ total agent moves and thus $O(f)$ agent moves per message when $m = O(M)$, where m is the number of links in the network and M is the total number of messages created in the simulated execution of Z. The previous algorithm [22] can tolerate $k-1$ agent crashes but requires $O((m+nM)k)$ total agent moves, where n is the number of nodes in the network. Therefore, our algorithm improves the total number of agent moves for $f = k-1$ and requires a smaller number of total moves if f is smaller, and moreover, our algorithm is asymptotically optimal in terms of the total number of agent moves and the number of agent moves per message.

Our second algorithm simulates a message-passing algorithm which never terminates (e.g.

mutual exclusion and token circulation) with $O(f)$ agent moves per message. As for this algorithm, the number of agent moves per message is asymptotically optimal.

6.2 Preliminary

6.2.1 Network

In this chapter, we consider static networks with arbitrary topology, i.e., all the links are always present.

We consider two different computation models on the network, a mobile agent model for simulating algorithms and a message-passing model for simulated algorithms, which are defined in the following subsections. When a simulated message-passing algorithm requires a node ID, a node has ID (and an agent can use the ID to simulate local computation of the message-passing algorithm).

6.2.2 Mobile agent model

There is a set A of k agents. An agent can use a whiteboard. Each agent a has a unique ID $a.id$ and we assume each ID is represented in $O(\log k)$ bits. Every agent does not know n. Each agent a is initially allocated to some node called a *homebase* of a. We assume $k \leq n$ and homebases of agents are arbitrary.

Each link in the network is FIFO, that is, when agents a_1 and a_2 move from node u to node v in this order, a_1 arrives at v before a_2 unless a_1 crashes during the movement. The system is asynchronous, that is, the time required for an agent to move from a node to its neighbor is finite but unbounded.

We assume that the target model (or the message-passing model) is reliable but the host model (or the mobile agent model) is prone to faults. An agent may crash (or disappear) when it moves through a link, but it never crashes when it is on a node. We assume at most $f \leq k - 1$ agents crash. An agent knows f while it has no knowledge about k. (Note that even when an agent does not know f and k, the number of agent moves can be bounded by k.) After an agent leaves a node, it arrives at the next node eventually unless it crashes during the movement. Once an agent

crashes, it disappears from the network forever. We say an agent is faulty (resp., non-faulty) if it crashes (resp., never crashes) during the execution. Note that agents cannot recognize faulty agents as long as they work correctly.

6.2.3 Message-passing model

In the message-passing model, a node u (not an agent) executes computation. A node v executes the following operations atomically in each step:

1. it receives a message or initiates an algorithm spontaneously,
2. executes local computation and updates its own state, and
3. if necessary, sends messages to its neighbors by using the primitive $SEND(msg, \lambda_v(e_{uv}))$ repeatedly for all destinations (node u can send a message msg to node v by using the primitive $SEND(msg, \lambda_u(e_{uv}))$).

The above actions are executed asynchronously: although 1 and 2 are instantly completed, the required time to complete 3 is finite but unbounded. Every process executes computing and (if any) sends a message only when it receives a message except for the case of spontaneous initiation. There exists at least one spontaneous initiator, which is assigned the special initial state and initiates an algorithm spontaneously (by receiving the null message). Note that, since the set of initiators is unknown in advance, algorithms should work correctly for any set of initiators.

Communication in the message-passing model is reliable, that is, it satisfies the following:

- [A1] Every message sent by node u to its neighbor v is eventually received by v exactly once.
- [A2] A message is received by node u only when it was previously sent to u by neighbor v.

Each link in the network is FIFO, that is, when u sends messages msg_1 and msg_2 to v in this order, v receives msg_1 before msg_2.

6.2.4 Lower bound of move complexity

We conclude this section by giving the lower bound of the move complexity to deliver a message in the above model.

Theorem 6.2.1. *It requires $\Omega(f)$ moves to deliver a message correctly.*

Proof. We show that whether a message being delivered by f or less agents reaches its destination correctly is undecidable. Suppose that message *msg* is delivered by f agents and all the f agents crashes during the delivery. Although the message cannot be reached its destination, the system has no way to distinguish that from the situation where at least one agent is alive and delivering *msg*. This is because the network is asynchronous where the time required to move along a link is unbounded and unpredictable. Thus, the number of agent moves per message is $\Omega(f)$. Therefore, the theorem holds. □

6.3 Simulation of message-passing algorithms with a finite number of messages

In this section, we propose an agent-based simulating algorithm of a message-passing algorithm with a finite number of messages (i.e., an eventually terminating algorithm). We use Z to denote the simulated message-passing algorithm.

6.3.1 The description of a simulating algorithm

Our algorithm consists of two parts, *1)* searching initiators (search part) and *2)* simulating execution of nodes and delivering messages (delivery part). First, we present the search part, i.e., *1)* searching initiators. Each agent starts to search initiators from its homebase by the depth-first search. When it finds an initiator, it starts the delivery part, i.e., *2)* simulating execution of nodes and delivering messages. After completing the delivery part, it resumes the search part to find another initiator. An agent records its searching path of the search part by writing the incoming port with its agent ID in the whiteboard of each visited node so that it can backtrack.

In the search part, an agent backtracks to the previous node when at least one of the following conditions is satisfied.

1. There is no unsearched port at the current node.
2. A cycle is detected in its searching path of the search part.
3. The agent detects that other $f + 1$ agents have already visited the current node during their search part.

Conditions 1 and 2 come from the depth-first search. Condition 3 is introduced to save the total number of agent moves. Our algorithm can tolerate agent crashes by making multiple agents transfer a message, however $f + 1$ agents are enough to transfer a message since at most f agents crash (there is at least one non-faulty agent in the $f + 1$ agents). Thus, an agent backtracks when it detects that other $f + 1$ agents execute the search part. The agent terminates its execution when it completes the search part and returns to its homebase.

Next, we present the delivery part, i.e., *2)* simulating execution of nodes and delivering messages.

An agent starts the simulation when it finds an initiator during the depth-first search of the search part. Note that, by Condition 3 of the search part, at most $f + 1$ agents start simulation at an initiator.

An agent delivers messages successively in the depth-first fashion, that is, if agent a delivers a message to node v and there exists message *msg* to transfer from v to a neighbor of v, a takes *msg* from v and delivers *msg* to its destination node. An agent records its delivering path in the same way as the search part so that it can backtrack.

Since a message is transfered by at most $f + 1$ agents for fault-tolerance, the message may be delivered multiple times. However an agent simulates the action of a node on receipt of a message only when the message is received for the first time so that it is processed only once at the destination.

A message is deleted from a node when an agent which delivered the message returns after delivering the message to its destination. For this purpose, when an agent takes a message from a node to deliver it to a neighbor, the agent stores its ID to *send-member* of the message in the whiteboard of the current node to indicate that the agent is transferring the message. If the agent returns the node and finds its ID in *send-member*, the agent deletes the message corresponding to *send-member* and resets *send-member* of the current node to empty.

In the delivery part, an agent backtracks to the previous node when at least one of the following conditions is satisfied.

1. There is no message to transfer from the current node.
2. A cycle is detected in its delivering path of the delivery part.
3. The current node is locked using the port other than the one the agent arrives through. We describe the locking mechanism later.
4. The current node is locked but the agent is not a *lock-member* agent of the node when the agent backtracks to the node.

Condition 1 realizes message deliveries in the depth-first fashion. Condition 2 is introduced to prevent the delivering path from growing so long, which saves the memory space of nodes. Using only Conditions 1 and 2 may leave an undelivered message as explained later. Thus, Conditions 3 and 4 are introduced to guarantee deliveries of all the messages.

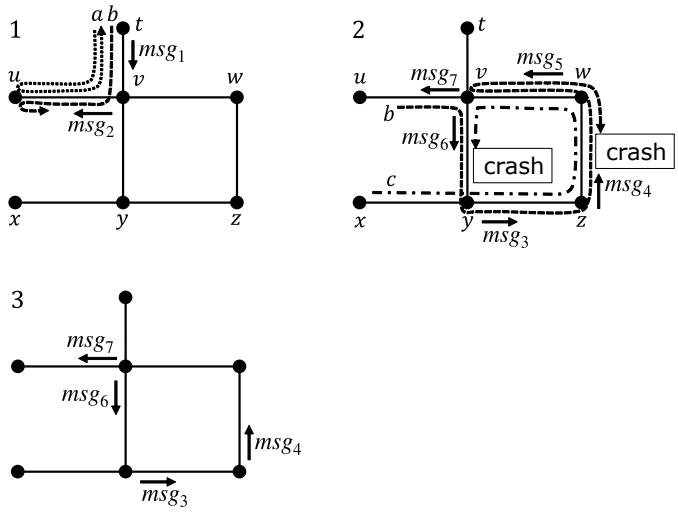

Figure 6.1: An example where Conditions 1 and 2 leave an undelivered message.

Consider the case of Figure 6.1. First, agent a arrives at t and delivers messages msg_1 and msg_2 from t to v and from v to u respectively, and agent b follows a and arrives at u. Then, a

backtracks to t and deletes the messages msg_1 and msg_2 at t and v while b is still in transit in link e_{uv} (Figure 6.1-1). Second, an agent c arrives at y from x and delivers messages msg_3, msg_4 and msg_5 from y to z, from z to w and from w to v. Then, c generates two messages msg_6 and msg_7 at v, one is to y and the other is to u in this order, and crashes when it is transferring msg_6 to y. After that, agent b arrives at v from u and delivers messages msg_6, msg_3, msg_4 and msg_5 from v to y, from y to z, from z to w and from w to v. Then, b detects a cycle at v, backtracks to w, and deletes the message msg_5. Then, while b backtracks from w to z, b crashes (Figure 6.1-2). Here, node v has message msg_7 to transfer to u but it is possible that no agent arrives at v after the situation since there is no undelivered message toward v (Figure 6.1-3). Thus, in this case, message msg_7 from v to u may be left undelivered forever.

