ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΕΦΑΡΜΟΣΜΕΝΩΝ ΜΑΘΗΜΑΤΙΚΩΝ ΚΑΙ ΦΥΣΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

Αναζήτηση εχθρικών κομβων σε δίκτυα

ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Ματούλας Πετρόλια

Επιβλέπων: Άρης Παγουρτζής
Επίκουρος Καθηγητής Ε.Μ.Π.

Συνεπιβλέπων: Ευριπίδης Μάρκου
Λέκτορας Πανεπιστημίου Στερεάς Ελλάδας

Αθήνα, Φεβρουάριος 2013

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΕΦΑΡΜΟΣΜΕΝΩΝ ΜΑΘΗΜΑΤΙΚΩΝ ΚΑΙ ΦΥΣΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

Αναζήτηση εχθρικών κομβων σε δίκτυα

ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Ματούλας Πετρόλια

Επιβλέπων: Άρης Παγουρτζής
Επίκουρος Καθηγητής Ε.Μ.Π.

Συνεπιβλέπων: Ευριπίδης Μάρκου
Λέκτορας Πανεπιστημίου Στερεάς Ελλάδας

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή στις 7 Φεβρουαρίου 2013

.....	.....	.....
Ευστάθιος Ζάχος	Άρης Παγουρτζής	Δημήτριος Φωτάκης
Καθηγητής	Επίκουρος Καθηγητής	Λέκτορας Ε. Μ. Π.
Ε. Μ. Π.	Ε. Μ. Π.	

Αθήνα, Φεβρουάριος 2013

ΠΡΟΛΟΓΟΣ

Η παρούσα διπλωματική εργασία εκπονήθηκε στα πλαίσια του διατμηματικού προγράμματος σπουδών 'Εφαρμοσμένες Μαθηματικές Επιστήμες' του Εθνικού Μετσόβιου Πολυτεχνείου.

Στα περιβάλλοντα κατανεμημένων υπολογισμών ένα θεμα που εμπνέει πολλή ανησυχία είναι η ασφάλεια. Από τις πιο σημαντικές απειλές για την ασφάλεια είναι μια στατική επιβλαβής διεργασία που βρίσκεται σε έναν κόμβο του δικτύου. Μια ομάδα κινητών πρακτόρων κινείται μέσα στο δίκτυο με σκοπό να εντοπίσει την ακριβή θέση αυτού του κόμβου.

Σε αυτή τη διπλωματική εργασία θα μελετήσουμε το πρόβλημα του εντοπισμού ενός εχθρικού/κακόβουλου κόμβου μέσα σε ένα δίκτυο. Ο πιο κοινός τύπος κακόβουλων κόμβων είναι η μαύρη τρύπα η οποία καταστρέφει όποιον κινητό πράκτορα φτάνει σε αυτήν χωρίς να αφήνει κανένα ίχνος. Θα παρουσιάσουμε διαφορετικές παραλλαγές του προβλήματος αναζήτησης εχθρικών κόμβων όπως και έναν διαφορετικό τύπο κακόβουλου κόμβου και καποιες αλγοριθμικές λύσεις.

Πιο συγκεκριμένα, στο Κεφάλαιο 1 θα παρουσιάσουμε σύντομα τις κεντρικές έννοιες και ιδέες αυτού του είδους προβλημάτων. Θα αναλύσουμε την ιδέα των κινητώμ πρακτόρων και θα εξηγήσουμε για ποιο λόγο οι πράκτορες χρησιμοποιούνται συχνά στους κατανεμημένους αλγόριθμους. Επιπλέον θα παρουσιάσουμε διάφορες παραλλαγές του προβλήματος καθώς και κάποια παρόμοια προβλήματα.

Στο Κεφάλαιο 2 θα μελετήσουμε το πρόβλημα αναζήτησης μιας μαύρης τρύπας ([FIS12, CDLM11, KMRS07]) παρουσιάζοντας αλγορίθμους για τη λύση καθενός από αυτά με σκοπό να κατανοηθεί η φύση αυτών των προβλημάτων.

Τέλος, στο Κεφάλαιο 3 μελετάμε το πρόβλημα της Περιοδικής Ανάκτησης Δεδομένων σε ένα δακτύλιο όπως παρουσιάστηκε στο [KM10] το οποίο βασιζεται και αυτο στον εντοπισμό ενός κακόβουλου κόμβου με κάποιες πρόσθετες προϋποθέσεις. Για το πρόβλημα αυτό καταφέραμε να αποδείξουμε ότι 4 πράκτορες είναι απαραίτητοι αλλά και επαρκείς για να λύσουν το πρόβλημα στην περίπτωση που η κοινή μνήμη των κόμβων είναι αξιόπιστη, εξαλείφοντας έτσι την απόσταση

ανάμεσα στα προηγούμενα γνωστά κάτω και άνω φράγματα των 3 και 9 πρακτόρων αντίστοιχα.

ΕΥΧΑΡΙΣΤΙΕΣ

Θα ήθελα να ευχαριστήσω όσους συντέλεσαν στο να ολοκληρωθεί αυτή η εργασία και κυρίως τον επιβλέποντα της εργασίας κ. Άρη Παγουρτζή, Επίκουρο Καθηγητή του Ε.Μ.Π., για τη βοήθεια, την εμπιστοσύνη και τη στήριξή του σε όλη τη διάρκεια της εκπόνησής της. Επίσης ευχαριστώ τον κ. Στάθη Ζάχο, Καθηγητή του Ε.Μ.Π. για τις γνώσεις και τις εμπειρίες που μοιράζεται μαζί μας αλλά και τον κ. Δημήτρη Φωτάκη, Λέκτορα του Ε.Μ.Π., για την εμπιστοσύνη του σ΄ εμένα, την κατανόησή του και τη συνέπειά του.

Φυσικά θα ήθελα να ευχαριστήσω τον κ. Ευριπίδη Μάρκου, Λέκτορα του Πανεπιστημίου Στερεάς Ελλάδας, ο οποίος είναι και συνεπιβλέπων της εργασίας μου, για τη μεγάλη βοήθειά του, το ενδιαφέρον και την εμπιστοσύνη που μου έχει δείξει. Επίσης τον Βαγγέλη Μπαμπά για τις πολύτιμες συζητήσεις που είχαμε και τη μεγάλη βοήθεια που μου έχει προσφέρει αλλά και όλα τα παιδιά του Corelab για τις συζητήσεις και τις ωραίες στιγμές, κυρίως τη Χριστίνα, για τη βοήθεια, υπομονή και συμπαράστασή της οποιαδήποτε ώρα του εικοστετραώρου, και τον Δημήτρη.

Τέλος και πιο πολύ θέλω να ευχαριστήσω τη μαμά μου, το μπαμπά μου και τον αδερφό μου για όλα όσα έχουν κάνει για μένα και τους φίλους μου.

ABSTRACT

The current thesis has been elaborated in fulfillment of the thesis requirement for the inter-departmental postgraduate program "Applied Mathematical Sciences" of the National Technical University of Athens.

In distributed mobile computing environments a very pressing concern is security. One of the most important security threats is a stationary harmful process which resides at a host. A team of mobile agents is moving in the network with the task to determine the location of this host.

In this thesis we will study the problem of locating a malicious node in a network. One of the most common types of malicious hosts is a black hole that destroys any incoming mobile agent without leaving any trace. We will present different settings of the black hole search problem problem as well as a different type of malicious host and some algorithmic solutions.

More precisely, in Chapter 1 we will present shortly the main concepts and notions of these kinds of problems. We will analyze the agent paradigm and we will explain why agents are very frequently used in distributed algorithms. Moreover we will present different settings and variations of malicious-host-search problems as well as some kinds of similar problems.

In Chapter 2 we will study the Black Hole Search problem in various settings ([FIS12, CDLM11, KMRS07]) presenting algorithmic solutions to each one of them in order to understand the nature of this kind of problems.

Lastly, in Chapter 3 we study the Periodic Retrieval problem in a ring presented in [KM10] which is also based on locating a malicious host in a network with some additional requirements. We have been able to prove that 4 agents are necessary and sufficient to solve the problem in the case of reliable whiteboards, closing the gap between the previously known lower and upper bounds of 3 and 9, respectively.

Contents

Chapter 1

Modeling the problem

1.1 Introduction

The problem of exploring an unknown environment is a fundamental problem with applications ranging from robot navigation to searching the World Wide Web. As such, a large body of work has focused on finding efficient solutions to variants of the problem with restrictive assumptions on the form of the environment.

A natural way to model the problem is by a robot (mobile agent) exploring a graph. The case in which the graph has undirected edges and labeled vertices can be solved in time linear in the number of edges by depth first search. Other search techniques [PP99] improve this bound by a constant factor. But often many exploration problems do not satisfy these constraints and the problem may be represented by a graph with directed edges. The problem with directed edges and labeled vertices can be solved by a greedy search algorithm in time $O(|V| \cdot |E|)$. More sophisticated techniques [AH00, DP99] give improved running times.

Mobile agents are a powerful tool for implementing distributed applications in computer networks and have been extensively used for graph exploration problems in which the agents have to search through the graph collecting information and interacting with each other. An agent is usually modeled as a finite automaton and the first known finite automaton algorithm designed for graph exploration was introduced by Shannon in 1951 ([FIP+04]).

Many variants of the problem have been studied focusing on different aspects of the setup (specific topologies, type of graphs, etc.) and the complexity measure (number of agents used, performance, efficiency, etc.).

However, these problems assume that the network and the nodes are safe.

Of course, this might not be true as the network might be faulty. The nature of faults can vary from faulty links to malicious hosts. In this thesis we consider security problems of the latter type, i.e. nodes which may harm agents in various ways (killing agents, changing the agents' status, deleting or changing stored data etc.). In order to solve such a problem we should find a way to determine the location of the harmful node and report it, if possible.

1.2 Mobile agents

The paradigm of mobile agent computing has seen use in fields as diverse as artificial intelligence, computational economics and robotics. Agents bring potential advantages in terms of efficiency, fault-tolerance, flexibility and simplicity. We will study mobile agents as modelled in distributed systems research and in particular within the framework of research performed in the distributed algorithms community. In this literature, the agents are generally modelled as automata that move on a network modelled as a graph.

Mobile agents can reduce the connecting time and bandwidth consumption by processing the data at the source and sending only the relevant results. By moving the agents to data-residing hosts, they can reduce communication costs.

A mobile agent in general has the following properties [KKM10a, CD12]:

- *Autonomy*: Mobile agents should work with some degree of independence from their creator and take control over their actions. They should be able to execute, move and make some decisions without supervision.
- *Mobility*: The agents must have the ability to move from one node to another in a distributed system. Note that when such an agent moves it is assumed that it carries its identity, execution state and program code so that it can resume its execution in the destination node after the move.
- *Interactivity*: An agent must be able to interact with its environment and with the other agents (if there are any). In most cases this is likely to be cooperative behavior but sometimes it is possible to be competitive behavior. The exact form of interaction depends on the model of the system.
- *Adaptability*: The ability of an agent to adapt to new situations and learn from previous experience increases its usefulness.

The advantages that the mobile agents offer over the traditional distributed computing approaches include among others [KKM10a]:

- *Efficiency*: Mobile agents offer potential savings in both latency and network bandwidth due to their compactness. In a situation where n sites must be visited in sequential order, where for instance the output from one site is used as part of the input to the next, a mobile agent can perform the task by moving along an n edge cycle incurring the cost of n communication steps whereas a centrally located agent would need $2n$ communication steps (to and from each site). In a situation where parallel access to the sites is possible, then a team of mobile agents can visit all of the sites faster than a single stationary agent.

