Fault-Tolerant Exploration of an Unknown Dangerous Graph by Scattered Agents

Paola Flocchini[1], Matthew Kellett[2], Peter C. Mason[2], and Nicola Santoro[3]

[1] School of Electrical Engineering and Computer Science, University of Ottawa
[2] Defence R&D Canada – Ottawa, Government of Canada
[3] School of Computer Science, Carleton University

Abstract. Black hole search (BHS) is the problem of mapping or exploring a network where there are dangerous sites (black holes) that eliminate any incoming searcher without leaving a discernible trace. Dangerous graph exploration (DGE) extends the BHS problem to include dangerous links (black links). In the literature, both problems have only been studied under the assumption that no faults occur in the network during the exploration. In this paper, we examine the impact that link failures can have on the exploration of dangerous graphs. We study the DGE problem under the following conditions: there are multiple black holes and black links, the network topology is unknown, the searchers are initially scattered in arbitrary locations, and the system is totally asynchronous. In this difficult setting, we assume that links can fail during the computation. We present an algorithm that solves the DGE in the presence of such dynamic link failures. Our solution to the problem works with an optimum number of searchers in a polynomial number of moves. This is the first result dealing with fault-tolerant computations in dangerous graphs.

1 Introduction

Network mapping is an important problem that goes all the way back to Claude Shannon's building of a physical maze solving machine [20]. Mapping and its associated problem of exploration, the visiting of every node in a network or the crossing of every edge, has been a significant focus of research in the mobile agent model of distributed computing. In the last decade, a significant portion of that work has focussed on exploration and mapping when the network is not safe for the agents.

A particular kind of danger is the presence in the network of *black holes*, network sites that eliminate agents arriving at them without leaving a discernible trace. The *black hole search* (BHS) problem is the problem of locating such harmful sites. In order to solve the problem, a team of agents must work together to find the black holes because some agents must be eliminated in order for it be detected. The *dangerous graph exploration* (DGE) problem extends the BHS problem to include *black links*, network links that act in the same way as black holes, destroying any agent traversing them without leaving a discernible

A.W. Richa and C. Scheideler (Eds.): SSS 2012, LNCS 7596, pp. 299–313, 2012.

trace. In both the BHS and DGE problems, there are three basic requirements for solvability and for termination: the *safe* portion of the network (i.e., the remaining network once the black links, as well as the black holes and their incident links are removed) has to be connected; the number k of agents must be greater that the size f of the *frontier* (i.e., the links from the safe nodes to the unsafe portion of the network); information about the number n_s of safe nodes or the size of the frontier must be known. We assume that such requirements are met.

The BHS problem was introduced by Dobrev et al. in [10], which focuses on finding a single black hole in an asynchronous ring network, and there has been extensive research since then [1–3, 5–9, 11, 13–19, 21]. The DGE problem was first investigated by Chalopin, Das, and Santoro in [4], where agents scattered in an anonymous network of unknown topology solved the DGE problem as a consequence of solving the mobile agent rendezvous problem. The same problem when the unknown network is not anonymous is examined in [12]. The solution presented there works in $O(nm)$ agent moves, where n is the number of nodes and m is the number of links, a cost which is proven to be optimal in [19].

In all existing investigations of the BHS and DGE problems, it is assumed that the environment is fault-free—that is, the computation is dangerous by nature, due to the presence of black holes and black links. What happens if the network topology is not static or if links could go down while the computation takes place? None of the existing solutions addresses these questions and none would tolerate even a single link failure.

The goal of this paper is to start examining the problem of *fault-tolerant* exploration of dangerous graphs. In particular, we focus on solving the DGE problem in presence of *dynamic link failures*. Dealing with link failures is a first step towards algorithms for fully dynamic networks, which would also have to deal with link insertions, node failures or departures, and node insertions. Link failures are of particular concern because if an adversary can take control of a link or links in a network—either physically in a wired network or through attacks such as the wormhole attack in wireless networks—a well-timed link failure could wreak havoc.

Main Contributions: We consider a rather difficult setting: a network of arbitrary topology with a multiplicity of black holes and black links. The agents initially scattered in arbitrary safe locations are unaware of the network topology, start at arbitrary times, and are totally asynchronous (that is, all their actions take a finite but arbitrary amount of time). The initial location of the agents and the timing and duration of their actions are arbitrary, as determined by an adversary.

In this already difficult setting, we allow network links to fail by disappearing during the computation; the timing, choice, and number of link failures is arbitrary, as determined by an adversary. We assume however that any such failure occurs only when no agent is traversing that link, and that the failures do not disconnect the safe part of the network (otherwise the DGE problem is clearly unsolvable).