A possible way to avoid such undelivered messages is not to introduce Condition 2. In this case, an agent continues to deliver messages as long as the current node has messages to transfer. But this allows the delivering path to become so long when a long message chain exists. It requires large memory spaces since the delivering path is recorded in the whiteboards of nodes. Thus, we insist on Condition 2 to save the whiteboard space. So we introduce the locking of nodes as another way to guarantee deliveries of all messages.

A reason why the above case happens is that agents which have distinct delivering paths (agents b and c in the above example) deliver the same message (message msg_6 in the above example). So we design the locking to prevent such a situation.

An agent locks the current node by writing, to the whiteboard, the port through which it arrives when the current node is unlocked. An agent that arrives at the locked node delivers a message from the node only when it arrives through the port that is used for the locking. Otherwise, it has to backtrack to the previous node in the delivering path. Note that, since all the delivering paths of the delivery part of agents start from initiators, the above strategy guarantees that agents which deliver the same message must have the same delivering path.

An agent stores its ID to *lock-member* in the whiteboard of a locked node when the agent locks the node or arrives through the port that is used for the locking. When a *lock-member* agent backtracks from the locked node, it unlocks the node and resets *lock-member* of the node to empty.

Condition 4 makes an agent backtrack to the previous node in the delivering path when it

backtracks to a node but is not a *lock-member* agent of the node. This implies that the node was already unlocked for the locking such that the agent was a *lock-member* agent, that is, an agent which made the current locking may have a distinct delivering path. This makes the agent keep backtracking along its delivering path until the agent reaches a node where the agent is a *lock-member* or it started the delivery part (simulation of nodes and delivering messages).

For clarification, consider the case of Figure 6.2, which is the same scenario as that in Figure 6.1. In Figure 6.2, the locking mechanism is adopted. In Figure 6.2-1, a delivers messages msg_1 and msg_2 from t to v and from v to u and locks t, v and u. Agent b follows a. At this moment, a and b are *lock-member* agents of v. Then, in Figure 6.2-2, a backtracks to t, unlocks u, v and t, and resets *lock-member* of v while b is still on a link e_{uv}. In Figure 6.2-3, c delivers messages msg_3, msg_4 and msg_5, locks them, and crashes when it is transferring msg_6 to y. In this case, since v is locked and b is not a *lock-member* agent of v, b backtracks to t through v, which makes difference from the case in Figure 6.1. Thus, d which arrives at x delivers messages msg_3 and msg_4 in order of y, z and w. Then it can arrive with msg_5 at v from w through the port used for locking v, so it continues to deliver messages msg_6 and msg_7 stored at v.

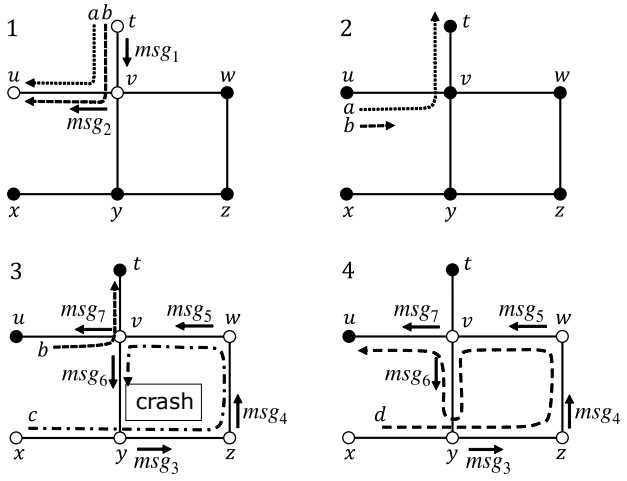

Figure 6.2: An example where the locking mechanism are applied. White nodes are locked.

An agent resumes the message delivery when it finds its ID in *lock-member*. It terminates the delivery part and resumes the search part (i.e., searching an initiator) when it reaches the node

where the agent started the delivery part but is not a *lock-member* agent.

6.3.2 The pseudo codes

Algorithms 6.1, 6.2, 6.3 and 6.4 are the pseudo codes of the fault-tolerant simulating algorithm.

We use operations $enqueue(q, M)$, $dequeue(q)$ and $head(q)$ to handle message queue q at a node. Operation $enqueue(q, M)$ for message sequence M is used to append M to the tail of q, $dequeue(q)$ is used to delete the head element of q and $head(q)$ is used to refer to the head element of q. Notation $v.var$ denotes variable var stored in the whiteboard of the current node v, and $a.var$ denotes variable var stored in the notebook of agent a.

We show the variables with their types and initial values in in the following. Actually, $v.port_{search}$, $v.parent_{search}$, $v.parent_{deliver}$ and $v.receive$ are sets of triplets or pairs (e.g., $v.port_{search}$ is a set of triplets $(agentID, port, binary)$) but, for convenience, we denote them as arrays in the following and pseudo codes.

$v.port_{search}[agentID][port]$ **:** It is a *binary* variable. Its initial value is 1. "$v.port_{search}[a.id][p] = 1$" implies port p of v is unsearched by agent a in the search part.

$v.parent_{search}[agentID]$ **:** It stores a *port number*. Its initial value is $\perp$. "$v.parent_{search}[a.id] = p$" implies that agent a arrives at v for the first time from port p in the search part.

$v.parent_{deliver}[agentID]$ **:** It stores a *port number*. Its initial value is $\perp$. "$v.parent_{deliver}[a.id] = p$" implies that agent a arrives at v from port p for the first time since it has started a delivery part.

$v.init$ **:** It is a *boolean* variable. The variable indicates whether v is an initiator of the target (message-passing) algorithm Z. It initially is *true* only if v is an initiator and is false otherwise.

$v.port_{lock}$ **:** It stores a *port number*. Its initial value is $\perp$. The variable indicates whether v is locked or not. It is $\perp$ when v is not locked, or port p if v is locked using p.

$v.send$ **:** It is a *message queue* storing messages to transfer to neighbors. It initially is a empty sequence.

$v.send\_member$ **:** It is a *set of agent IDs*. It initially is $\emptyset$. "$a.id \in v.send\_member$" implies a is *send-member* of the head message of $v.send$.

$v.lock\_member$ **:** It is a *set of agent IDs*. It initially is $\emptyset$. "$a.id \in v.lock\_member$" implies a is *lock-member* of v.

$v.state_n$ **:** It stores a *node state* of v of the target algorithm Z. Its initial value is the initial state in Z.

$v.receive[port]$ **:** It stores a *message*. Its initial value is $\perp$. $v.receive[p] = m$ implies message m is the latest message received from p.

$a.msg$ **:** It stores a *message*. Its initial value is $\perp$. It stores a message which a is delivering.

At the moment agent a starts Algorithm 6.1 at node v, if a finds $f+1$ other agents, a terminates the algorithm (Algorithm 6.1, line 1). Agent a stores 0 to $v.parent_{search}[a.id]$ to declare that v is the homebase of a (Algorithm 6.1, line 2). Then, a starts the depth-first search with recording the port through which a arrives in $v.parent_{search}[a.id]$ at each visited node v (Algorithm 6.1, line 19) and storing 0 to $v.port_{search}[a.id][p]$ for each searched port p (Algorithm 6.1, lines 8, 16, and 20). When a finds an initiator, a executes DELIVER() (Algorithm 6.2) to simulate the message-passing algorithm Z (Algorithm 6.1, line 5). For saving whiteboard space, if the current node's $v.parent_{search}[a.id]$ is not $\perp$ (it means v is included in the path of a), a backtracks to the previous node (Algorithm 6.1, lines 15-17). To decrease the number of movements, a also backtracks to the previous node if the current node's $v.parent_{search}$ has $f+1$ IDs (i.e., $f+1$ agents have already visited the node during their search part) (Algorithm 6.1, lines 11 and 12). Agent a terminates if the current node's $v.parent_{search}[a.id]$ is 0 (it means v is the homebase of a) and there is no $v.port_{search}[a.id][p]$ stored 1 (i.e., there is no unsearched port) (Algorithm 6.1, line 23).