- *Fault-tolerance*: In situations where a user has limited connectivity to the network, a mobile agent may overcome this deficit by acting on behalf of the user during blackout periods and returning useful information when connectivity returns. In situations where nodes may go down on a regular basis with limited notice, a mobile agent can potentially move to another node and continue operating.

- *Flexibility*: It is generally easy to add features to agents that allow them to adapt their behavior to new conditions. More sophisticated agents may be designed to incorporate learning from past experience.

- *Ease of use*: In many situations, it is natural for the programmer to imagine that they are dealing with autonomous agents. This makes the systems design and implementation easier to perform.

As mentioned above, mobile agents are usually modeled using a finite automaton consisting of a set of states and a transition function which takes as input the agents current state as well as the state of the node it resides in and outputs a new agent state, possible modifications to the current nodes state and a possible move to another node. We may also consider an agent to be a probabilistic automaton (*randomized* agent).

When studying a problem it is important to consider whether or not the agents have distinct labels or identities. When the agents are indistinguishable they are called *anonymous* agents. Anonymous agents are limited to running exactly the same program, i.e. they are identical finite automata. Agents with distinct identities may run different programs.

1.3 Network

An important factor that affects the problem's difficulty is the network that is about to be explored. The network is usually modeled by a graph whose vertices represent the computing nodes (or hosts) and edges represent the communication links.

The knowledge the agents have about the network can make a difference in the solvability and efficiency of many problems. For example, knowledge of the size of the network or its topology may be used as part of the agents program. If available to the agents, this information is assumed to be part of their starting state or the information is made available by the nodes of the network.

Like mobile agents, the nodes of a network may also be distinguishable or *anonymous*, i.e. the nodes have no identities. This means that an agent can not distinguish two nodes except perhaps by their degree. The outgoing edges of a node (ports) are usually considered distinguishable but an important distinction is made between a globally consistent edge-labeling versus a locally independent edge-labeling. For example, in a ring the ports may be consistently marked as clockwise and counterclockwise or they may be arbitrarily marked. If the labeling satisfies certain coding properties, it is called a *sense of direction* [FMS98] and it greatly affects the solvability and the efficiency of solution of many problems in distributed computing.

Another classification of the networks is made by how they deal with time. The network may be *synchronous* or *asynchronous*. In a synchronous network there exists a global clock available to all nodes and it is used by the agents as well. It is also assumed that the agents perform their tasks in synchronization and that it takes one time unit to take a step. In an asynchronous network the time needed for an agent to move from one node to another is not bounded but it is still finite.

Since the networks we are studying in this thesis are assumed to be unsafe, there are many variations depending on the kind of the fault they present but we will talk about the variety of faults in 1.5.

1.4 Communication

Most of the times to locate a malicious host in a network we need more than just one agent. Thus, it is important to consider how the agents interact and the kind of the communication mechanism they use each time is an important aspect of any model. One might consider the case where the agents are able to detect the presence of other agents at a node and possibly read their state.

In general, all nodes are assumed to provide enough space to store the agent temporarily and computing power for it to perform its tasks but in some cases we may consider nodes that provide some form of a long-term storage which may be shared among all agents visiting the nodes. This service is called a *whiteboard* and it is one of the communication mechanisms that have been proposed and have been mostly studied in the literature:

- the *whiteboard* model: there is a whiteboard at each node of the network where the agents can leave messages,
- the *token* model where the agents carry tokens which they can leave at nodes ("pure" tokens) or at nodes and edges ("enhanced" tokens), and
- the *time-out* mechanism (only for synchronous networks) in which one agent explores a new node and then (after a pre-determined fixed time) informs another agent who waits at a safe node.

The whiteboard model is the most powerful of these mechanisms not only because of the ability that it gives to agents to exchange messages but also because of the mutual exclusion mechanism that is used in order for an agent to have access. If the agents start at the same node, due to mutual exclusion, they can get distinct identities and then assign different labels to all nodes. Hence in this model, if the agents are initially co-located, both the agents and the nodes can be assumed to be non-anonymous without any loss of generality[M$^+$12]. The whiteboard model has been studied in asynchronous networks for initially collocated (e.g. [DFPS06, DFPS01, BFMS10]) or scattered agents (e.g. [FKMS09b]) while synchronous networks do not seem to need it.

The token model has mostly been used in safe network explorations. Except the pure and enhanced tokens we also may have *movable* and *unmovable* tokens. Movable tokes are those which can be placed at a node and then an agent can pick them up and move them somewhere else while the unmovable ones are the ones that once placed they cannot be moved. The token model has also been studied in both synchronous and asynchronous networks (e.g. [FIS12, BFR$^+$02, CDLM11]).

Other properties that may be considered include mechanisms for sending and/or receiving messages via message passing, whether or not the agents have the ability to clone themselves, i.e., produce new identical copies of themselves and whether or not they have the ability to "merge" upon meeting.

1.5 Malicious hosts

There are different types of malicious nodes in a graph which represent various kinds of malicious/faulty hosts in a network. The most common and most studied type of malicious host is a *black hole*. A black hole is a node that contains a stationery process which destroys anything that reaches that node (hence the agents as well) without leaving any trace.

An alteration to the black hole concept is the *grey hole* which is a node that may alter its behaviour between the behaviour of a black hole and that of a safe node. This means that a grey hole may destroy an agent which resides on it, without leaving any trace, or it may let the agent perform all its tasks and computations and even move to another node safely, as if it were a completely safe node.

Except these two types there are a lot of other variations since, in reality, a host has many ways to harm the agents: it may not only kill any agent residing in it at any time, it may also duplicate agents, introduce fake agents, tamper the runtime environment (e.g. changing the contents of the whiteboard), or disobey communication protocols (e.g. do not execute agents in FIFO order) etc.

1.6 Other variations

Networks containing malicious hosts may be the most common scenario in network security problems but other scenarios have been studied as well, not that extensively though.

The case of black links in arbitrary networks has been studied in [CDS07, FKMS09a], respectively for anonymous and non-anonymous nodes providing non-optimal solutions in the case of multiple black holes and links in [CDS07], and conditions for solvability in [FKMS09a]. All the previous literature considers black holes in undirected graphs; the case of directed graphs has been recently investigated in [CDK$^+$10], where it is shown that the requirements in number of agents change considerably.

1.7 Similar problems

Apart from the malicious host search problems there are many other problems that are somehow related and can be solved using mobile agents and similar techniques.

The *rendez-vous* problem it the one studied the most and it is the problem of how to achieve a rendez-vous, the gathering of two or more agents

at the same node of a network. Of course, as above, the problem can be studied under various models ([KMP08, KKM11, KKM10b, KKM06]). Such an operation is a useful subroutine in more general computations that may require the agents to synchronize, share information etc. For example, in a black hole search problem it might be convenient to make the agents gather at one certain node so that they can work as collocated agents.

Another problem analogous to the previous ones is the *firefighter* problem. The firefighter problem models the situation where an infection, a computer virus, a fire etc. is spreading through a network and the goal is to save as many nodes of the network as possible through targeted vaccinations. The number of nodes that can be vaccinated at a single time-step is typically one, or more generally O(1). In a non-standard model, the so called spreading model, the vaccinations also spread in contrast to the standard model[FLP11].

Since searching for a piece of information is one of the most common tasks in a distributed environment, the problem of locating a user or an item in a network is also studied ([HKKK08, KKS08]). A mobile agent tries to find it doing queries to the nodes. The question is of the following form: "when did you see it for the last time?" or "how do I get to it?", respectively. And the answer is of the form: "k time units before" ($k = \infty$ if the user was never visited the node, and $k = 0$ if the user is right now at the node) in the case of searching for users or, in the other case, the node may answer with an edge on the shortest path to the item. A node may answer truthfully or it may lie; in the latter case we call this node a *liar*. In some cases, there is no liar among the nodes.

Chapter 2

The Black Hole Search Problem

2.1 Black Hole Search in Asynchronous Rings

First we will present the Black Hole Search Problem in an asynchronous ring with mobile agents endowed with one token.

Note that in an asynchronous ring it is impossible to determine whether or not there is a black hole in the ring as it is impossible for an agent to distinguish between a slow link and a black hole. Thus the existence of a black hole in the ring must be common knowledge. For the same reason, even if the existence of a black hole is known to the agents, it is impossible to find the black hole without knowing the size of the ring.

The optimal algorithm for this model is presented in [FIS12].

2.1.1 Network model and problem definition

The network in this model is a ring whose size n is known to the agents. At each node there is a distinct label associated to each of its incident links.

All agents start from the same node of the network (*homebase*) and have a complete map of the graph (in this case they know it is a ring and the size of it). Every agent has a *token*; all tokens are identical. An agent can carry a token, put it down at a node (if no other token is already there) and pick it up from a node (if it does not carry another token). An agent entering a node can see if there is a token there but it might not be able to see other agents or to see whether they carry a token or not.

The network is asynchronous. This means that each agent can enter the system at an arbitrary time, traversing an edge may take a finite but otherwise unpredictable amount of time and an agent might be idle at a node for an upredictable but still finite amount of time.

The basic computational step of an agent (executed either when the agent arrives to a node, or upon wake-up) is to look for a token at the node, drop or pick up the token, if wanted, and leave the node through some chosen port. The whole computational step is performed in local mutual exclusion as an atomic action, i.e. as if it takes no time to execute it.

Links are not assumed to be FIFO, i.e. two agents traversing the same edge to the same node may arrive in an arbitrary order.

It is also assumed that the transition between two states of the agent and the corresponding move are instantaneous, i.e. there is no delay due to asynchrony before the move of the agent. Furthermore, it is assumed that the actions of the agents at different nodes occur at different instants.

The network contains exactly one black hole. The problem is to locate the black hole, i.e. at least one agent must survive and know the location of the black hole.

It is also assumed that the clockwise direction is the same for both agents. An agent exploring to the right (clockwise) is said to be a *right* agent and an agent exploring to the left (counterclockwise) is said to be a *left* agent. An agent might *change role*, i.e. become a right agent when it was a left and vice versa.

The two measures of complexity of a solution protocol are the number of agents used to locate the black hole and the total number of moves.

2.1.2 The algorithm

Firstly, we will describe the main moves of the agents. We say that a right agent traverses an edge u, v from u to (right neighbor) v using *cautious walk* when it has one token, drops it at u, moves to v, comes back to u, picks up the token and goes to v. A left agent traverses an edge u, v from u to (left neighbor) v using *double cautious walk* when it has one token while the other is at u, goes to v, drops its token at v, comes back to u, picks up the other token and goes to v again. So, In the first phase, exploration to the right is always done using cautious walk, while exploration to the left is always done using double cautious walk.

For $i \geq 0$, a node at distance i to the right of the homebase will be called node i and a node at distance i to the left of the homebase will be called node $-i$.

The idea of the *Ping-Pong* algorithm is that one agent explores the right side of the ring and the other explores the left side. However, only one agent at a time is allowed to explore, thus the agents cannot make progress simultaneously in two different directions.

Now, suppose we have two agents, A and B. The algorithm has two phases.

Phase 1:

- Initially both agents explore to the right using cautious walk.
- At some point eventually, one agent (say A) will find the token of agent B at a node.
- A wants to steal B's token so it goes to the previous node and comes back to pick up B's token.
  - If, in the meantime, B has picked its token and moved on, then A resumes its cautious walk in the same direction.
  - Otherwise, A steals B's token and becomes a left agent, starting to explore left using double cautious step.
- If B comes back and finds out that its token has been stolen, it goes left until it finds a token, then picks it up, returns to the last node it had visited and resumes exploration to the right.
- If A realizes that one of its tokens has been stolen, it changes role and explores to the right using cautious walk.

The first phase of the algorithm is already correct by itself but Phase 2 is used to decrease the number of moves. Phase 2 happens when at least two nodes have been explored to the right.