At the moment agent a starts DELIVER(), a simulates execution of an initiator by $Process(\perp, \perp)$ if v is an initiator and is not processed yet (Algorithm 6.2, line 1). Then, a locks v if v is not locked, adds $a.id$ to $v.lock\_member$, and a stores 0 to $v.parent_{deliver}[a.id]$ to declare that v is the starting node of DELIVER() (Algorithm 6.2, lines 2-4). After that, a transfers and processes messages successively in the depth-first fashion with recording the port through which a arrives

Algorithm 6.1 Simulation algorithm for Z

```
1: if (|{a.id | v.parent_search[a.id] ≠ ⊥}| ≥ f + 1) then terminate;
2: v.parent_search[a.id] ← 0;
3: while (1)
4:   //the current node is an initiator
5:   if (v.init = true) ∨ (v.port_lock = 0) then DELIVER();
6:   //there is an unsearched port
7:   if (there is p s.t. v.port_search[a.id][p] ≠ 0) then
8:     v.port_search[a.id][p] ← 0;
9:     move through p, then arrive from q;
10:    //the current node is visited by f + 1 agents
11:    if (|{a.id | v.parent_search[a.id] ≠ ⊥}| ≥ f + 1) then
12:      move through q; //return to the previous node
13:    else
14:      //find a's own ID
15:      if (v.parent_search[a.id] ≠ ⊥) then
16:        v.port_search[a.id][q] ← 0;
17:        move through q; //return to the previous node
18:      else //arrive at v for the first time
19:        v.parent_search[a.id] ← q;
20:        v.port_search[a.id][q] ← 0;
21:  else //there is no unsearched port
22:    p ← v.parent_search[a..id];
23:    if (p = 0) then break;
24:    else move through p; //return to the previous node
25: end while
```

in $v.parent_{deliver}[a.id]$ (Algorithm 6.2, line 16) and its ID in $v.send\_member$ at each visited node (Algorithm 6.2, line 9). For saving whiteboard space, if the current node's $v.parent_{deliver}[a.id]$ is not $\perp$ (it means v is included in the delivering path of a), a backtracks to the previous node

(Algorithm 6.2, lines 17 and 18). Agent a also backtracks to the previous node when v is locked using a port other than the one a arrives through, that is, $v.port_{lock}$ of the current node is not $\perp$ and other than the one a arrives through (Algorithm 6.2, lines 17 and 18).

It stores $a.id$ in $v.lock\_member$ when a arrives at v with carrying a message and v is not locked or a arrives through the port used for the locking (Algorithm 6.2, line 15). If there is not $a.id$ in $v.lock\_member$ at the current node when a backtracks to v, a executes GoBack() until a finds $a.id$ in $v.lock\_member$ (Algorithm 6.2, lines 18 and 25). If GoBack() outputs 0, a terminates Deliver() and resumes Algorithm 6.1. If GoBack() outputs 1, a continues Deliver() to transfer messages.

Function GoBack() (Algorithm 6.3) is called in Deliver() when a backtracks to the previous node. Agent a continues to backtrack through the port in $v.parent_{deliver}[a.id]$ until a finds $a.id$ in $v.lock\_member$ (Algorithm6.3, lines 2-13). If a finds $a.id$ in $v.lock\_member$, GoBack() outputs 1 (Algorithm 6.3, line 6) and a restarts Deliver() to transfer messages from the node. If a does not find $a.id$ in $v.lock\_member$, GoBack() outputs 0 (Algorithm 6.3, line 10) and a terminates Deliver() and resumes the depth-first search for finding an initiator. While searching $a.id$, a removes the messages which it delivered, i.e., $a.id$ is included in $v.send\_member$ (Algorithm 6.3, lines 4 and 5).

Function Process() (Algorithm 6.4) is used to simulate execution of nodes in Z. If the current node is an unprocessed initiator, a simulates execution of the node (Algorithm 6.4, lines 2-4). To simulate the execution of an initiator, a uses $simulate(v.state_n, \perp)$ and it gets a new node state s and a new message sequence MSG. To simulate the execution of a node on receipt of a message msg, a uses $simulate(v.state_n, msg)$ and it gets a new node state s and a new message sequence MSG (Algorithm 6.4, lines 6-8).

Each message may be delivered multiple times by agents on Algorithm 6.2. To make sure that each message is processed once, the latest message delivered from each port p is stored in $v.receive[p]$ (Algorithm 6.4, line 7). A message is not processed again if it has been already stored in $v.receive[p]$.

Note that when f is not given, agents ignore lines 14,15 in Algorithm 6.1.

Algorithm 6.2 DELIVER()

```
1:  Process(⊥, ⊥); //process a unprocessed initiator
2:  if (v.port_lock = ⊥) then v.port_lock ← 0;
3:  v.lock_member ← v.lock_member ∪ {a.id};
4:  v.parent_deliver[a.id] ← 0; //mark 0 on the starting node of DELIVER();
5:  while (1)
6:    if (v.send ≠ ∅) then
7:      //copy the head message of v.send
8:      a.msg ← head(v.send);
9:      v.send_member ← v.send_member ∪ {a.id};
10:     move through the destination port p of a.msg;
11:     arrive from q;
12:     Process(a.msg, q); a.msg ← ⊥;
13:     if ((v.port_lock = ⊥) ∨ (v.port_lock = q)) ∧ (v.parent_deliver[a.id] = ⊥) then
14:       if (v.port_lock = ⊥) then v.port_lock ← q;
15:       v.lock_member ← v.lock_member ∪ {a.id};
16:       v.parent_deliver[a.id] ← q;
17:     else //backtrack to a node
18:       if (Go_back(q) = 0) then return;
19:   else
20:     if (a.id ∈ v.lock_member) then
21:       v.port_lock ← ⊥; v.lock_member ← ∅;
22:     q ← v.parent_deliver[a.id]; v.parent_deliver[a.id] ← ⊥;
23:     if (q = 0) then return;
24:     else //backtrack to a node s.t. a is a lock-member
25:       if (Go_back(q) = 0) then return;
26: end while
```

6.3.3 Correctness

In this section, we show that the proposed algorithm simulates Z correctly.

Algorithm 6.3 GoBack(p)

```
1: move through p (return to the previous node);
2: while (1)
3:   //a has delivered the head message of v.send
4:   if (a.id ∈ v.send_member) then
5:     v.send_member ← ∅; dequeue(v.send);
6:   if (a.id ∈ v.lock_member) then return 1; //a resumes deliveries
7:   else
8:     q = v.parent_deliver[a.id]; v.parent_deliver[a.id] ← ⊥;
9:     if (q = 0) then //the starting node of Deliver();
10:      return 0; //return from Deliver();
11:    else //a is not a v.lock_member agent
12:      move through q; // return to the previous node
13: end while
```

Algorithm 6.4 Process(msg, p)

```
1: //simulate the execution of an initiator
2: if (v.init = true) then
3:   v.init ← false; (s, MSG) ← simulate(v.state_n, ⊥);
4:   v.state_n ← s; enqueue(v.send, MSG);
5: //simulate the execution of a node receiving msg from q
6: if (msg ≠ ⊥) ∧ (msg ≠ v.receive[p]) then
7:   v.receive[p] ← msg; (s, MSG) ← simulate(v.state_n, msg);
8:   v.state_n ← s; enqueue(v.send, MSG);
```

First, we define the time instants of send and receive operations in the simulation of message-passing algorithm Z.

- The time instant that v sends message msg in the simulation of Z is defined as the time instant that an agent stores msg to $v.send$.

- The time instant that v receives message msg in the simulation of Z is defined as the time

instant that an agent with carrying message *msg* arrives at v for the first time and simulates local computation of v initiated by receipt of *msg*.

Since an agent never crashes during an atomic step, it completes an atomic step of Z. Hence, to prove the correctness of the proposed algorithm, it is sufficient to show that *1)* local computation of every initiator is started and *2)* every message is delivered in a reliable and FIFO manner.

Hereafter, we say an agent is in the delivery mode when it executes procedures DELIVER(), GOBACK() or PROCESS(), and an agent is in the search mode otherwise. We say the node, say u, specified by $v.port_{lock}$ is the locking node of v and u locks v. We first show that every initiator is processed.

Lemma 6.3.1. *In the execution of Algorithm 6.1, each node is visited by at least one non-faulty agent of the search mode and hence every initiator starts execution of Z.*

Proof. If a non-faulty agent in the search mode visits an initiator node that has not started its first local computation, it starts the local computation. Hence it is sufficient to show that each node is visited by at least one non-faulty agent.

We prove the lemma by contradiction. We assume that, when every agent terminates the algorithm, there exists a node that is not visited by any non-faulty agent. Since the network is connected, there exist adjacent nodes v_1 and v_2 such that v_1 is visited by at least one non-faulty agent and v_2 is not visited by any non-faulty agent.

Let us assume a be a non-faulty agent that visits v_1. Since a continues to make a forward move of the depth-first search unless it visits a node that $f + 1$ agents have already visited. This implies that, since a does not move from v_1 to v_2, $f + 1$ agents have already visited v_1. Because there exist at most f faulty agents, at least one of the $f + 1$ agents is non-faulty and the non-faulty agent visits v_2. This is a contradiction. □

Next, we show that agents simulate reliable communications in the following lemmas.

Lemma 6.3.2. *In the execution of Algorithm 6.1, when a node, say v, has a message in v.send, v is locked.*

Proof. Node v has no message and is not locked initially. When a message is delivered or v is

an initiator and is processed by an agent, v is locked. Node v is unlocked only when there is no message in $v.send$. Thus, when v has a message in $v.send$, v is locked. □

Lemma 6.3.3. *In the execution of Algorithm 6.1, agents simulate reliable communication.*

Proof. First, we show that all messages which were transfered are eventually delivered. Since messages are delivered if they are deleted, it is enough to prove that all messages have already been deleted when execution of the algorithm terminates. We assume for contradiction that there is a node v with a message in $v.send$ after the execution of the algorithm terminates.

From Lemma 6.3.2, v is locked. Without loss of generality, we assume that v locks no node. Let u be v's locking node and p be the port of (u, v) at v. We show by contradiction that u is never unlocked before v is unlocked. Assume that u is unlocked before v. Suppose that agent a and agent b is at u and there is a message msg_1 to v in $u.send$. Only the following two cases are possible.