Phase 2:

- If a right agent A finds the token of right agent B at a node p with $p \geq 2$ then steals it, returns to node $p-1$, ignores the token A had released there and starts cautious walk to the left, becoming a left agent.
  - If agent B returns at node p to pick up its token and does not find it there, it starts moving to the left and finds a token it at node $p-1$. That indicates that Phase 2 has started and that the remainig unexplored nodes have been divided in two groups: A explores the left side and B explores the right side. Thus, B starts cautious walk exploring to the right.

- If agent B has not been back to node p before A finishes exploring its part, then A computes the new workload and divides it again in two parts. Then starts exploring again its part. This may happen more than once since B might be blocked or vanished.
- If the latter happens more than once, then if B goes back to node p to pick up its token and does not find it there, again it starts moving to the left until it finds a token. Now the number of steps it takes until it finds one indicate the number of times that the above case happened so B can compute the current unexplored part and thus its current workload.

Obviously, Phase 2 may never be executed in some cases. In any case though, when only one node remains unexplored, this remaining node is the black hole and the algorithm terminates.

2.1.3 Bounds

As mentioned above, the first part of the algorithm (Phase 1) solves the problem by itself but the number of moves can be $\Theta(n^2)$ in the worst case. It is proved that Phase 2 decreases the worst case number of moves to $O(n\log n)$ which is the optimal bound [DFPS01]. This algorithm is also optimal in the sense that it cannot be solved with fewer agents and clearly not with fewer tokens.

Another very interesting result of this paper is the proof that the token model is computationally as powerful as the whiteboard model (for networks of known topology) since the same optimal bounds are obtained for number of agents and number of moves.

2.1.4 Modified algorithm for arbitrary graphs

The algorithm we described above can be modified so that it works for arbitrary networks. It is interesting that the same bounds as before are obtained.

The idea that allows that algorithm to be applied to general graphs is to find a cyclic ordering of the network nodes, i.e. a mapping between the node numbers and the actual nodes on the network in a way that allows an agent to know what means "go left" and "go right" at any point.

Since the network does not necessarily contain an Hamiltonian cycle, a single edge in the "virtual" ring may correspond to a longer path in the actual network. In order not to increase too much the move complexity, the cyclic ordering of the network nodes is constructed by following a DFS traversal of some spanning tree avoiding the current suspicious node (the node where the

other agent is apparently blocked). Since the location of the suspicious node changes throughout the execution of the algorithm, so does the spanning tree and thus the cyclic ordering. (In fact, only the ordering of the unexplored nodes may change.) This is done in such a way that both agents agree on the cyclic ordering when it is necessary.

2.2 Black Hole Search in Tori

In [CDLM11] the BHS problem is examined under some different settings. First of all the network is a torus and it is synchronous. The agents use tokens as well and are initially scattered in the network.

2.2.1 Network model and problem definition

The model consists of $k \geq 2$ anonymous and identical agents, placed at distinct nodes of an anonymous, synchronous torus of size $n \times m$, $n \geq 3$, $m \geq 3$. The torus is also oriented in the sense that at each node the four incident agents are consistently marked as North, South, East and West. Each mobile agent owns a constant number of t identical tokens which can be placed at any node visited by the agent. A node may contain at most three tokens at the same time and an agent carries at most three tokens at any time. A token or an agent at a given node is visible to all agents on the same node, but is not visible to any other agent. The agents follow the same deterministic algorithm and begin execution at the same time and being at the same initial state.

At any single time unit, a mobile agent occupies a node u of the network and may 1) detect the presence of one or more tokens and/or agents at node u, 2) release/take one or more tokens to/from the node u, and 3) decide to stay at the same node or move to an adjacent node.

All computations by the agents are independent of the size $n \times m$ of the network since the agents have no knowledge of n or m. There is exactly one black hole in the network. An agent can start from any node other than the black hole and no two agents can start from the same node. Once an agent detects a link to the black hole, it marks the link permanently as dangerous (i.e., disables this link). Since the agents do not have enough memory to remember the location of the black hole (an agent is a finite automaton), it is required that at the end of a black hole search scheme, all links incident to the black hole are marked dangerous, that none of the other links of the network is marked as dangerous and that there is at least one surviving agent. This is a slightly different definition for a succesful BHS scheme. The

time complexity of a BHS scheme is the number of time units needed for completion of the scheme, assuming the worst-case location of the black hole and the worst-case initial placement of the agents.

2.2.2 Bounds

Before we describe the algorithms that solve the problem, we will present some results which provide lower bounds for the Black Hole Search problem in a torus.

First of all, the following theorem is proved which tells us that we will only need to consider movable tokens for this problem.

Theorem 2.2.1. *No algorithm can solve BHS in all oriented tori containing one black hole and k scattered agents, where each agent has constant memory and t unmovable tokens, for any constant numbers k, t.*

The next lemma gives a lower bound for the number of agents, despite of the number of tokens they may have.

Lemma 2.2.2. *Two agents carrying any number of movable tokens cannot solve the BHS problem in an oriented torus even if the agents have unlimited memory.*

The proof of lemma 2.2.2 is based on the fact that the adversary may decide which nodes the agents will start from and also the location of the black hole in the torus.

Without loss of generality we may assume that the first move of the agents is East. The idea is that the adversary may place the black hole at the East neighbor of an agent so that it vanishes into the black hole right after its first move and also place the second agent such that it vanishes into the black hole right after its first vertical move.

Thus, for any algorithm the number of agents in order to solve the problem must be at least three.

The next lemma gives a lower bound on the number of tokens each agent carries.

Lemma 2.2.3. *There exists no algorithm that can solve the BHS problem in all oriented tori using three agents with constant memory and one movable token each.*

Thus, for an algorithm using three agents, each agent must have at least two tokens.

From lemmas 2.2.2 and 2.2.3 the next theorem can be deduced.

Theorem 2.2.4. *At least three agents are required to solve the BHS problem in an oriented torus of arbitrary size. Any algorithm solving this problem using three agents requires at least two movable tokens.*

The main algorithm presented in the paper is an optimal algorithm since it matches the lower bounds for the number of agents and number of tokens per agent mentioned above (i.e. it solves the BHS problem in a torus using three agents and two tokens per agent).

2.2.3 The algorithms

Three algorithms are presented, the first one being a really simple one and the third one being really involved. We will try to describe them as well as possible. All algorithms use obviously only movable tokens and the main technique used is the *Cautious Walk*.

Algorithm BHS-torus-33

The first algorithm we are going to describe is the simplest one which needs three agents provided with three tokens each.

- An agent leaves one token on its starting node (*homebase*) and moves East performing cautious walk with two tokens (i.e. puts down two tokens, goes one step East, goes back, picks up both tokens and goes one step East).
- An agent repeats cautious walk in the East direction until it meets a node containing one token twice. (A node containing one token is a homebase either of this agent or of another agent.)
- After the agent meets two homebases it performs a cautious walk in the South direction with three tokens (it picks up the homebase token as well)
- If it survives and the node it reaches contains, it repeats cautious walk in the South direction with two tokens until it reaches an empty node. This empty node is its new homebase and it repeats the same process.
- If during its cautious walk the agent meets a node with two tokens then it concludes that the black hole is the neighboring node to East.
- If during its cautious walk the agent meets a node with three tokens then it concludes that the black hole is the neighboring node to South.

- In any of the last two cases the agent stops its execution and traverses a cycle around the black hole marking all links leading to it as dangerous.

The idea in this algorithm is that an agent may visit an unexplored node only while going East or South. Thus, if one agent enters the black hole going East (resp. South), there will be two (resp. three) tokens on the previous node and thus, no other agent would enter the black hole through the same link. Therefore, at least one agent always survives. Whenever an agent meets two or three tokens at the end of a Cautious-Walk, the agent is certain of the location of the black hole since any alive agent would have picked up its Cautious-Walk tokens in the second step of the cautious walk.

Note that this algorithm solves the problem if there are $k \geq 3$ agents. This means it would work even if we had more than 3 agents though we will see that this is not always the case.

Algorithm BHS-torus-42

We will now present the second algorithm which uses only two tokens per agent but on the other side it needs at least four agents to solve the problem. But first we need the following definition.

Definition 2.2.5. *If there is node v such that there are exactly two tokens at its North neighbor and exactly two tokens at its West neighbor, then we say that there exists a* Black-Hole-Configuration *at v.*

The agent puts down both its tokens on its homebase and performs a cautious walk in the East direction. If it survives, it returns, picks up one of the tokens and repeats the cautious walk with one token in the East direction until it reaches a node with one or two tokens.

- If it reaches a node with two tokens then the black hole is either the neighbor on the East or the neighbor in the South. Then the agent stops its execution and checks whether the black hole is the East or the South neighbor.

- If it reaches a node with one token then it performs a cautious walk in the East direction with two tokens and then continues the cautious walk with one token in the East direction. If it reaches a node with one token three times then it performs a cautious walk with two tokens in the South direction. If the node it reaches contains a token, it repeats cautious walk with two tokens to the East until it reaches an empty node. Then it repeats the same exploration using that empty node as its homebase. Whenever it reaches a node with two tokens its stops an

checks whether the black hole is the East or the South neighbor of that node.

Now we will describe how that checking process happens. Suppose that the agent has reached a node u which contains two tokens and it wants to check whether the black hole is the East neighbor v or the South neighbor w. The agent visits the West neighbor of w, x (with two steps and waiting one step in between). If it finds two tokens on x then w is te black hole. If x is empty or contains one token, it performs a cautious walk to w with one or two tokens respectively. If it survives then the black hole is at v. Otherwise it will vanish into the black hole leaving a Black Hole Configuration at node w which was apparently the black hole. An agent that discovers the location of the black hole traverses a cycle around it and, as in the previous algorithm, it marks all incident links as dangerous.

Note that the waiting step during the checking process happens due to synchronization reasons so that all surviving agents have picked up their cautious walk tokens. Also, it is easy to see from the description of the algorithm that at most three agents can enter the black hole: at most one agent going South (and thus leaving two tokens at the North neighbor of the black hole) and at most two agents going East (each leaving one of its tokens at the West neighbor of the black hole). Therefore, at least one agent will survive.

Also, we can observe that after an agent starts exploring a horizontal ring, one of the following will happen: if the ring is safe then the agent will move to the next horizontal link below or if the ring is not safe either all agents will vanish into the black hole or one of them will mark all the links incident to the black hole as dangerous. This property concludes the corectness of the algorithm described.

Lastly, note again that this algorithm may use $k \geq 4$ agents to solve the problem.

Algorithm BHS-torus-32 (Optimal)

In [CDLM11] there is finally presented an algorithm which matches both lower bounds proven earlier (i.e. for agents and for tokens per agent) and therefore is an optimal one. This algorithm we are about to roughly describe uses 3 agents and each agent carries 2 tokens.

This algorithm contains a lot of technical details and is quite complicated thus we will only present the main ideas. The techniques used in the previous algorithms are now used again here.

The main and most important idea of this algorithm is to make two agents meet when they are relatively close to each other. If two agents meet then

the problem is easy to solve using a very simple technique we will describe later.

Before we start describing the algorithm let us redefine the notion of the Black-Hole-Configuration (BHC) for the purposes of this algotithm.