1. Agent a backtracks from v to u immediately after delivering msg_1 and b delivers msg_1 to v from u before a reaches u, continues message delivery, and locks v. Then, u is unlocked by a.

2. Agent a delivers msg_1 to v and continues message delivery and b backtracks from u and unlocks u.

Consider the first case. Since a backtracks from v, v has no message, v is locked by a port other than p, or a cycle is detected (i.e., a has visited v in its current delivery mode).

Suppose that a detects a cycle at v but b detects no cycle at v. This leads to that there exists node w such that $w.parent_{deliver}[a.id]$ and $w.parent_{deliver}[b.id]$ are different. An agent which can deliver a message from w however comes from a port specified by $w.port_{lock}$ and either of $w.parent_{deliver}[a.id]$ and $w.parent_{deliver}[b.id]$ is different from $w.port_{lock}$. This is a contradiction. Suppose that v has no message. This is a contradiction since b continues message delivery from v. Suppose that v is locked by a port other than p. In this case, b backtracks from v immediately after delivering msg_1 since v is locked by a port other than p or v has no message if v is unlocked when b visits v. This is a contradiction.

Consider the second case. Since b backtracks from v, v has no message, v is locked by a port other than p, or a cycle is detected. By the similar argument, when b detects a cycle, a detects a cycle. Since a continues message delivery, v has a message and is locked by p which is also used by b. This is a contradiction.

By applying the above claim inductively, we get that an initiator is locked. By Lemma 6.3.1, however, every initiator is unlocked eventually, which is a contradiction.

Thus, all messages which are generated on Z are deleted and delivered.

Because agents simulate local computation initiated by receipt of a message exactly once, condition [A1] in Section 6.2.3 holds. Every message received by a node is carried by an agent and before that it is put into a message queue of the sender node (otherwise, the message cannot be delivered by an agent). This implies condition [A2] in Section 6.2.3 also holds. Hence, agents simulate reliable communication. □

Lemma 6.3.4. *In the execution of Algorithm 6.1, agents simulate the FIFO order of message communication.*

Proof. Let us consider two messages msg_1 and msg_2 from node v to node u such that msg_1 is sent before msg_2. Then, we show that msg_1 is received by u before msg_2. Since an agent transfers the message at the head of queue, msg_1 is deleted before msg_2 is sent. Message msg_1 is deleted only when the agent which delivers msg_1 returns. At this time, msg_1 has been received. Thus, msg_1 is received before msg_2. □

From Lemmas 6.3.1, 6.3.3 and 6.3.4, the proposed algorithm initiates execution of all initiators and delivers all messages to their destinations correctly. This implies the following theorem (about the correctness of the proposed algorithm) holds.

Theorem 6.3.1. *Algorithm 6.1 simulates Z correctly when at most $f \leq k - 1$ agents are faulty.*

6.3.4 Complexities

In this part, we evaluate the move and memory complexity of agents. Clearly it depends on the target message-passing algorithm Z. Let M and L be the number and the maximum size of messages created in the simulated execution of algorithm Z respectively, and M_v be the maximum

number of messages created at node v in Z. Cardinality $|array|$ denotes the number of elements of *array*. Note that, if f is not given, f is replaced by $k-1$. This is because a search (resp, delivery) mode agent a returns from a node v when it finds its $a.id$ in $v.parent_{search}$ (resp, $v.parent_{deliver}$) of v.

Theorem 6.3.2. *Algorithm 6.1 simulates Z with $O((m+M)f)$ total agent moves, $O(f)$ agent moves per message (when $m = O(M)$), $O(L)$ agent memory and $O((M_v+\Delta)L + f\Delta\log(k\Delta))$ additional memory for each node.*

Proof. We first evaluate the number of total agent moves. For the search mode, at most $f+1$ agents search each link in each direction. One search consists of a forward move and a backward move. Thus, the move complexity of the search mode is $2 \cdot 2 \cdot m \cdot (f+1) = 4m(f+1)$. For the delivery mode, at most $f+1$ agents carry each message of Z by forward moves. Every agent makes one backward move for each forward move. Thus, the total move complexity of the delivery mode is $2M(f+1)$. Therefore, the move complexity is $4m(f+1) + 2M(f+1) = O((m+M)f)$ and the number of agent moves per message is $O(f)$.

Second, we evaluate the memory complexity of agents. Memory complexity of agents is $O(L)$ since an agent retains at most one message of Z.

And finally, we evaluate the additional memory complexity of nodes. Nodes retain variables of our algorithm as additional variables. Variable $v.send$ is used for storing messages that are generated at node v and waiting for being delivered to their destination, which requires $O(M_v L)$ space. Variable $v.port_{search}$ requires $O(f\Delta\log(k\Delta))$ space since each element of $v.port_{search}$ is triplet $(agentID, port, boolean)$ and requires $O(\log k + \log\Delta)$ and $|v.port_{search}|$ is $O(f\Delta)$. Since each element of $v.parent_{search}$ is pair $(agentID, port)$ and requires $O(\log k + \log\Delta)$ and $|v.parent_{search}|$ is at most $f+1$, $v.parent_{search}$ requires $O(f\log(k\Delta))$ space. Since each element of $v.parent_{deliver}$ is pair $(agentID, port)$ and requires $O(\log k + \log\Delta)$ and $|v.parent_{deliver}|$ is at most $f+1$ (each message is delivered by at most $f+1$ agents), $v.parent_{deliver}$ requires $O(f\log(k\Delta))$ space. Since $v.init$ retains whether v is an initiator or not, $v.init$ requires $O(1)$ space. Since $v.port_{lock}$ retains the port used for lock, $v.port_{lock}$ requires $O(\log\Delta)$ space. Since $v.send\_member$ retains IDs of agents which deliver the head message of $v.send$, $v.send\_member$ requires $O(f\log k)$ space. Since $v.lock\_member$ retains IDs of agents which arrived from the same port with carrying the same

message, $v.lock\_member$ requires $O(f \log k)$ space. Since $v.receive$ retains the latest message for each port, $v.receive$ requires $O(\Delta L)$ space.

From these, the amount of additional memory required at node v is $O((M_v + \Delta)L + f\Delta \log k\Delta)$.

□

Theorem 6.3.3. *The number of moves of all the agents to simulate message passing algorithms with a finite number of messages is* $\Theta((m + M)f)$.

Proof. Because of asynchronous communication and f faulty agents, all the links must be searched by at least $f + 1$ agents and all the messages must be transferred by at least $f + 1$ agents. Thus, the number of total agent moves is $\Omega((m+M)f)$. From this and Theorem 6.3.2, the theorem holds. □

6.4 Simulation of message-passing algorithms with an infinite number of messages

The simulating algorithm we proposed in Section 6.3 cannot simulate a message-passing algorithm Z with an infinite number of messages denoted by Z_{inf} as explained below.

Consider the case of Figure 6.3. There are two independent infinitely long message chains C_a and C_b. If all the agents in the network transfer the messages included in C_a in the depth-first fashion, the messages included in C_b are not delivered forever.

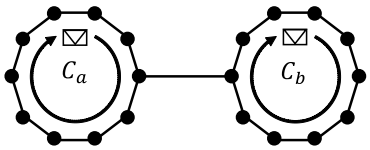

Figure 6.3: An example where the algorithm proposed in Section 6.3 cannot simulate message-passing algorithm Z_{inf}.

To simulate message-passing algorithm Z_{inf}, we introduce, to the depth-first message delivery, restriction on the number of message deliveries and modify the algorithm as follows.

1. Instead of the depth-first message delivery in Section 6.3, the depth-first delivery with restriction ℓ on the number of delivered messages is adopted. It is a modification of the depth-first delivery such that an agent backtracks to the node where it started the current delivery mode when the (combined) number of (distinct) messages delivered by the agent reaches ℓ.

2. Each agent repeats the depth-first search of the search part infinitely and visits all nodes in each of the depth-first search of the search part.

3. An agent of the search mode stops execution of the simulating algorithm when it finds $f+1$ (distinct) delivery mode agent names in the current node. Otherwise, it can happen that either each message is delivered by more than $f+1$ agents or an agent traverses whole network during one message is delivered. In both cases, the move complexity per message gets $\omega(f)$.

4. An agent starts the delivery part not only when it finds an initiator, but also when it finds a message to transfer on a non-locked node.

We insist on the locking mechanism so that the number of agent moves per message becomes $O(f)$: without the locking, the number of delivered messages per traversal may become constant, i.e., the number of agent moves per message is $O(nk)$. Recall the example presented in Section 6.3. In the example, six messages are successfully delivered and an undelivered message is left after all the agents finish their traversals. When this happens in every traversal, the number of moves per message becomes $O(nk)$, i.e., only six messages are delivered in every traversal.

To combine locking mechanism and restriction on the number of message deliveries, the agents in the same message delivery tree have to share the combined number of distinct messages delivered by the agents. Otherwise, some node can remain locked with an undelivered message.