Definition 2.2.6. *If there is node v such that there is one or two tokens at both the West and the North neighbors of v, then we say that there exists a* Black-Hole-Configuration *at v.*

The two main procedures that the algorithm uses are *First-Ring* and *Next-Ring*:

- **First-Ring** is only used once, when the algorithm starts. With this procedure each agent explores only the first horizontal ring (i.e. the horizontal ring that contains their starting location) moving in the East direction and using a cautious walk with one token (the second token is left on the agent's homebase).
  - If it enters a node that contains two tokens then the next node is the black hole.
  - If it enters a node containing a single token then this is a homebase (not necessarily its own). In this case it leaves its other token there and continues without tokens.
    - If it finds another node containing one single token then the next node is the black hole.
    - If this is not the case then the agent keeps travelling to the East until it meets six times a node containing two tokens and the procedure terminates, the agent being at a node with two tokens and ready for the Next-Ring procedure.
- **Next-Ring** is used every time an agent has finished the exploration of an horizontal ring, to explore the next ring. The agent starts from a node with two tokens, leaving one on the homebase and using the other one it performs a cautious walk to the South. Thus, it checks the node below and goes back to its homebase. Then it moves to the East direction with one token and again performs a cautious step to the South and keeps doing this until it meets a node with one token on the safe ring. In this case the agent does not go South but instead it goes West and South and
  - if it meets a token there then we have a BHC.

– if there is no token there, it leaves a token and goes East. If the black hole was there, then the agent has left a BHC.

If it survives, it goes back to its exploration, doing the same check every time it encounters a node with one token on the safe ring. When it has met six such nodes, the procedure terminates.

An additional procedure called *Init-Next-Ring* is used every time an agent is about to start Next-Ring, in order to choose its new homebase for exploring next ring. Without this particular procedure there would be a possibility that the homebases of two agents would form a BHC. Actually, if this was the case (i.e. two agents' tokens form a BHC) then this procedure ensures that the two agents meet, thus they can solve the problem on their own.

If two agents manage to meet then they can locate the black hole by executing *BHS-with-collocated-agents*, a procedure that is really simple and is based on the time-out mechanism. The first thing they do once they meet at a node is pick up all their tokens from the nodes they had left them. When both agents have their tokens, one agent becomes the leader and the other one becomes the follower (breaking symmetry at this point is easy to do since they arrive at the node from a different direction). Then they perform a special Cautious-Walk:

- First they leave 3 tokens at the node they start from
- The leader goes to the next node in the East direction while the follower waits at the current node. Then the leader comes back and they move together to the now explored node and continue this process.
- When they meet their 3 tokens, the leader goes one step South while the follower waits and then comes back and they move together with their tokens to the ring below and start exploring it as above.
- If at some point the leader does not return within two time units, the follower knows exactly where the black hole is.

Note that only the leader may be killed falling into the black hole. Also note that apart from this procedure, there is no other part of the algorithm where we may have a node with 3 tokens, thus if the third agent comes across a node containing three tokens it knows that the other two agents have met and stops executing the algorithm. Similarly, if the two agents meet the third one while executing BHS-with-collocated-agents, the third agent stops executing the algorithm.

The algorithm obviously contains many more procedures for different technical details which we will not describe here as they are very involved and we wanted to just give the main ideas of it.

Let us note at this point that although the algorithm BHS-Torus-32 is an optimal algorithm regarding the number of agents needed, there is one downside to it: it only works with 3 agents. That means that BHS-Torus-32 will not correctly locate the black hole if the agents are more than 3. If we combine this algorithm with algorithm BHS-Torus-42 then we can solve the problem for any $k \geq 3$, k the number of agents. But the number k has to be known in advance since these two algorithms cannot be combined to give an algorithm for solving the problem for $k \geq 3$ without the knowledge of k.

Hence, it remains an interesting question whether there exists an algorithm that solves the problem for $k \geq 3$ without knowing k in advance. Another interesting question would be to determine the number of agents required to solve the problem carrying only one token each.

2.3 Black Hole Search in Arbitrary Networks

An algorithm for solving the Black Hole Search Problem in arbitrary networks was presented in [KMRS07].

2.3.1 Network model and problem definition

The model consists of a connected, undirected graph with no multiple edges and no self loops. The network is synchronous and there is at most one black hole in it. There are two agents initially collocated on the starting node s. The only node known to be safe is the starting node. The agents can communicate only when they are in the same node.

Now to define the BHS problem, suppose we have a connected undirected graph $G(V, E)$ and a node $s \in V$. An *exploration scheme* $\mathcal{E}_{G,s} = (\mathbb{X}, \mathbb{Y})$ for G and s consists of two equal-length sequences of nodes in G, $\mathbb{X} = \langle x_0, x_1, \ldots, x_T \rangle$, $\mathbb{Y} = \langle y_0, y_1, \ldots, y_T \rangle$. The measure is the cost of the BHS based on $\mathcal{E}_{G,s}$ which is the worst execution time over all possible configurations. The objective is to find an exploration scheme which minimizes the cost of the BHS based on it.

Agent-1 and *Agent-2* follow the paths defined by $\mathbb{X}$ and $\mathbb{Y}$, respectively. When an agent deduces the existence of a black hole and its exact location, aborts the exploration and returns to s traversing only safe nodes. The exploration is deterministic and the scheme is calculated before the exploration

starts. Then, the agents must follow the sequences until one realizes that the other has died.

Also, note that a node may have been visited by an agent but becomes explored only when the agents meet and that the explored territory is defined for an exploration scheme $\mathcal{E}_{G,s}$, not for the BHS based on it. that is, it doesn't take into account the possible existence of a black hole.

The exploration scheme must satisfy these constraints to guarantee the feasibility of it:

1. Both agents start from the same node s and end at the same node.
2. During each step, an agent can either wait in the node v where it was, or move to a node adjacent to v.
3. Each node in V is visited at least once by at least one agent during the exploration.
4. During each phase, an agent can visit at most one unexplored node (a) and the two agents cannot visit the same unexplored node during the same phase (b).

Using these constraints we can observe that the first node of the exploration is always the starting node s and that at the last step of the exploration scheme the explored territory contains all the nodes of the network.

Now it is easy to see that each phase of the exploration scheme must have length at least two (otherwise the fourth constraint would be violated). There two kinds of such phases that can be used in exploration schemes and that increase the explored territory by two nodes. If m is the meeting point at step j, *Agent-1* and *Agent-2* visit unexplored nodes adjacent to m, v_1, v_2 respectively, at step j+1. Then, at step $j+2$ the agents meet:

- in m (**b-split**(m, v_1, v_2)), or
- in a node $m' \neq m$, adjacent to v_1, v_2 (**a-split**(m, v_1, v_2, m')).

Finally, in the case that there is no black hole in the network, the execution time is defined to be the lenght of the exploration scheme $\mathcal{E}_{G,s}$ plus the shortest path from the last node of the exploration scheme to s, otherwise the execution time is the length of the part of $\mathcal{E}_{G,s}$ the agents have executed until one falls into the black hole plus the shortest path from the last meeting point to s.

An agent discovers the location of the black hole using a time-out mechanism. If the second agent doesn't show up at a meeting point then it know the exact location of the black hole as the agents visit only one unexplored node during a phase each. Then the surviving agent returns to s.

2.3.2 NP-hardness of BHS

The BHS problem in planar graphs is NP-hard. To prove this, a reduction from a specific version of the Hamiltonian cycle problem to the decision version of the BHS problem is used.

Hamiltonian cycle problem for cubic planar graphs (cpHP)

Instance: a cubic planar 2-edge connected graph $G(V, E)$ and an edge $(x, y) \in E$.
Question: does G contain a Hamiltonian cycle that includes edge (x, y)?

The cpHP problem without the requirement that the Hamiltonian path passes through a given edge is NP-complete and so this one is also NP-complete because of a simple reduction.

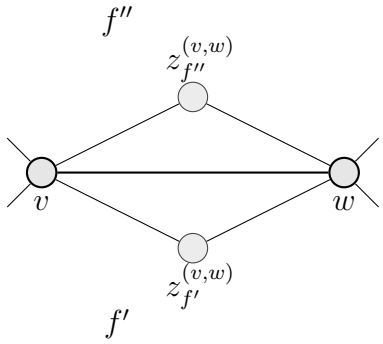
Decision Black Hole Search problem for planar graphs (dBHS)

Instance: a planar graph $G' = (V', E')$, a starting node $s \in V'$ and a positive integer X.
Question: does there exist an exploration scheme $\mathcal{E}_{G',s}$ for G' starting from s, such that the BHS based on $\mathcal{E}_{G',s}$ has cost at most X?

For the reduction from cpHC to dBHS, let $G = (V, E)$ and (x, y) an instance for cpHC. We construct the corresponding instance of the dBHS problem following the steps described below.

1. Replace edge (x, y) in G with edges (x, s), (s, y), s a new node. Get graph $\overline{G}$.
2. $\mathcal{F}$ the set of the faces of $\overline{G}$. Identify each $f \in \mathcal{F}$ with the sequence of the consecutive edges adjacent to f.
3. For each f and each edge (v, w) adjacent to f, add a new node $z_f^{(v,w)}$ and two edges $(v, z_f^{(v,w)})$, $(w, z_f^{(v,w)})$.
4. For each $f = \langle e_1, e_2, \ldots, e_3 \rangle \in \mathcal{F}$, add the shortcut edges $(z_f^{e_1}, z_f^{e_2})$, $(z_f^{e_2}, z_f^{e_3})$, $\ldots$, $(z_f^{e_q}, z_f^{e_1})$.
5. For each $v \in V \cup \{s\} \setminus \{x\}$ add a new node v^F (*flag node*) and edge (v, v^F).

6. Obtain G'. Set $X = n' - 1 = 5n + 2$, where $n' = n + 1 + 2(e + 1) + n = 5n+3$ the number of nodes in G', n, e nodes and edges in G respectively ($e = \frac{3}{2}n$).



Note that each edge e in $\overline{G}$ is adjecent to exactly two faces f', f''. The nodes $z_{f'}^{e}$, $z_{f''}^{e}$ added are called *twin nodes* for e. Also, G' is planar and can be constructed in linear time. The nodes in G' inherited from $\overline{G}$ are called *original nodes*.

The following Lemmas prove that graph G has a Hamiltonian cycle passing through edge (x, y) if and only if there is an exploration scheme for G' and the starting node s with cost at most $X = 5n + 2$.

Lemma 2.3.1. *If G has a Hamiltonian cycle that includes (x, y), then there exists an exploration scheme $\mathcal{E}^*_{G',s}$ on G' from s such that the BHS based on it has cost at most $5n + 2$.*

Lemma 2.3.2. *If there exists an exploration scheme $\mathcal{E}_{G',s}$ on G' starting from s such that the cost of BHS based on it is at most $5n + 2$, then G has a Hamiltonian cycle that includes edge (x, y).*

The proof of lemma 2.3.1 uses the following exploration scheme for graph G'

Let $\{v_1 = y, e_1, v_2, \ldots, e_{n-1}, v_n = x, e_n, v_1 = y\}$ be a Hamiltonian cycle in G that includes (x, y).

1. **b-split**(s, s^F, y).
2. **a-split**(s, z_1, z_2, y), z_1, z_2 the twin nodes of (s, y).
3. for each v_i of the Hamiltonian cycle ($i = 1, \ldots, n - 1$):
   (a) let v_j be the third neighbor of v_i (other than v_{i-1}, v_{i+1}); if $j > i$ then **b-split**(v_i, z_1, z_2), z_1, z_2 the twin nodes of (v_i, v_j).

(b) **b-split**(v_i, v_i^F, v_{i+1}).

(c) **a-split**(v_i, z_1, z_2, v_{i+1}), z_1, z_2 the twin nodes of (v_i, v_{i+1}).

4. **a-split**(x, z_1, z_2, s), z_1, z_2 the twin nodes of (x, s).

Lemmas 2.3.1, 2.3.2 imply the following theorem.

Theorem 2.3.3. *The dBHS problem for planar graphs is NP-hard.*

2.3.3 An approximation algorithm for the BHS

The idea to construct an approximation algorithm for the BHS problem in arbitrary networks is to find a spanning tree in G and explore the graph by traversing its edges. A simple example of this approach is having both agents traverse the tree together in depth-first order. One explores a new node p while the other waits in the parent of p. This example gives an approximation ratio of 4 as observed in [CKMP05], in the sense that any exploration of an n-node graph requires at least $n-1$ steps while the example we described explores an n-node tree within $(4(n-1)-2l)$ steps, where l is the number of leaves.