For example, consider the following adversarial scenario depicted in Figure 6.4 where $f = 1$ and $\ell = 3$. Also in the figure, locked (non-locked) nodes are colored white (black). Agent a first delivers messages msg_1 and msg_2 from u to w via v and then backtracks to u. After that, a and b deliver msg_3 from u to x. Since a has delivered three messages, it backtracks to u and resumes searching, whereas b delivers msg_4 and crashes during the delivery. At this moment, x remains

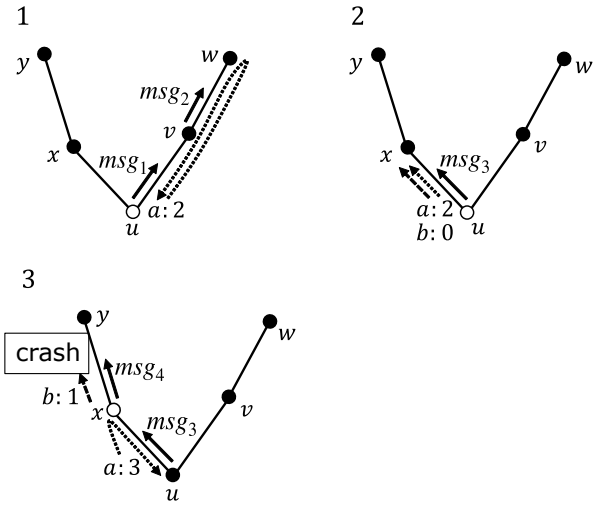

Figure 6.4: An example without sharing the number of delivered messages where $\ell = 3$. In this case, x remains locked with an undelivered message msg_4.

locked by the port leading to u and msg_4 remains undelivered. Furthermore, x is kept locked by the port and msg_4 is never delivered unless a message is delivered from u.

To share the number of distinct delivered messages, an agent stores the number to share in a variable of a node. The variable of node v is updated to the variable of an agent reaching v after the agent reaches v with a new message and increments the variable of the agent or when the agent backtracks v and the agent is a *lock-member* of v. Otherwise, the value of the variable is copied to the variable of an agent.

Let us consider the previous case again with sharing the number of distinct delivered messages depicted in Figure 6.5. Agent a first has the value 0 for its variable. It delivers messages msg_1 and msg_2 from u to w via v and stores 1 in v and 2 in w. Then, a backtracks to u with storing 2 in v and u. At this moment, b also starts from u and it copies 2 from u to its variable. Before b delivers msg_3, a delivers msg_3, stores 3 in x, and backtracks to u. When b delivers msg_3, b also has 3 and backtracks to u. In this case, even when one of the agents crashes, x does not remain locked.

There is another problem: allowing a non-locked node to have a message can also lead to the situation where some node may remain locked with an undelivered message.

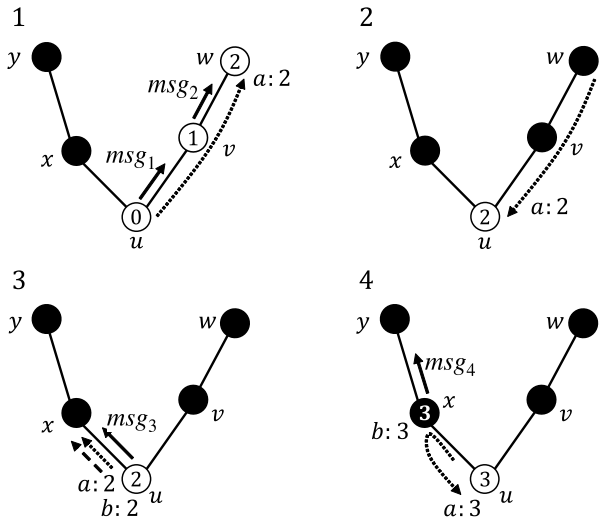

Figure 6.5: An example with sharing the number of distinct delivered messages where $\ell = 3$.

For example, consider the following adversarial scenario depicted in Figure 6.6 where $f = 1$. Suppose that v is locked at first and a and b are delivering msg_1 from u to v. Agent a reaches v before b and, when reaching, v is still locked. Since v is locked, a backtracks from v right after delivering msg_1. Then, v is unlocked with a message msg_2 and, after that, b reaches v. Since v is not locked, b locks v and delivers msg_2 and, during delivering msg_2, b crashes. In this case, v remains locked by the port leading to u and msg_2 remains undelivered. Furthermore, v is kept locked by the port and msg_2 is never delivered unless a message is delivered from u.

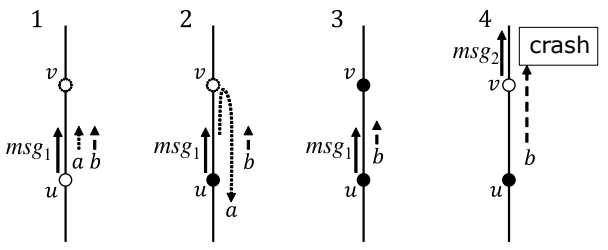

Figure 6.6: An example where an agent can deliver message while another agent has already returned. In this case, v remains locked with an undelivered message msg_2.

To avoid the above case, an agent tells the other agents that it backtracks right after message delivery by setting a flag.

6.4.1 Pseudo codes

Algorithms 6.5, 6.7, 6.8 and 6.9 are the pseudo codes of the fault-tolerant simulating algorithm of Z_{inf}.

Algorithm 6.5 is based on Algorithm 6.1 in Section 6.3.2. The difference from Algorithm 6.1 is as follows:

1. agents traverse all the node infinitely often (expressed in Algorithm 6.5 as infinite loop at line 2),

2. an agent finding other $f+1$ agents executing DeliverInf() on a node terminates (Algorithm 6.5, line 5),

3. when an agent traverses all the links of the network for the first time, it constructs its DFS tree, i.e. it labels $\perp$ on a port p which is not included the DFS tree (Algorithm 6.5, lines 11 and 13).

By the third change, the agent can traverse all the nodes of the network with $O(n)$ moves in the second and later traversal.

Algorithm 6.6 and 6.7 which are based on Algorithm 6.2 in Section 6.3.2 are the pseudo codes of function DeliverInit and DeliverInf.

To memorize and share between agents the number of delivered messages, we introduce new variables $a.deliver$ for agent a and $v.deliver[p]$ (assigned to a port p when $1 \leq p \leq deg_v$ and not port when $p = 0$) for node v. When a for the first time reaches v with a new message or backtracks to v and $a.id$ is in $v.lock\_member$, $a.deliver$ gets stored in $v.deliver[q]$ (Algorithm 6.7, lines 9, 13, and 19) where q is the port corresponding to $v.parent_{deliver}[a.id]$. Otherwise, $v.deliver[q]$ gets stored in $a.deliver$ (Algorithm 6.7, lines 10, 14, and 20).

To tell agents that an agent backtracks right after message delivery, we also introduces a new variable $v.backtrack[p]$ ($1 \leq p \leq deg_v$) for node v. When agent a backtracks right after message delivery from port p and $v.backtrack[p] = 0$, a sets 1 to $v.backtracks[p]$ (Algorithm 6.2, lines 13

Algorithm 6.5 Simulation algorithm for Z_{inf}

1: $v.parent_{search}[a.id] \leftarrow 0$;
2: **while** (1)
3: //the current node is initiator or has messages
4: **if** $(v.init = true) \vee (v.port_{lock} = 0) \vee ((v.port_{lock} = \perp) \wedge (v.send \neq \emptyset))$ **then**
5: **if** $(|v.lock\_member| = f + 1)$ **then terminate**;
6: DeliverInf();
7: **if** (there is p s.t. $v.port_{search}[a.id][p] = 1$) **then** //there is an unsearched port
8: $v.port_{search}[a.id][p] \leftarrow 0$;
9: move through p, then arrive from q;
10: **if** $(v.parent_{search}[a.id] \neq \perp)$ **then** //find a's own ID
11: $v.port_{search}[a.id][q] \leftarrow \perp$;
12: move through q, then arrive from p; //return to the previous node
13: $v.port_{search}[a.id][p] \leftarrow \perp$;
14: **else** //arrive at v for the first time
15: $v.parent_{search}[a.id] \leftarrow q$; $v.port_{search}[a.id][q] \leftarrow 0$;
16: **else** //there is no unsearched port
17: $v.port_{search}[a.id][p] \leftarrow 1$ for all p s.t. $v.port_{search}[a.id][p] = 0$;
18: $q \leftarrow v.parent_{search}[a.id]$;
19: **if** $(q \neq 0)$ **then** move through q; //return to the previous node
20: **end while**

and 19). When an agent visits node v from port p and $v.backtrack[p] = 1$, the agent backtracks the node corresponding to p.

Algorithm 6.8 which is based on Algorithm 6.3 in Section 6.3.2 is the pseudo code of function GoBackInf. A main modification here is that when agent a backtracks node v and $a.id \in v.lock\_member$, $a.deliver$ gets stored in $v.deliver[q]$ where q is the port by which v is locked.

Algorithm 6.9 which is based on Algorithm 6.4 in Section 6.3.2 is the pseudo code of function ProcessInf. The only modification here is that when message m from port p is delivered to v for

Algorithm 6.6 DELIVERINIT()

1: $a.deliver \leftarrow 0$; PROCESSINF($\perp$, $\perp$); //process an unprocessed initiator
2: $v.lock\_member \leftarrow v.lock\_member \cup \{a.id\}$; $v.parent_{deliver}[a.id] \leftarrow 0$;
3: **if** ($v.port_{lock} = \perp$) **then** $v.port_{lock} \leftarrow 0$;
4: **if** ($v.send\_member = \emptyset$) **then** $v.deliver[0] \leftarrow a.deliver$;
5: **else** $a.deliver \leftarrow v.deliver[0]$;

the first time, $v.backtrack[p]$ is set to 0.

Note that when f is not given, agents ignore lines 5, 6 in Algorithm 6.5.