To follow this approach effectively a good exploration scheme for trees and an algorithm for constructing spanning tree of G, suitable for the scheme, are needed.

We will first try to describe the exploration scheme for trees given. This scheme guarantees an approximation ratio of at most $3\frac{3}{8}$.

Let T be an n-node tree rooted at s, $n \geq 2$. The idea for the scheme is to partition the x children of an internal node p into two groups, $\lfloor x/2 \rfloor$, $\lceil x/2 \rceil$. Then agents follow the depth-first traversal of internal nodes and when *Agent-1* comes to a new node it visits all its children in the first group and respectiveley, when *Agent-2* comes to a new node it visits the children in the second group.

Now we will define an order L_T of all unexplored nodes of T. We order the children of a node according to the number of its descendants so that one with more descendants comes before and the ties are resolved arbitrarily. If $I_T = \langle w_1, \ldots, w_b \rangle$ is the sequence of the internal nodes in the depth first order, then L_T is the same sequence if we replace each w_i with its children (ordered). Then L_T contains all nodes of T except s and each node occurs only once in L_T. Let v_i be the i-th node in the order L_T and P_i be v_i's parent.

We also need to classify all nodes of T except s:

- *type-1* nodes: the leaves;

- *type-3* nodes: the internal nodes with at least one sibling;
- *type-4* nodes: the internal nodes without siblings.

We define x_t to be the number of *type-t* nodes.

The exploration scheme

Consider a tree T with no *type-4* nodes but with an odd number of nodes $n = 2q + 1 \geq 3$.

We also define $P(u, r\rangle$ to be the sequence of the nodes on the tree path from node u to node r, excluding u. If $u = r$, $P(u, r\rangle$ is the empty sequence.

Now, we define the exploration sequences $\mathbb{X}_T$, $\mathbb{Y}_T$ for *Agent-1*, *Agent-2* respectively as follows:

$$\mathbb{X}_T = \langle s\rangle \circ \phi_1^1 \circ \phi_2^1 \circ \ldots \circ \phi_q^1$$

$$\mathbb{Y}_T = \langle s\rangle \circ \phi_1^2 \circ \phi_2^2 \circ \ldots \circ \phi_q^2$$

where

$$\phi_j^1 = P(p_{2j-2}, p_{2j-1}\rangle \circ \langle v_{2j-1}, p_{2j-1}\rangle \circ P(p_{2j-1}, p_{2j}\rangle$$

$$\phi_j^2 = P(p_{2j-2}, p_{2j-1}\rangle \circ P(p_{2j-1}, p_{2j}\rangle \circ \langle v_{2j}, p_{2j}\rangle.$$

It is easy to see that the sub-sequences ϕ_j^1, ϕ_j^2 have the same length and both end at the same node (p_{2j}). Moreover, the described exploration scheme satisfies the constrains given earlier. The cost of the BHS based on it is equal to $x_1 + 3x_3$.

Because the above scheme is constructed for trees with no *type-4* nodes, we will extend it so that it also applies to trees containing *type-4* nodes. Consider a tree T that may **have type-4 nodes**. Then for each *type-4* node, add a new leaf l as its sibling and if the total number of nodes is odd, add one more leaf to an arbitrary internal node. Now the obtained tree T' has no *type-4* nodes and has an odd number of nodes, thus we obtain an exploration scheme $\mathcal{E}_T = (\mathbb{X}_T, \mathbb{Y}_T)$ from $\mathcal{E}_{T'} = (\mathbb{X}_{T'}, \mathbb{Y}_{T'})$ by replacing the traversal of the new edges with waiting.

Lemma 2.3.4. *If T is a tree rooted at s with $n \geq 2$, the exploration scheme $\mathcal{E}_T = (\mathbb{X}_T, \mathbb{Y}_T)$ for T is feasible, can be constructed in linear time and its cost is at most*

$$x_1 + 3x_3 + 4x_4 + 1.$$

Generating a good spanning tree

We will now present an algorithm which computes a spanning tree T_G of a given graph $G = (V, E)$, trying to achieve a relatively small value of $x_1 + 3x_3 + 4x_4 + 1$, thus trying to avoid creating *type-4* nodes.

We define to be the set of nodes in the current tree T (initially, $V_T = \{s\}$), $\overline{V}_T = V \setminus V_T$ the set of *external nodes*, an *external neighbor of* u is a neighbor of u in G that belongs to $\overline{V}_T$, an *expandable leaf* in T is a leaf which has at least two external neighbors and an *expandable external node*is a node in $\overline{V}_T$ which has at least one neighbor in T and at least two external neighbors or at least three external neighbors.

We are ready now to describe the algorithm which is divided in two parts.

Part 1

1. If there is an expandable leaf in T, extend T by selecting an arbitrary expandable leaf u and attaching to it all its external neighbors.

2. If there is no expandable leaf and if there is an expandable external node, then:

   - $P = (u_1, u_2, \ldots, u_k)$ a path in G of external nodes, u_1 is the only one adjacent to T, u_k is the only expandable external node on P.
   - u_0 a node in T adjacent to u_1;
   - $w_1, w_2, \ldots, w_k$ the neighbors of u_k which are neither in T nor on P;
   - u_0 must be a leaf in T;

3. extend T by attaching P to u_0 and $w_1, \ldots, w_k$ as children of u_k. Now P is called a *mid-tree path.*

Let T_1 be the tree at the end of the first part. Then there is no expandable external node left in T_1 and what remained from G is just paths. Only end nodes of these paths are adjacent to T_1, at least one for each path. Let $\mathcal{P}$ be the collection of these paths.

Part 2

- If a path P has two end nodes adjacent to T_1, split to P', P'' and replace P with those in $\mathcal{P}$.

- Now each P has only one end node adjacent to T_1.

- For each P, extend T_1 by attaching P to a neighbor of w_1 in T_1 (leaf). If P has at least two nodes, it's called a **leaf path** (without w_k).

When the second part ends (i.e. all paths from $\mathcal{P}$ are attached to T), T becomes a spanning tree T_G and it is returned by the algorithm.

This algorithm completed the attempt to find a solution for the BHS problem in an arbitrary network. The technique used for this algorithm is based on finding a suitable spanning tree of the input graph and using an algorithm for constructing exploration schemes for trees, construct an exploration scheme for the spanning tree and take it as an exploration scheme for the given graph. It can be shown that no graph exploration using this technique can guarantee a better approximation ratio than $\frac{3}{2}$. Also, the $3\frac{3}{8}$ ratio for the STE algorithm can be improved for restricted graphs, or using other methods (like selecting a maximal forest of bushy trees and then connecting them into a spanning tree), but it is believed that to obtain a much better ratio, one would need to abandon the idea of spanning trees, or at least improve it with some new ideas.

Chapter 3

The Periodic Data Retrieval Problem

The Periodic Data Retrieval problem is the problem of locating a malicious host and then visiting every other (safe) node infinitely often. We will examine this problem in an asynchronous ring with FIFO links. This problem was introduced by Miklik, Kralovic in [KM10], where they proved that the problem cannot be solved with two agents, using the fact that the network is asynchronous and therefore one cannot distinguish the case of an agent that has beem killed from the one that the agent is too slow.

3.1 Network model and problem definition

The agents operate in a ring network where each node contains one host (we will use the terms "host" and "node" interchangeably). Each host is identified by a unique label, and is connected to each of its two neighbors via labeled communication ports. Each port is associated with two order-preserving queues: one for incoming agents and a second one for outgoing agents. Additionally, each host contains a whiteboard, i.e., a piece of memory that is shared among the agents present in the node at any given time, and a queue of agents who are waiting to acquire access to the whiteboard. Neighboring hosts are connected via bidirected asynchronous FIFO links, forming an undirected graph G.

The agents are modeled as deterministic three-tape Turing machines: the first tape serves as the private memory of the agent, the second tape holds the label of the port to which the agent wishes to be transferred, and the third tape holds a copy of the whiteboard of the current node, if the agent has acquired access to the whiteboard. All agents are initially located on the same

node of the network, which we will call "the homebase." Each agent possesses a distinct identifier and knows the complete map of the network. The only way for agents to interact with each other is through the whiteboards: they are not aware of the presence of other agents on the same node or on the same link, and they cannot exchange private messages.

Each host is responsible for removing agents from the front of its incoming queues and executing them. We assume that this happens in one atomic step, i.e., as soon as one agent is removed from the front of an incoming queue, no other agent in that node can execute a transition before it executes its own first transition. The host is also responsible for executing the agent that is at the front of the whiteboard queue. Finally, the host is responsible for removing agents from the front of its outgoing queues and transmitting them over the link to the neighboring node (the whiteboard tape is not transmitted). The host has to perform these tasks while ensuring fairness: no queue can be neglected for an infinite amount of time. Each host is capable of executing multiple agents concurrently. The set of states of each agent contains special states corresponding to the following actions:

1. *Request the whiteboard lock* (q_{req}): When an agent enters this state, it is inserted in the whiteboard queue. We assume that this happens atomically, i.e., any other agent who subsequently enters this state will be placed in the whiteboard queue *behind* this agent. Its execution is suspended until it reaches the front of the queue. When this happens, the host continues to execute this agent (possibly concurrently with other agents who are not accessing the whiteboard) without removing him from the whiteboard queue. Simultaneously with the transition from q_{req}, the whiteboard of the node is copied to the third tape of the agent.

2. *Release the whiteboard lock* (q_{rel}): When an agent enters this state, its whiteboard tape is copied back to the whiteboard of the node and the agent is removed from the whiteboard queue.

3. *Leave through a specified port* (q_{port}): When an agent enters this state, it is atomically inserted in the outgoing queue of the port indicated on its second tape. If the agent has not yet released the whiteboard lock, its whiteboard tape is also copied back to the whiteboard of the node and the agent is removed from the whiteboard queue.

The system is asynchronous, meaning that any agent can be stalled for an arbitrary but finite amount of time while executing any computation or traversing any link. We assume that the system contains exactly one

malicious host which may deviate from the system specification in several ways:

1. It can choose to *kill* any agent which is stored in any of its queues or is being executed. In this case, the agent disappears without leaving any trace, apart from what it may have already written on the whiteboards.

2. It can operate without fairness, i.e., it can neglect one or more of its queues forever.

3. It can choose to transmit an agent to a node different from the one that it requested to be transmitted to, or it can transmit an agent without the agent asking for a transmission.

4. It can choose to execute (resp. forward) any agent in the incoming or the whiteboard (resp. outgoing) queues, without respecting the queue order.

5. It can create and execute multiple copies of an agent at any stage.

6. It can choose to provide to each agent that requests access to the whiteboard a completely different whiteboard tape, possibly erroneous or inconsistent with the whiteboard tapes that it has provided to the other agents.

7. It can tamper with the state of any agent which it is executing.

The agents do not have any information on the location of the malicious host, except from the fact that the homebase is safe.

In the case that the malicious host can only deviate from the system specification in the first way (i.i. kill an agent when it chooses to), we say that it is a *gray-hole*.

In our case, the malicious host x, apart from killing a visiting agent when it wishes, can also transmit the visiting agent to a different destination node than the agent requested or even without the agent requests a transmission. Host x may also serve incoming agents in different order than the order they arrived. Finally x may copy an incoming agent A and forward A more than once. We call such a malicious host a *red-hole*. The location of the malicious host is unknown to the agents. Unless explicitly mentioned, the red-hole cannot change the contents of its whiteboard or the state (i.e., memory) or the identity of a visiting agent.