6.4.2 Correctness

In this subsection, we show that the proposed algorithm simulates Z_{inf} correctly.

The time instants of send and receive operations in the simulation of message-passing algorithm Z_{inf} are defined in the similar way in Section 6.3.3 as follows:

- the time instant that v sends message *msg* in the simulation of Z_{inf} is defined as the time instant that an agent stores *msg* to $v.send$, and
- the time instant that v receives message *msg* in the simulation of Z_{inf} is defined as the time instant that an agent with carrying message *msg* arrives at v for the first time and simulates local computation of v initiated by receipt of *msg*.

Also in for the algorithm, to prove the correctness of the proposed algorithm, it suffices to show that *1)* local computation of every initiator is started (which is proved in Lemma 6.4.5) and *2)* each message is delivered in a reliable and FIFO manner (which is proved in Lemma 6.4.6).

Hereafter, we say an agent is in the delivery mode when it executes procedures DELIVERINF(), GOBACKINF() or PROCESSINF(), and an agent is in the search mode otherwise. We say the node, say u, specified by $v.port_{lock}$ is the locking node of v and u locks v. The following lemmas hold.

Lemma 6.4.1. *In the execution of Algorithm 6.5, there remains at least one non-faulty agent.*

Proof. A non-faulty agent stops execution if it finds $f + 1$ delivery mode agents during the search mode. At least one of the $f + 1$ delivery mode agents must be a non-faulty agent. □

Algorithm 6.7 DeliverInf()

```
1: DeliverInit();
2: while (1)
3:   if (v.send ≠ ∅) then
4:     a.msg ← head(v.send); //copy the head message of v.send
5:     v.send_member ← v.send_member ∪ {a.id};
6:     move through the destination port p of a.msg, then arrive from q;
7:     a.msg ← ⊥; a.deliver = a.deliver + 1; ProcessInf(a.msg, q);
8:     if (v.port_lock ∈ {⊥, q})∧(v.parent_deliver[a.id] = ⊥)∧(a.deliver ≠ ℓ)∧(v.backtrack[q] = 0)
     then
9:       if (v.port_lock = ⊥) then v.port_lock ← q; v.deliver[q] ← a.deliver;
10:      else a.deliver ← v.deliver[q];
11:      v.lock_member ← v.lock_member ∪ {a.id}; v.parent_deliver[a.id] ← q;
12:    else //backtrack to a node
13:      if (v.backtrack[q] = 0) then v.backtrack[q] ← 1; v.deliver[q] ← a.deliver;
14:      else a.deliver ← v.deliver[q]
15:      if (GoBackInf(q) = 0) then return;
16:  else //there is no message
17:    q ← v.parent_deliver[a.id];
18:    if (a.id ∈ v.lock_member) then
19:      v.port_lock ← ⊥; v.lock_member ← ∅; v.backtrack[q] ← 1; v.deliver[q] ← a.deliver;
20:    else a.deliver ← v.deliver[q];
21:    q ← v.parent_deliver[a.id]; v.parent_deliver[a.id] ← ⊥;
22:    if (q = 0) then return;
23:    else //backtrack to a node s.t. a is a lock-member
24:      if (GoBackInf(q) = 0) then return;
25: end while
```

Lemma 6.4.2. *In the execution of Algorithm 6.5, when a node, say v, has a message in v.send, v is locked.*

Algorithm 6.8 GoBackInf(p)

```
1: move through p (return to the previous node);
2: while (1)
3:   if (a.id ∈ v.send_member) then
4:     //a has delivered the head message of v.send
5:     v.send_member ← ∅; dequeue(v.send);
6:   q = v.parent_deliver[a.id];
7:   if (a.id ∈ v.lock_member) then v.deliver[q] ← a.deliver;
8:   else a.deliver ← v.deliver[q];
9:   if (a.id ∈ v.lock_member) ∧ (a.deliver < ℓ) then return 1; //a resumes deliveries
10:  else
11:    v.parent_deliver[a.id] ← ⊥;
12:    if (a.id ∈ v.lock_member) then
13:      v.lock_member ← ∅; v.port_lock ← ⊥;
14:      v.backtrack[q] ← 1;
15:    if (q = 0) then //the starting node of DeliverInf()
16:      return 0; //return from DeliverInf()
17:    else //if a is not a v.lock_member agent
18:      move through q; //return to the previous node
19: end while
```

Proof. The lemma can be proven by the same argument in the proof of Lemma 6.3.2 and its proof is omitted here. □

Lemma 6.4.3. *In the execution of Algorithm 6.5, after an agent, say a, starts its delivery mode, a.deliver eventually reaches ℓ unless a crashes.*

Proof. We show that *a.deliver* increases eventually. During the delivery mode, *a.deliver* is changed at node v in the following cases:

1. *a.deliver* is incremented when a delivers a message and a is the first agent delivering the message;

Algorithm 6.9 PROCESSINF(msg, p)

```
1: //simulate the execution of an initiator
2: if (v.init = true) then
3:    v.init ← false;
4:    (s, MSG) ← simulate(v.state_n, ⊥);
5:    v.state_n ← s;
6:    enqueue(v.send, MSG);
7: //simulate the execution of a node receiving msg from p
8: if (msg ≠ ⊥) ∧ (msg ≠ v.receive[p]) then
9:    v.receive[p] ← msg;
10:   v.backtrack[p] ← 0
11:   (s, MSG) ← simulate(v.state_n, msg);
12:   v.state_n ← s;
13:   enqueue(v.send, MSG);
```

2. $a.deliver$ is copied from $v.deliver[p]$ where p is $v.parent_{deliver}[a.id]$ when a delivers a message and a is not the first agent delivering the message;

3. $a.deliver$ is copied from $v.deliver[p]$ when a backtracks to v and is not in $v.lock\_member$.

In the first case, $a.deliver$ increases and thus we show the lemma for the second and third cases. We first show that in the third case, when a stops backtracking and resumes message delivery, $a.deliver$ is the number of messages delivered in its current delivery path. When a unlocks node u and starts to backtrack from u, $a.deliver$ is the number of messages delivered in its current delivery path and $a.deliver$ is copied to $u.deliver[q]$ where q is $u.parent_{deliver}[a.id]$. When a starts to backtrack from non-locked node u, $a.deliver$ is copied from $u.deliver[q]$. By applying the claim inductively, we have that $a.deliver$ and $v.deliver[p]$ is the number of messages delivered in its current delivery path when a stops backtracking and resumes message delivery. Also for the second case, by the above arguments, $v.deliver[p]$ is the number of messages delivered in its current delivery path when the second case occurs. Since the number of messages delivered in its current delivery path keeps increasing until reaching ℓ, $a.deliver$ eventually reaches ℓ. □

Lemma 6.4.4. *In the execution of Algorithm 6.5, every agent terminates each of its delivery modes or crashes.*

Proof. An agent, say a, terminates its delivery mode if there is no message to deliver or $a.deliver$ reaches ℓ. As long as there is a message to deliver, a keeps delivering a message and $a.deliver$ eventually reaches ℓ by Lemma 6.4.3 and terminates its delivery mode. □

Lemma 6.4.5. *In the execution of Algorithm 6.5, each node is visited by a non-faulty agent of the search mode infinitely often and hence every initiator starts execution of Z_{inf}.*

Proof. By Lemmas 6.4.1 and 6.4.4, there remains at least one non-faulty agent which keeps traversing the whole network infinitely often. Thus, each node is visited an infinitely often and every initiator is processed by a non-faulty agent. □

Lemma 6.4.6. *In the execution of Algorithm 6.5, agents simulate reliable communication.*

Proof. First, we show that all messages which were transfered are eventually delivered. Since messages are delivered if they are deleted, it is enough to prove that all messages have already been deleted when execution of the algorithm terminates. We assume for contradiction that there is a node v with a message which is never delivered and v locks no node.

From Lemma 6.4.2, v is locked. Let u be v's locking node and p be the port of (u, v) at v. We show by contradiction that u is never unlocked before v is unlocked. Assume that u is unlocked before v. Suppose that agent a and agent b is at u and there is a message msg_1 to v in $u.send$. Only the following two cases are possible.

1. Agent a backtracks from v to u immediately after delivering msg_1 and b delivers msg_1 to v from u before a reaches u, continues message delivery, and locks v. Then, u is unlocked by a.

2. Agent a delivers msg_1 to v and continues message delivery and b backtracks from u and unlocks u.

Consider the first case. Since a backtracks from v, v has no message, v is locked by a port other than p, a cycle is detected (i.e., a has visited v in its current delivery mode), or $a.deliver$

is ℓ. We only consider the last case where $a.deliver$ is ℓ since other cases can be shown by the similar argument to the proof of Lemma 6.3.3. Since $a.deliver$ is ℓ, $v.deliver[p]$ is also ℓ where p is $v.parent_{deliver}[a.id]$ from the argument in the proof of Lemma 6.4.3. When b delivers msg_1, $v.deliver[p]$ is copied to $b.deliver$ and then b also backtracks from u. This is a contradiction.

Consider the second case. Since b backtracks from v, v has no message, v is locked by a port other than p, a cycle is detected, or $b.deliver$ reaches ℓ. Also in this case, we only consider the last case where $b.deliver$ reaches ℓ. Let w be u's locking node. Since both a and b pass through w, both $a.deliver$ and $b.deliver$ are copied from $w.deliver[q]$ where q is $w.parent_{deliver}[a.id]$ or $a.deliver$ is copied to $w.deliver[q]$ and $b.deliver$ is copied from $w.deliver[q]$. In either case, $a.deliver$ equals to $b.deliver$ and $a.deliver$ also reaches ℓ at u. This is a contradiction.