The agents operate in the network and their aim is to deliver the data from any safe node to the homebase infinitely many times. During the delivery,

the data may be stored at an intermediate node and possibly read by another agent before reaching the homebase. This problem is known as the *Periodic Data Retrieval* problem.

Definition 3.1.1 ([KM10])**.** *Consider an arbitrary execution of the system. A node w is said to be* reported from time t *if there exist a natural number r, a sequence of (not necessarily distinct) agents $A_0, \ldots, A_r$, a sequence of nodes $v_0, \ldots, v_r$, and an increasing sequence of times $t_0 < \cdots < t_r$, such that v_0 is w, v_r is the homebase, $t \leq t_0$, and, for each i, agent A_i visits node v_i at time t_i and node v_{i+1} at time t'_i, where $t_i < t'_i < t_{i+1}$.*

An algorithm solves the Periodic Data Retrieval problem if in any execution, for any node v and any time t, v is reported from time t.

3.2 Lower Bounds

Before we present the algorithms for the Periodic Data Retrieval problem we will give a new lower bound for the problem proving that no algorithm can solve this problem using three agents.

Theorem 3.2.1. *No algorithm can solve the Periodic Data Retrieval problem in rings containing a red-hole with reliable whiteboard using less than four agents.*

Proof. Clearly any algorithm using only one agent cannot solve the problem. Consider an algorithm using more than one agents. Let u, v, w, z be adjacent nodes in clockwise order. Suppose agent A is the first agent blocked while going from node v to node w. One of the remaining agents has to visit at least one of the nodes v, w, since otherwise agent A may have been vanished while leaving v or entering w and not all safe nodes are visited. If a second agent B visits v or w then B may also vanish and therefore two agents are not enough. Hence a second agent, say B has to visit v or w and for the same reason, a third agent C has to visit v or w. If agents B and C visit the same node then all of them may vanish. Hence suppose that B wants to visit node v and C wants to visit node w.

If B tries to visit v traveling counterclockwise then both A, B may have been vanished either while A exits v and B enters v or while both enter w (from different directions). Hence the adversary may select the location of the red hole to be either v or w and therefore agent C will vanish as well when it visits any of those two nodes.

Suppose that B visits v traveling clockwise and C visits w traveling counterclockwise. Suppose also that agent B is unblocked. If B next visits w then

all agents may vanish. For the same reason if agent C is unblocked and C visits v then again all agents may vanish. Hence when B is unblocked it must visit u and when C is unblocked it must visit z. Now suppose that agent A is unblocked and sees that B has left v going to u and C has left w going to z.

- If agent A decides to visit z then both A, C may vanish on the link (w, z) and since B has to visit either w or z all agents may vanish.
- Similarly as before, if agent A decides to visit u then both A, B may vanish on the link (v, u) and since C has to visit either v or u all agents may vanish.
- If agent A does not visit u, z then it can only move on the link (v, w). Since all safe nodes have to be visited, agents B and C have to travel in opposite directions: B counterclockwise and C clockwise. If they never visit the same node then one of them may vanish and therefore, after that, there will be nodes never visited by the other agent. If they visit the same node then both may vanish and (since A moves between u and v forever), not all safe nodes will be visited periodically.

♢

Therefore an algorithm which solves the Periodic Data Retrieval problem requires at least four agents.

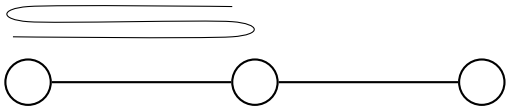
3.3 The Algorithms

The first algorithm we will describe is the one presented in [KM10].

First of all, when an agent enters a node and since it has the whiteboard lock and before it performs any other actions, it writes on the whiteboard its ID, the number of the steps it has walked, the node it came from and the node it wants to visit next. Also each agent has an internal variable called `transfer_ID` for the ID of the node it wants to be transfered to and another one called `steps` which counts the numbers of its steps.

We will call an agent *clockwise* if it travels in the clockwise direction on the ring and respectively *counterclockwise* if it travels in the counterclockwise direction.

The agents use the *cautious walk* to visit a node, i. e. one step forward, on step back, one step forward. When they start the first step forward of their cautious walk they leave a *flag* at the port they left from and they remove the flag when they start their second step forward.



If an agent makes the first step of its cautious walk and finds that there is a flag at the port it would like to use, then it *bounces*, i.e.makes the back step but does not complete the cautious walk; it starts walking using cautious walk in the opposite direction.

Now, to prevent the malicious node from forwarding an agent without running it or sending an agent to a node without the agent's request, the agent when entering a node compares its `transfer_ID` to the node's ID and if it is not equal, it dies. Therefore, forwarding an agent without running it or sending an agent to a node without the agent's request is equivalent to killing the agent.

Forwarding an agent more than once can also be prevented using the `steps` variable. Since an agent's steps are also stored in nodes, the agent can detect if it was forwarded over a link more than once and, if this is the case, die.

When an agent is about to leave a node while still holding the whiteboard lock it first gets a copy of the list with the agents that left the node before from the same port. When it arrives to the next node it compares that list to the one which contains the agents who came to that new node from that port. If it finds a difference it dies. Therefore, if the malicious node try to break the FIFO property of the nodes the agent will know and die so breaking the FIFO property is equivalent to killing the agent.

If the malicious host decides to kill an agent in the middle of its computation, then the lock of the whiteboard is never releases thus every other agents that reaches the malicious host will wait forever effectively being killed too.

So without loss of generality we can assume that the malicious node cannot do any of the above.

Královic and Miklík in [KM10] proved that with this algorithm at most 8 agents die, therefore 9 agents can solve the Periodic Data Retrieval problem.

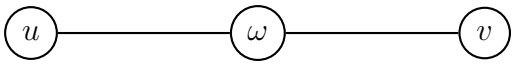
3.3.1 A new 7-agent algorithm

Agents move using the cautious walk. Suppose that an agent A moves clockwise and suppose it moves from node u to node v. Since A moves using the cautious walk, it takes on step in the clockwise direction and then one step in the counterclockwise direction, from v to u. During this back step, the agent won't take a copy of the list with the agents that left v to visit u. Thus it ignores all counterclockwise agents.

We will use the notation u^+ (resp. u^- for a flag that a clockwise (resp. counterclockwise) agent left at node u. Thus, a u^+ flag can be found only at the clockwise port of u and a u^- flag can be found only at the counterclockwise port of u.

Therefore, the agents will follow the algorithm described above but with this modification which makes the algorithm even simpler but also allows us to prove that 7 agents can solve this problem.

The following lemmas lead us to this conclusion. For all lemmas we assume that ω is the black hole and its two adjacent nodes are u and v.



Lemma 3.3.1. *If a flag is left at u^+ at time t then no clockwise agent is killed after t.*

Proof. If a flag is left at u^+ then every other clockwise agent that reaches u will bounce and continue in the opposite direction. Therefore, no clockwise agent can be killed. ♢

Lemma 3.3.2. *If a flag is left at ω^+ at time t then at most one clockwise agent is killed after t.*

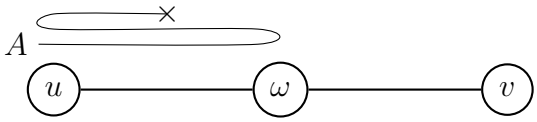
Proof. Let B be the next clockwise agent that is killed. Then ω can kill B only right before entering or right before leaving ω, otherwise B would bounce and continue in the opposite direction because of the flag at ω^+. Thus B leaves a flag at u^+ and by Lemma 3.3.1 no other clockwise agent will be killed. ♢

Lemma 3.3.3. *If an agent is killed at ω at time t_1 and leaves no flag at u^+, ω^+, then the next clockwise agent that is killed at ω at time $t_2 > t_1$ leaves a flag at u^+ or ω^+ or the third clockwise agent that is killed at ω leaves a flag at u^+.*

Proof. Let A be the first clockwise agent that is killed at ω without leaving a flag at u^+, ω^+, B the second clockwise agent that is killed at ω and C the third one.

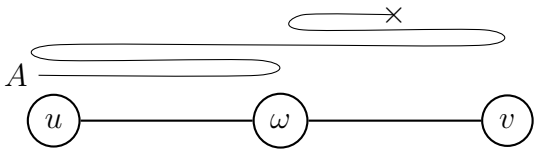
The only possible ways of A dying without leaving a flag are while traversing the link $u \to \omega$ or $\omega \to v$ for the second time.

(a) A dies while traversing the link $u \to \omega$ for the second time.



Then B dies traversing the same link for the first time because of A's death and leaves a flag at u^+.

(b) A dies while traversing the link $\omega \to v$ for the second time.



Then one of the following will happen:

(i) ω kills B right before or right after visiting ω for the first time thus B leaves a flag at u^+.

(ii) ω kills B right before entering ω for the second time. Then C will die while traversing the link $u \to \omega$ for the first time because of B's death and leaves a flag at u^+.

(iii) B is killed while traversing the link $\omega \to v$ for the first time (by ω or because of A's death) leaving a flag at ω^+.

♢

By lemmas 3.3.1, 3.3.2 and 3.3.3 we have:

Theorem 3.3.4. *At most three agents coming from the same direction die at ω.*

The above theorem immediately implies that at most six agents will die in ω.

Theorem 3.3.5. *At most six agents die at ω.*

3.3.2 An optimal algorithm

Before presenting an optimal algorithm for the problem let us give some preliminary definitions.

- The whiteboard at any node v contains a queue $WBoardQueue(v)$ and an another queue $WBoardList(v)$. Any agent A at v requests access to the whiteboard when $WBoardQueue(v)$ is empty. When the access is granted (assuming a mutual exclusion mechanism), agent A inserts its ID in $WBoardQueue(v)$ and until $WBoardQueue(v)$ is again empty, no other agent coming to v may request access. Agent A completes its computations at v and departs from v removing its entry from $WBoardQueue(v)$.

- When an agent A is ready to depart from a node v with destination an adjacent node w in selected direction dir, it first gets a copy of a queue $WBoardList(v)$ which is stored in the whiteboard (initially $WBoardList(v)$ is empty). Then, the agent inserts the message $< AgentID, depart, dir, Stp >$ in $WBoardList(v)$, where $AgentID$ is A's ID, $depart$ is a constant, dir is the selected direction (ClockWise or CounterClockWise), and Stp is the total number of traversals done by A. Then A may (depending on its computations) put a flag on node v which will be v^- (if the selected direction is CounterClockWise) or v^+ (if the selected direction is ClockWise). Then A removes its entry from $WBoardQueue(v)$ and moves in direction dir to an adjacent node w.

- Immediately after an agent A has come to a node w from a node v traveling in direction dir and after has been granted access to its whiteboard, it gets a copy of a queue $WBoardList(w)$ which is stored in the whiteboard (initially $WBoardList(w)$ is empty). Then A checks for inconsistencies between $WBoardQueue(v)$ and $WBoardQueue(w)$. If A finds an inconsistency then it places a flag at the incoming port of w, removes its entry from $WBoardQueue(w)$ and deactivates itself. A also checks whether it itself is a copy created by the red-hole. If this is the case it can imply that the red-hole location is at node v and in that case can periodically visit all safe nodes without visiting v. Finally, A checks whether w is the node which had been selected as a destination by A before moving from v. If w was not the selected destination then if the distance between v and w is 1 it implies that the red-hole location is at node v, otherwise the red-hole location is at the selected destination. And in any of those cases it can periodically visit all safe nodes without visiting the red-hole location. If no errors are found, A inserts the message $< AgentID, arrive, dir, Stp + 1 >$ in $WBoardList(v)$, where $AgentID$ is A's ID, $arrive$ is a constant, dir is the direction (ClockWise or CounterClockWise) at which A was moving before entering w, and $Stp+1$ is the total number of traversals

done by A. Then A may (depending on its computations) remove a flag from node w which had been placed by A earlier and is w^- (if *dir* is ClockWise) or w^+ (if *dir* is CounterClockWise). Then A computes a new direction to move, removes its entry from $WBoardQueue(w)$ and is ready to depart.

Now we are ready to present the algorithm. Let u, v be two adjacent nodes, where v is clockwise of u. Consider an agent A at u. A high level description of the algorithm is the following.

- If u is the HomeBase then A waits until $WBoardQueue(u)$ is empty and at least one port of u does not have a flag. Agent A inserts its ID in $WBoardQueue(u)$ and then chooses the direction which does not have a flag (or ClockWise if both ports of u are free).

- If u is not the HomeBase then A waits until $WBoardQueue(u)$ is empty and then inserts its ID in $WBoardQueue(u)$. If A can continue moving in the same direction as before (i.e., the port of u in that direction is free) then A selects this direction, otherwise A selects the opposite direction (i.e., going back).

Suppose that the selected direction is ClockWise (i.e., there is no flag u^+). A leaves a message at u and a flag u^+, removes its ID from $WBoardQueue(u)$ and moves to node v. After A reaches v it waits until $WBoardQueue(v)$ is empty and inserts its ID in $WBoardQueue(v)$. (We sometimes refer to this move from u to v as 'the first part of the phases'.) Then:

- If there is a flag v^+ or v^-, agent A leaves a message at v, removes its ID from $WBoardQueue(v)$ and returns to u. After A reaches u it waits until $WBoardQueue(u)$ is empty, inserts its ID in $WBoardQueue(u)$ and then A removes flag u^+, and leaves a message at u. If A can continue moving in the same direction (i.e., there is no flag u^-) then A selects this direction, otherwise A selects the opposite direction (i.e., towards v). Then A removes its ID from $WBoardQueue(u)$. We call this cycle (together with the first part of the phases) from u to v and back to u **Phase-1**. Phase-1 starts when agent A removes its ID from $WBoardQueue(u)$ and finishes when A (after coming back to u) removes its ID from $WBoardQueue(u)$.

- If there is a no flag at v, agent A leaves a message at v and a flag v^-, removes its ID from $WBoardQueue(v)$ and returns to u. After A reaches u it waits until $WBoardQueue(u)$ is empty, inserts its ID in $WBoardQueue(u)$ and then A removes flag u^+, leaves a message

at u, removes its ID from $WBoardQueue(u)$ and again moves to v. After A reaches v it waits until $WBoardQueue(v)$ is empty, inserts its ID in $WBoardQueue(v)$ and then A removes flag v^- and leaves a message at v. If A can continue moving in the same direction (i.e., there is no flag v^+) then A selects this direction, otherwise A selects the opposite direction (i.e., towards u). Then A removes its ID from $WBoardQueue(v)$. We call this zig-zag (together with the first part of the phases) from u to v, back to u and then again to v **Phase-2**. Phase-2 starts when agent A removes its ID from $WBoardQueue(u)$ and finishes when A (after coming for the second time to v) removes its ID from $WBoardQueue(v)$.

The agent is instructed to execute an infinite sequence of phases.

In the analysis of the algorithm we assume that no agent has found the exact location of the red-hole, since in such a case, this agent (which has a map of the ring) can periodically visit all safe nodes of the ring.

Let u, v be adjacent nodes. Suppose that an agent A starts a phase at u, after having selected to move towards v. We say that 'agent A *enters* link (u, v)'. We also say that 'agent A *exits* link (u, v)' when the phase has finished.

Before proving the correctness of the algorithm we give a few preliminary lemmas.

Function Symbol(*dir*)

```
1: if dir = ClockWise then
2:     return '+'
3: else
4:     return '-'
```

Lemma 3.3.6. *Let u, x, v be adjacent nodes in clockwise order and x be the red-hole node. Consider the following configuration: there are, i) flag u^+ or flag v^- or an agent blocked at u (i.e., having the access of the whiteboard at u) or an agent blocked at v (i.e., having the access of the whiteboard at v) and ii) an agent blocked (or vanished) at x (i.e., having the access of the whiteboard at x) or a flag at x. Then at most one more agent (executing Algorithm* `PeriodicTraverse`*) can enter either link (u, x) (starting its phase at u) or link (v, x) (starting its phase at v).*

Proof. Suppose wlog there is either a flag u^+ or an agent blocked at u (i.e., having the access of the whiteboard at u). Since there is also a flag at x or an

Procedure **Check**($DepList, ArrList, src, v, w, dir, Stp$)

1: **if** $v \neq w$ **then**
2: **if** $distance(src, w) = 1$ **then**
3: $HoleFound(src)$
4: **else**
5: $HoleFound(v)$
6: **if** $\exists$ N: $< AgentID, arrive, dir, N > \in ArrList$, where $Stp < N$ **then**
7: $HoleFound(src)$
8: **for all** M: $< M, depart, dir, N > \in DepList$ **do**
9: **if** $< M, arrive, dir, N+1 > \notin ArrList$ **then**
10: Put a flag $Symbol(Opposite(dir))$ at w
11: Remove $< AgentID >$ from $WBoardQueue(w)$
12: Deactivate
13: **for all** M: $< M, arrive, dir, N+1 > \in ArrList$ **do**
14: **if** $< M, depart, dir, N > \notin DepList$ **then**
15: Put a flag $Symbol(Opposite(dir))$ at w
16: Remove $< AgentID >$ from $WBoardQueue(w)$
17: Deactivate

agent blocked at x (i.e., having the access of the whiteboard at x), another agent C can only enter link (x, v) starting starting its phase at node v and thus leaving a flag v^- at v. Moreover, this agent C will be either blocked at x or executing a Phase-1. At the end of its phase, C may be blocked for some time at v. In any case no other agent can enter links (u, x) or (v, x) until C exits link (x, v) from v. ♢

Lemma 3.3.7. *Let u, x, v be adjacent nodes in clockwise order and x be the red-hole node. Consider two agents (following Algorithm* `PeriodicTraverse`*) that have entered at links (u, x), (v, x). Then the following configuration appears and remains until one agent exits both links: there are flags u^+ and v^- or both (i) and (ii) below hold:*

- *i) a flag u^+ or a flag v^- or an agent blocked at u (i.e., having the access of the whiteboard at u) or an agent blocked at v (i.e., having the access of the whiteboard at v) and*
- *ii) an agent blocked (or vanished) at x (i.e., having the access of the whiteboard at x) or a flag at x.*

Proof. Let A, B be the two agents that have entered at links (u, x), (v, x).

Function **Phase**($CurrNode, PhaseState, dir, Stp$)

1: Let v be the node which is adjacent to $CurrNode$ in direction dir
2: $DepList \leftarrow WBoardList(CurrNode)$
3: Insert $< AgentID, depart, dir, Stp >$ at $WBoardList(CurrNode)$
4: **if** (($PhaseState = 0$) OR ($PhaseState = 3$)) **then**
5: Put a flag $Symbol(dir)$ at $CurrNode$
6: Remove $< AgentID >$ from $WBoardQueue(CurrNode)$
7: Move in direction dir to v
8: Let w be the reached node
9: Wait until $WBoardQueue(w)$ is empty
10: Insert $< AgentID >$ at $WBoardQueue(w)$
11: $ArrList \leftarrow WBoardList(w)$
12: $Check(DepList, ArrList, CurrNode, v, w, dir, Stp)$
13: Insert $< AgentID, arrive, dir, Stp + 1 >$ at $WBoardList(v)$
14: **if** $PhaseState = 0$ **then**
15: **if** there are no flags at v **then**
16: $PhaseState \leftarrow 3$
17: **else**
18: $PhaseState \leftarrow 1$
19: **else**
20: $PhaseState \leftarrow PhaseState + 1$
21: **return** $PhaseState$

Procedure HoleFound($HoleNode$)

1: Visit periodically all nodes apart from $HoleNode$ ignoring any flags

- Case 1: Both agents started their entering phases at the same adjacent node of x. Suppose without loss of generality that this node is u. Suppose also wlog that agent A departed first from node u (hence placing a flag u^+). Since B cannot leave node u going towards x before A returns to u and removes the flag, A must have been executing a Phase-2. Then A is either vanished or blocked before finishing its phase or exits link (u, x) from x. Agent B may enter link (u, x) and until either B exits link (u, x) from u or A exits link (u, x) from x there is a flag u^+ (or B is blocked at u) and either A is blocked at x or there is a flag at x.

- Case 2: Agent A started at u and B at x entering link (x, u).

  a) If A is vanished before getting access to x's whiteboard and B is van-

Function Initialize(s)

1: Wait until (($WBoardQueue(s)$ is empty) AND
(at least one port of s without a flag))
2: Insert $< AgentID >$ at $WBoardQueue(s)$
3: **if** there are no flags at s **then**
4: $dir \leftarrow ClockWise$
5: **else**
6: Let dir be the direction of s without a flag
7: **return** dir

ished before getting access to u's whiteboard then there are permanent flags u^+ and x^-.

b) If A gets access to x's whiteboard before B appears at x to start its entering phase then:

- If there is no flag x^+ at x (flag x^- is impossible since A, B are the only ones entered links (u, x) and (v, x)) then A will continue performing a Phase-2, placing a flag x^-. Hence in that case B will not be able to enter link (x, u) before A finishes its phase.
- If there is a flag x^+ at x then A will continue performing a Phase-1. If A is vanished while coming back to u then after B enters link (x, u) (placing a flag x^- at x) will block itself at u. Otherwise, until A finishes its phase and exits link (u, x) or B exits link (u, x), there will be (an agent blocked at u or a flag u^+) and (an agent blocked at x or a flag x^- at x).

c) If A gets access to x's whiteboard after B appears at x to start its entering phase then if B is blocked at x, A will be also blocked while there is a flag u^+. The remaining case is that A gets access to x's whiteboard after B has entered link (x, u) having placed a flag x^-. Hence A continues performing a Phase-1. If B is vanished before getting access to u's whiteboard then flags u^+ and x^- permanently remain. Otherwise B will get access to u's whiteboard before A finishes its phase. Hence B will continue performing a Phase-1 and until A exits link (u, x) or B finishes its phase (there is a flag u^+ or an agent blocked at u) and (there is a flag x^- or an agent blocked at x).

d) If B gets access to u's whiteboard before A appears at u to start its entering phase then if B performs a Phase-2 then A will not be able to enter link (u, x) until B finishes its phase. Hence B should perform a Phase-1. If A or B vanish before B finish its phase then there will be a

Algorithm PeriodicTraverse

```
1: PhaseState ← 0
2: Stp ← 0
3: Let s be the HomeBase node
4: dir ← Initialize(s)
5: NewPhaseState ← Phase(s, PhaseState, dir, Stp)
6: Stp ← Stp + 1
7: loop
8:    Let s be the current node
9:    PhaseState ← NewPhaseState
10:   switch (PhaseState)
11:   case 1, 3:  dir ← Opposite(dir))
12:   case 2:
13:      Remove flag Symbol(Opposite(dir)) from s
14:      PhaseState ← 0
15:      if there is a flag Symbol(dir) at s then
16:         dir ← Opposite(dir))
17:   case 4:
18:      Remove flag Symbol(Opposite(dir)) from s
19:      dir ← Opposite(dir))
20:   case 5:
21:      Remove flag Symbol(Opposite(dir)) from s
22:      PhaseState ← 0
23:      if there is a flag Symbol(dir) at s then
24:         dir ← Opposite(dir))
25:   end switch
26:   NewPhaseState ← Phase(s, PhaseState, dir, Stp)
27:   Stp ← Stp + 1
```

permanent flag u^+ and either a permanent flag x^- or an agent blocked forever at x. If none of them vanish until B finishes its phase there will be a flag u^+ and either an agent blocked at x or a flag at x.