By applying the above claim inductively, we get that a node where message delivery is started is locked. By Lemma 6.3.1, however, every node is visited infinitely often and thus is unlocked eventually, which is a contradiction.

Thus, all messages which are generated on Z_{inf} are deleted and delivered.

Because agents simulate local computation initiated by receipt of a message exactly once, condition [A1] in Section 6.2.3 holds. Every message received by a node is carried by an agent and before that it is put into a message queue of the sender node (otherwise, the message cannot be delivered by an agent). This implies condition [A2] in Section 6.2.3 also holds. Hence, agents simulate reliable communication. □

Lemma 6.4.7. *In the execution of Algorithm 6.5, agents simulate the FIFO order of message communication.*

Proof. This lemma can be proved in the same way in the proof of 6.3.4 and the proof of the lemma is omitted here. □

From Lemmas 6.4.5, 6.4.6 and 6.4.7, the proposed algorithm initiates execution of all initiators and delivers all messages to their destinations correctly. This implies the following theorem (about the correctness of the proposed algorithm) holds.

Theorem 6.4.1. *Algorithm 6.5 simulates Z_{inf} correctly when at most $f \leq k-1$ agents are faulty.*

6.4.3 Complexities

In this section, we show that the number of agent moves per message is $O(f)$ in the modified algorithm.

First, we define *the start of the i-th traversal* of agent a as the i-th moment when a is on its homebase v_h and $v.h.port_{search}[a.id][p]$ is 1 or $\perp$ for all p, and define *the i-th traversal* of a as the interval between the starts of the i-th and the $(i+1)$-th traversals of a. We say the nodes which are not locked and have messages (i.e. agents start deliveries of messages from the nodes) are *sub-initiators*. Node set V_i^a is a set of *sub-initiators* and initiators existing at the start of the i-th traversal of a, e.g. V_1^a is a set of initiators.

The following lemma and theorem hold.

Lemma 6.4.8. *In the execution of Algorithm 6.5, at least ℓ messages are delivered during the i-th traversal of the search part of an agent.*

Proof. We show that by induction on i that ℓ messages are delivered during the i-th traversal of agent a. Note that, *1)* since the number of messages of Z_{inf} is infinite, there is always at least one node from which message delivery continues infinitely and *2)* by construction, when a unlocks a node with a undelivered message (i.e., when a new sub-initiator is formed), a has delivered ℓ messages.

Basis : There are initiators in the network at the start of the execution of the algorithm. All the initiators are processed before an agent completes *the first traversal*. When all the initiators are processed, there must be at least one *sub-initiator* since Z_{inf} continues execution forever. Thus, for $i = 1$, ℓ messages are delivered.

Inductive step : Assume that ℓ messages have been delivered during the i-th traversal of a. Note that when we count the number of messages delivered in the i-th traversal of a, the messages generated from V_{i+1}^a and the messages generated by receipt of them are not included in the number. Since all the nodes of V_{i+1}^a are processed by the start of the $i+2$-th traversal of a, ℓ messages are delivered during the $i+1$-th traversal. □

Theorem 6.4.2. *Algorithm 6.5 simulates Z_{inf} with $O(f)$ agent moves per message.*

Proof. In the modified algorithm, every message is transfered by at most $f+1$ agents as in the

algorithm in Section 6.3. From Lemma 6.4.8, at least ℓ messages are delivered during each depth-first search of the search part of an agent, which takes m agent moves in the first search and n agent moves in the second or later search. That is, at least ℓ messages are delivered during $O(kn + f\ell)$ ($O(km + f\ell)$, for the first depth-first search) agent moves. Since an agent traverses the whole network, every agent can get k and n in the first depth-first search of the search part. With setting ℓ to be kn/f, the number of agent moves per message becomes $O(f)$. □

Consequently, the modified algorithm requires $O(f)$ agent moves per message as the algorithm in Section 6.3, which not only improves the previous algorithm in [22] but also is asymptotically optimal by Theorem 6.2.1 while each agent uses $O(\log(kn/f))$ spaces. Note that also in this case, f is replaced by k when f is not given.

6.5 Concluding Remarks

In this chapter, we proposed two simulating algorithms to simulate a message-passing algorithm in the mobile agent model, one is for message-passing algorithms which eventually terminate and the other is for message-passing algorithms which never terminate. Our first (resp. second) algorithm requires $O((m + M)f)$ total agent moves and $O(f)$ agent moves per message (resp. $O(f)$ agent moves per message) to tolerate at most $f \leq k - 1$ faulty (or crashed) agents where m is the number of links in the network and M is the number of messages in the simulated execution of the message-passing algorithm. The existing algorithm requires $O((m + nM)k)$ total agent moves or $O(nk)$ agent moves per message when at most $k - 1$ agents crash. Thus, our algorithm improves the previous algorithm [22] in the number of agent moves.

Chapter 7

Conclusion

7.1 Summary of the Results

In this dissertation, we consider algorithms adapting to dynamic networks or crash faults.

In Chapter 3, we considered perpetual exploration of temporal graphs with arbitrary and unknown topology, focusing on the number of agents that are necessary and sufficient to perform the task. We considered two common dynamic models: temporally connected networks, and 1-interval connected (or always connected) networks with a bounded number of missing edges at each round. We derived tight bounds for both models under fully synchronous and semi-synchronous settings, both when the agents are anonymous and when there is a leader. Our algorithms use at each node v a rotor-router mechanism; this can be implemented with either a constant number of movable tokens that can be placed on the ports of v, or with a whiteboard of size $O(\log \delta_v)$ bits.

In Chapter 4, we considered group exploration of the dynamic torus consisting 1-interval connected rings. We proposed exploration algorithms with termination and showed that the link presence detection has a considerable influence on the number of agents required to explore the $\nu \times \mu$ dynamic torus ($\nu \leq \mu$). Specifically, we showed that, without the link presence detection, $\nu + 1$ agents are necessary and sufficient to explore and, with the link presence detection, $\lceil \nu/2 \rceil + 1$ agents are necessary and sufficient when $\nu \neq 4$ and $\lceil \nu/2 \rceil + 2$ agents are necessary and sufficient when $\nu = 4$ to explore the $\nu \times \mu$ dynamic torus.

In Chapter 5, we introduced the (H, S) view by which an agent can see when the links within H hops from its current node appear during S time steps from the current time. The (H, S) view can be used to model some situations where an agent (or robot) can partly see their nearby environment or can predict the near-future changes of the environment. For a single agent with the (H, S) view, we studied the exploration of 1-interval connected rings. We give some fundamental results, i.e., impossibility of the exploration for $H + S < n$ or $S < \lceil n/2 \rceil$, possibility of the exploration for $H + S \geq n$ and $S \geq \lceil n/2 \rceil$, and upper bounds and a lower bound of the exploration time for some cases.

In Chapter 6, we proposed two simulating algorithms to simulate a message-passing algorithm in the mobile agent model, one is for message-passing algorithms which eventually terminate and the other is for message-passing algorithms which never terminate. Our first (resp. second) algorithm requires $O((m + M)f)$ total agent moves and $O(f)$ agent moves per message (resp. $O(f)$ agent moves per message) to tolerate at most $f \leq k - 1$ faulty (or crashed) agents where m is the number of links in the network and M is the number of messages in the simulated execution of the message-passing algorithm. The existing algorithm requires $O((m + nM)k)$ total agent moves or $O(nk)$ agent moves per message when at most $k - 1$ agents crash. Thus, our algorithm improves the previous algorithm [22] in the number of agent moves.

Acknowledgments

I have been fortunate to receive assistance from many people. I would especially like to express my gratitude to my supervisor Professor Toshimitsu Masuzawa for his guidance and encouragement. I have also received precious advice from Professors of the Graduate School of Information Science and Technology, Osaka University. Among them, I would like to extend my gratitude to Professor Shinji Kusumoto, Professor Fumihiko Ino, and Associate Professor Taisuke Izumi, for their valuable comments on this dissertation.

I would like to acknowledge Professor Paola Flocchini at University of Ottawa, Professor Nicola Santoro at Carleton University, Professor Hirotsugu Kakugawa at Ryukoku University, and Associate Professor Fukuhito Ooshita at Nara Institute of Science and Technology, for their constructive discussions. I would like to thank to Professors and staffs of Osaka University Humanware Innovation Program for their financial support. I also thank to Ms. Hisako Suzuki and Ms. Noriko Sato for their kind support. I am also grateful to the staffs and students of Algorithm Engineering Laboratory, the Graduate School of Information Science and Technology, Osaka University. In particular, I thank to Assistance Professor Yuichi Sudo for his time and kindness.

Finally, I wishes to thank all of my families for their continuous encouragement and support.

Bibliography

[1] J. Cao and S.K. Das. *Mobile Agents in Networking and Distributed Computing*. John Wiley & Sons, 2012.

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

[3] K. Erciyes. *Distributed Graph Algorithms for Computer Networks*. Springer Science & Business Media, 2013.

[4] A.S. Fraenkel. Economic traversal of labyrinths. *Mathematics Magazine*, 43(3):125–130, 1970.

[5] Y. Afek and E. Gafni. Distributed algorithms for unidirectional networks. *SIAM Journal on Computing*, 23(6):1152–1178, 1994.