- Case 3: Agent A started at u and B at x entering link (x, v). If B executes a Phase-1 then until B finishes its phase or A finishes its phase (which has to be also a Phase-1) and exits link (u, x) from u, there is a flag u^+ and either a flag x^+ or an agent blocked at x. If B executes a Phase-2 then it places a flag v^- and upon B's return to x, flag u^+ remains. If B removes x^+ from x while performing its Phase-2 before A reaches x then A will place a flag x^- at x (performing a

Phase-2) before removing flag u^+ from u. Flag v^- and either a flag x^- or an agent blocked at x remain until either A finishes its phase or B exits link (x, v).

- Case 4: One agent started its entering phase at u and the other one at v. Suppose wlog that agent A started at u and B at v. If both of them vanish before they get access to x's whiteboard then there are permanents flags u^+ and v^-. Suppose wlog that A is the first one that gets access to x's whiteboard. As long as A is blocked at x the flags u^+ and v^- remain. If A is unblocked then he has to continue its phase which will be a Phase-2 (B has not yet been able to place a flag at x). Hence A places a flag x^- at x. There is (a flag x^- or agent A is blocked at x or a flag x^+) and v^- which remain until either A finishes its phase (and until then B cannot enter link (x, u)) or B exits link (v, x) from v.

- Case 5: Both agents started their entering phases at x. Let A be the first agent starting a phase at x entering wlog link (x, v). Then before A finishes its phase, B should have entered either link (u, x) from u or link (v, x) from v in both cases performing a Phase-2 (otherwise B will not finish to x for starting its entering phase from x).

  - Suppose that before A finishes its phase, B has entered link (u, x) from u performing a Phase-2. If A executes a Phase-1 then B cannot enter link (x, v) before A finishes its phase. Also B cannot enter link (x, u) starting at x after A has started its phase and before finishing because this would require that B approaches first node x which does not have flags (hence placing a flag x^-) and in that case A could not perform a Phase-2 from v or u ending first at x to start its entering phase. If A executes a Phase-2 then the only way that B can start its phase at x before A finishes is when B starts its phase at x after A has passed from x for the second time (otherwise B could not end up at x performing a Phase-2 and if B approaches x before A appears then A cannot end up at x first by a Phase-2 from u or v). Then until A finishes its phase there is a flag x^+ and either a flag v^- or an agent blocked at v.
  - Suppose that before A finishes its phase, B has entered link (v, x) from v performing a Phase-2. If B approaches x before A appears at x then (since B places a flag x^+), A cannot enter link (x, v) before B finishes its phase. Hence A will not be the first starting a phase at x entering link (x, v). If B approaches x after A has

entered link (x, v), then A should perform a Phase-2 (otherwise B will not be able to perform a Phase-2 before A finishes). But it is impossible for both of them to perform a Phase-2 before A finishes its phase.

♢

Lemma 3.3.8. *Let u, x, v be adjacent nodes in clockwise order and x be the red-hole node. Suppose that three agents have entered the links (u, x), (v, x) and they have not exited. Then the remaining agent following Algorithm* `PeriodicTraverse` *periodically visits all safe nodes.*

Proof. In view of Lemmas 3.3.7 and 3.3.6, after two agents have entered links (u, x), (v, x) and have not exited at most one more agent can also enter the links. Hence a forth agent will never enter those links. An agent following Algorithm `PeriodicTraverse` never permanently leaves a flag unless is vanished at the red-hole or it is blocked at the red-hole where another agent has vanished. Hence all nodes except x are periodically visited. ♢

In view of Lemma 3.3.8 and Theorem 3.2.1 we get:

Theorem 3.3.9. *Algorithm* `PeriodicTraverse` *solves the Periodic Data Retrieval problem in rings containing a red-hole using an optimal number of four agents.*

Conclusion

In this thesis we first studied the Black Hole Search problem under various settings, presenting also some interesting algorithms and impossibility results.

The main part of the thesis dealt with the Periodic Data Retrieval Problem which we studied in the case of reliable whiteboards and we provided a better upper bound for the number of agents needed and a matching lower bound, thus giving an oprimal algorithm. As a byproduct we gave a simple 7-agent algorithm for the problem. Our optimal algorithm solves the Periodic Data Retrieval Problem with 4 agents.

Regarding future research, it would be ineresting to study the following:

- the case of unreliable whiteboards, i.e. the malicious host may change the content of the whiteboard; we wish to improve the number of agents needed (the upper bound for this case is 27 agents as proved in [Mik10]) and provide new lower bounds for the problem,
- consider much stronger malicious host who may alter the agent's state, delete or change their memory, change their IDs, change their program or even produce malicious agents,
- consider other topologies such as trees, grids, general graphs etc.

Bibliography

[AH00] Susanne Albers and Monika R. Henzinger, *Exploring unknown environments*, SIAM J. Comput. **29** (2000), no. 4, 1164–1188.

[BFMS10] Balasingham Balamohan, Paola Flocchini, Ali Miri, and Nicola Santoro, *Time optimal algorithms for black hole search in rings*, Proceedings of the 4th international conference on Combinatorial optimization and applications - Volume Part II (Berlin, Heidelberg), COCOA'10, Springer-Verlag, 2010, pp. 58–71.

[BFR+02] Michael A. Bender, Antonio Fernndez, Dana Ron, Amit Sahai, and Salil Vadhan, *The power of a pebble: Exploring and mapping directed graphs*, Information and Computation **176** (2002), no. 1, 1 – 21.

[CD12] Jiannong Cao and Sajal K. Das, *Mobile agents and applications in networking and distributed computing*, pp. 1–16, John Wiley & Sons, Inc., 2012.

[CDK+10] Jurek Czyzowicz, Stefan Dobrev, Rastislav Kralovic, Stanislav Miklk, and Dana Pardubska, *Black hole search in directed graphs*, Structural Information and Communication Complexity (Shay Kutten and Janez erovnik, eds.), Lecture Notes in Computer Science, vol. 5869, Springer Berlin Heidelberg, 2010, pp. 182–194.

[CDLM11] Jeremie Chalopin, Shantanu Das, Arnaud Labourel, and Euripides Markou, *Black hole search with finite automata scattered in a synchronous torus*, Distributed Computing (David Peleg, ed.), Lecture Notes in Computer Science, vol. 6950, Springer Berlin Heidelberg, 2011, pp. 432–446.

[CDS07] Jeremie Chalopin, Shantanu Das, and Nicola Santoro, *Rendezvous of mobile agents in unknown graphs with faulty links*,

Distributed Computing (Andrzej Pelc, ed.), Lecture Notes in Computer Science, vol. 4731, Springer Berlin Heidelberg, 2007, pp. 108–122.

[CKMP05] Jurek Czyzowicz, Dariusz Kowalski, Euripides Markou, and Andrzej Pelc, *Searching for a black hole in tree networks*, Principles of Distributed Systems (Teruo Higashino, ed.), Lecture Notes in Computer Science, vol. 3544, Springer Berlin Heidelberg, 2005, pp. 67–80.

[DFPS01] Stefan Dobrev, Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro, *Mobile search for a black hole in an anonymous ring*, Distributed Computing (Jennifer Welch, ed.), Lecture Notes in Computer Science, vol. 2180, Springer Berlin Heidelberg, 2001, pp. 166–179 (English).

[DFPS06] Stefan Dobrev, Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro, *Searching for a black hole in arbitrary networks: optimal mobile agents protocols*, Distributed Computing **19** (2006), no. 1, 1–99999 (English).

[DP99] Xiaotie Deng and Christos H. Papadimitriou, *Exploring an unknown graph*, J. Graph Theory **32** (1999), no. 3, 265–297.

[FIP+04] Pierre Fraigniaud, David Ilcinkas, Guy Peer, Andrzej Pelc, and David Peleg, *Graph exploration by a finite automaton*, MFCS, 2004, pp. 451–462.

[FIS12] Paola Flocchini, David Ilcinkas, and Nicola Santoro, *Ping pong in dangerous graphs: Optimal black hole search with pebbles*, Algorithmica **62** (2012), no. 3-4, 1006–1033 (English).

[FKMS09a] Paola Flocchini, Matthew Kellett, Peter Mason, and Nicola Santoro, *Map construction and exploration by mobile agents scattered in a dangerous network*, Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing (Washington, DC, USA), IPDPS '09, IEEE Computer Society, 2009, pp. 1–10.

[FKMS09b] Paola Flocchini, Matthew Kellett, Peter C. Mason, and Nicola Santoro, *Map construction and exploration by mobile agents scattered in a dangerous network*, IPDPS, 2009, pp. 1–10.

[FLP11] Peter Floderu, Andrzej Lingas, and Mia Persson, *Towards more efficient infection and fire fighting*, Proceedings of the Seventeenth Computing: The Australasian Theory Symposium - Volume 119 (Darlinghurst, Australia, Australia), CATS '11, Australian Computer Society, Inc., 2011, pp. 69–74.

[FMS98] Paola Flocchini, Bernard Mans, and Nicola Santoro, *Sense of direction: Definitions, properties, and classes*, Networks **32** (1998), no. 3, 165–180.

[HKKK08] Nicolas Hanusse, Dimitris J. Kavvadias, Evangelos Kranakis, and Danny Krizanc, *Memoryless search algorithms in a network with faulty advice*, Theor. Comput. Sci. **402** (2008), no. 2-3, 190–198.

[KKM06] Evangelos Kranakis, Danny Krizanc, and Euripides Markou, *Mobile agent rendezvous in a synchronous torus*, LATIN, 2006, pp. 653–664.

[KKM10a] Evangelos Kranakis, Danny Krizanc, and Euripides Markou, *The mobile agent rendezvous problem in the ring*, Synthesis Lectures on Distributed Computing Theory **1** (2010), no. 1, 1–122.

[KKM10b] Evangelos Kranakis, Danny Krizanc, and Euripides Markou, *The mobile agent rendezvous problem in the ring*, Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool Publishers, 2010.

[KKM11] Evangelos Kranakis, Danny Krizanc, and Euripides Markou, *Deterministic symmetric rendezvous with tokens in a synchronous torus*, Discrete Applied Mathematics **159** (2011), no. 9, 896–923.

[KKS08] Evangelos Kranakis, Danny Krizanc, and Sunil M. Shende, *Tracking mobile users in cellular networks using timing information*, Nord. J. Comput. **14** (2008), no. 3, 202–215.

[KM10] Rastislav Kralovic and Stanislav Miklik, *Periodic data retrieval problem in rings containing a malicious host*, Structural Information and Communication Complexity (Boaz Patt-Shamir and Tinaz Ekim, eds.), Lecture Notes in Computer Science, vol. 6058, Springer Berlin Heidelberg, 2010, pp. 157–167.

[KMP08] Ralf Klasing, Euripides Markou, and Andrzej Pelc, *Gathering asynchronous oblivious mobile robots in a ring*, Theor. Comput. Sci. **390** (2008), no. 1, 27–39.

[KMRS07] Ralf Klasing, Euripides Markou, Tomasz Radzik, and Fabiano Sarracco, *Hardness and approximation results for black hole search in arbitrary networks*, Theoretical Computer Science **384** (2007), no. 2-3, 201 – 221, Structural Information and Communication Complexity (SIROCCO 2005).

[M+12] Euripides Markou et al., *Identifying hostile nodes in networks using mobile agents*, Bulletin of the EATCS (2012), no. 108, 93–129.

[Mik10] Stanislav Miklik, *Exploration in faulty networks*, Ph.D. thesis, Department of Computer Science, Faculty of Mathematics, Physics and Informatics, Comenius University, 2010.

[PP99] Petrisor Panaite and Andrzej Pelc, *Exploring unknown undirected graphs*, Journal of Algorithms **33** (1999), no. 2, 281 – 295.