[6] V. Yanovsky, A. Bruckstein, and I. Wagner. A distributed ant algorithm for\protect efficiently patrolling a network. *Algorithmica*, 37(3):165–186, 2003.

[7] E. Bampas, L. Gasieniec, N. Hanusse, D. Ilcinkas, R. Klasing, A. Kosowski, and T. Radzik. Robustness of the rotor-router mechanism. *Algorithmica*, 78(3):869–895, 2017.

[8] A. Kosowski and D. Pajak. Does adding more agents make a difference? a case study of cover time for the rotor-router. *Journal of Computer and System Sciences*, 106:80–93, 2019.

[9] D. Ilcinkas, R. Klasing, and A.M. Wade. Exploration of constantly connected dynamic graphs based on cactuses. In *Proc. 21st International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, pages 250–262, 2014.

[10] T. Erlebach, M. Hoffmann, and F. Kammer. On temporal graph exploration. In *Proc. 42th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 444–455, 2015.

[11] D. Ilcinkas and A.M. Wade. Exploration of the T-interval-connected dynamic graphs: the case of the ring. *Theory of Computing Systems*, 62(5):1144–1160, 2018.

[12] T. Erlebach, F. Kammer, K. Luo, A. Sajenko, and J.T. Spooner. Two moves per time step make a difference. In *Proc. 46th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 141:1–141:14, 2019.

[13] M. Bournat, S. Dubois, and F. Petit. Computability of perpetual exploration in highly dynamic rings. In *Proc. 37th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 794–804, 2017.

[14] M. Bournat, A.K. Datta, and S. Dubois. Self-stabilizing robots in highly dynamic environments. *Theoretical Computer Science*, 772:88–110, 2019.

[15] G.A. Di Luna, S. Dobrev, P. Flocchini, and N. Santoro. Distributed exploration of dynamic rings. *Distributed Computing*, 33(1):41–67, 2020.

[16] T. Gotoh, P. Flocchini, T. Masuzawa, and N. Santoro. Tight bounds on distributed exploration of temporal graphs. In *Proc. 23rd International Conference on Principles of Distributed Systems (OPODIS)*, pages 22:1–22:16, 2020.

[17] T. Gotoh, Y. Sudo, F. Ooshita, H. Kakugawa, and T. Masuzawa. Exploration of dynamic tori by multiple agents. *Theoretical Computer Science*, 850:202–220, 2020.

[18] M. Bournat, S. Dubois, and F. Petit. Gracefully degrading gathering in dynamic rings. In *Proc. 20th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 349–364, 2018.

[19] S. Das, G.A. Di Luna, and L.A. Gasieniec. Patrolling on dynamic ring networks. In *Proc. 45th International Conference on Current Trends in Theory and Practice of Informatics (SOFSEM)*, pages 150–163, 2019.

[20] S. Das, G. Di Luna, L. Pagli, and G. Prencipe. Compacting and grouping mobile agents on dynamic rings. In *Proc. 15th International Conference on Theory and Applications of Models of Computation (TAMC)*, pages 114–133, 2019.

[21] G.A. Di Luna, P. Flocchini, L. Pagli, G. Prencipe, N. Santoro, and G. Viglietta. Gathering in dynamic rings. *Theoretical Computer Science*, 811:79–98, 2020.

[22] S. Das, P. Flocchini, N. Santoro, and M. Yamashita. Fault-tolerant simulation of message-passing algorithms by mobile agents. In *Proc. 14th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, pages 289–303, 2007.

[23] A. Casteigts, P. Flocchini, W. Quattrociocchi, and N. Santoro. Time-varying graphs and dynamic networks. *International Journal of Parallel, Emergent and Distributed Systems*, 27(5):387–408, 2012.

[24] F. Harary and G. Gupta. Dynamic graph models. *Mathematical and Computer Modelling*, 25(7):79–87, 1997.

[25] A. Ferreira. Building a reference combinatorial model for *manets*. *IEEE Network*, 18(5):24–29, 2004.

[26] A. Casteigts, P. Flocchini, B. Mans, and N. Santoro. Deterministic computations in time-varying graphs: Broadcasting under unstructured mobility. In *Proc. IFIP International Conference on Theoretical Computer Science (TCS)*, pages 111–124, 2010.

[27] A. Casteigts, P. Flocchini, B. Mans, and N. Santoro. Measuring temporal lags in delay-tolerant networks. *IEEE Transactions on Computers*, 63(2):397–410, 2014.

[28] D. Ilcinkas and A.M. Wade. On the power of waiting when exploring public transportation systems. In *Proc. 15th International Conference on Principles of Distributed Systems (OPODIS)*, pages 451–464, 2011.

[29] P. Flocchini, M. Kellett, P. Mason, and N. Santoro. Searching for black holes in subways. *Theory of Computing Systems*, 50(1):158–184, 2012.

[30] P. Flocchini, B. Mans, and N. Santoro. On the exploration of time-varying networks. *Theoretical Computer Science*, 469:53–68, 2013.

[31] T. Erlebach and J.T. Spooner. A game of cops and robbers on graphs with periodic edge-connectivity. In *Proc. 46th International Conference on Current Trends in Theory and Practice of Informatics (SOFSEM)*, pages 64–75, 2020.

[32] F. Kuhn, N.A. Lynch, and R. Oshman. Distributed computation in dynamic networks. In *Proc. 42nd ACM Symposium on Theory of Computing (STOC)*, pages 513–522, 2010.

[33] F. Kuhn and R. Oshman. Coordinated consensus in dynamic networks. In *Proc. 30th Symposium on Principles of Distributed Computing (PODC)*, pages 1–10, 2011.

[34] B. Haeupler and F. Kuhn. Lower bounds on information dissemination in dynamic networks. In *Proc. 26th International Symposium on Distributed Computing (DISC)*, pages 166–180, 2012.

[35] C. Shannon. Presentation of a maze solving machine. In *Proc. 8th Conference of the Josiah Macy Jr. Foundation (Cybernetics)*, pages 169–181, 1952.

[36] S. Albers and M. Henzinger. Exploring unknown environments. *SIAM Journal on Computing*, 29(4):1164–1188, 2000.

[37] P. Panaite and A. Pelc. Exploring unknown undirected graphs. *Journal of Algorithms*, 33(2):281–295, 1999.

[38] X. Deng and C.H. Papadimitriou. Exploring an unknown graph. *Journal of Graph Theory*, 32(3):265–297, 1999.

[39] P. Fraigniaud, D. Ilcinkas, G. Peer, A. Pelc, and D. Peleg. Graph exploration by a finite automaton. *Theoretical Computer Science*, 345(2–3):331–344, 2005.

[40] P. Fraigniaud, D. Ilcinkas, and A. Pelc. Impact of memory size on graph exploration capability. *Discrete Applied Mathematics*, 156(12):2310–2319, 2008.

[41] R. Cohen, P. Fraigniaud, D. Ilcinkas, A. Korman, and D. Peleg. Label-guided graph exploration by a finite automaton. *ACM Transactions on Algorithms*, 4(4):1–18, 2008.

[42] J. Chalopin, P. Flocchini, B. Mans, and N. Santoro. Network exploration by silent and oblivious robots. In *Proc. 36th International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*, pages 208–219, 2010.

[43] Y. Dieudonné and A. Pelc. Deterministic network exploration by anonymous silent agents with local traffic reports. *ACM Transactions on Algorithms*, 11(2):1–29, 2014.

[44] S. Dobrev, L. Narayanan, J. Opatrny, and D. Pankratov. Exploration of high-dimensional grids by finite automata. In *Proc. 46th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 1–16, 2019.

[45] S. Das. Graph exploration with mobile agents. Chapter 16 of [55], pages 403–422, 2019.

[46] O. Michail and P. Spirakis. Traveling salesman problems in temporal graphs. *Theoretical Computer Science*, 634:1–23, 2016.

[47] C. Avin, M. Koucky, and Z. Lotker. How to explore a fast-changing world. In *Proc. 35th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 121–132, 2008.

[48] I. Lamprou, R. Martin, and P. Spirakis. Cover time in edge-uniform stochastically-evolving graphs. *Algorithms*, 11(10):149, 2018.

[49] G.A. Di Luna. Mobile agents on dynamic graphs. Chapter 20 of [55], pages 549–584, 2019.

[50] A. Agarwalla, J. Augustine, W. Moses, S. Madhav, and A.K. Sridhar. Deterministic dispersion of mobile robots in dynamic rings. In *Proc. 19th International Conference on Distributed Computing and Networking (ICDCN)*, pages 19:1–19:4, 2018.

[51] M. Fukuda, L.F. Bic, M.B. Dillencourt, and J.M. Cahill. Messages versus messengers in distributed programming. *Journal of Parallel and Distributed Computing*, 57(2):188–211, 1999.

[52] L. Barrière, P. Flocchin, P. Fraigniau, and N. Santor. Can we elect if we cannot compare? In *Proc. 15th ACM symposium on Parallel algorithms and architectures (SPAA)*, pages 324–332, 2003.

[53] J. Chalopin, E. Godard, Y. Métivier, and R. Ossamy. Mobile agent algorithms versus message passing algorithms. In *Proc. 10th International Conference on Principles of Distributed Systems (OPODIS)*, pages 187–201, 2006.

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

[55] P. Flocchini, G. Prencipe, and N. Santoro (Eds). *Distributed Computing by Mobile Entities*. Springer, 2019.