The main contribution of this paper is the proof that it is still possible to solve the DGE problem in spite of such dynamic link failures. Interestingly, this can be done with the same optimal number $f+1$ of agents as in the fault-free case. The proof is constructive. We present a protocol that using $f+1$ agents solves this new DGE with link deletions (DGE-LD) problem. Not surprisingly, the protocol, based on the fault-free algorithm from [12], is rather complex; interestingly, its cost is $O(nm^2)$ moves, showing that the price of *fault-tolerance* is at most a factor $O(m)$ from the optimal cost without faults.

This is the first result about fault-tolerant computations in dangerous graphs.

The rest of the paper is structured as follows. We describe the model and introduce the terminology in Section 2. We describe the algorithm in detail in Section 3. Finally, we prove the correctness and analyze the complexity of the algorithm in Section 4. Some proofs are omitted due to space limitations.

2 Model

We model the network as a simple undirected graph $G = (V, E)$ with $n = |V|$ vertices or nodes and $m = |E|$ edges or links. The edges incident at a node are locally labeled with distinct values (port numbers). There is a set A of $k = |A|$ agents working in G. They all follow the same protocol or algorithm. Each has a distinct id, its own memory, and can move from node to neighbouring node. They start scattered in the network at arbitrary times. They move and compute asynchronously, meaning that all their actions take a finite but unpredictable amount of time. The initial locations of the agents as well as the timing of their actions are determined by an adversary.

The agents communicate with each other using shared memory in the form of whiteboards located at each node. Each node's whiteboard can be accessed in fair mutual exclusion by the agents resident on that node. The mutual exclusion property allows the agents to operate as if the links are first in, first out (FIFO) and as if the nodes have unique ids. Without loss of generality, we assume that the links and nodes have these properties.

In the network are sets of black holes and black links, nodes and links that eliminate agents arriving at or traversing them without leaving a discernible trace. Let $V_B \subset V$ be the set of black holes and $E_B \subseteq E$ be the set of black links. All other nodes and links are said to be safe. Let $E_I = \{[u, v] \in E : u, v \in V_B\}$ be the set of inaccessible links, black or safe, connecting pairs of black holes. Let $F_B = \{[u, v] \in E : u \in V \setminus V_B \wedge v \in V_B\}$ be the set of frontier links, black or safe, connecting safe nodes to black holes. We can now define the safe portion of the network as $G_S = (V_S, E_S)$, where $V_S = V \setminus V_B$ is the set of safe nodes and $E_S = E \setminus (E_B \cup E_I \cup F_B)$ is the set of safe links. The choice of the sets V_B and E_B is made by an adversary; however, the safe portion of the graph is connected (the problem is otherwise unsolvable). The value $n_S = |V_S|$ is known to the agents (otherwise termination is impossible). Additionally, since at least one agent must survive, the number of agents must be greater than the number f of links incident on a safe node exploring which an agent will die: $f = |F_B| + 2|E_B \setminus (E_I \cup F_B)|$; thus we assume that there are $k \geq f + 1$ agents.

In this setting, we allow network links to fail by disappearing during the computation. An edge failure is locally detectable at an incident node only in the sense that, if information about that edge (identified by its port number) is written on the whiteboard, an agent can notice the absence of an edge with such a port number; if no information is written, it is like the edge never existed. The timing, choice, and number of link failures is arbitrary, as determined by an adversary. However, any such failure occurs only when no agent is traversing that link, and the failures do not disconnect the safe part of the network (otherwise, the DGE problem is clearly unsolvable). Note that allowing failures during the execution of the algorithm is not equivalent to removing the links that will fail before the algorithm starts. Any solution to exploration in a dangerous graph requires the team of agents to coordinate their search. A link failure during the execution can severely disrupt this coordination.

The dangerous graph exploration with link deletions (DGE-LD) problem is for a team of agents to visit every accessible link in a network and, within finite time, to mark locally all frontier links F_B and accessible black links $E_B \setminus E_I$ as dangerous. During the execution of the algorithm, an adversary can delete any link as long as no agents are traversing it. We say that the problem is solved if, within finite time, at least one agent survives, all accessible links have been visited, all frontier links and accessible black links are marked locally as such, and all surviving agents enter a terminal state.

3 The Algorithm

We present an algorithm, *ExploreDG-LD*, that solves the DGE-LD problem. We start by describing the basic work activities of exploring, verifying, and merging. We then describe how an agent deals with deletions during its work.

3.1 Overview

In general, algorithm *ExploreDG-LD* works as follows. The agents build spanning trees of the safe area starting from their homebases in the *exploration* process. The root of each tree contains coordinating information for the agents working on the same tree. The *cautious walk* technique is used during exploration to ensure that only one agent is eliminated per frontier link and at most two agents are eliminated per black link. The *verification* process is used to detect when a newly explored link connects two trees. When two trees are found to be adjacent, they are merged in the *merging* process. An agent terminates the algorithm when the current tree contains n_S nodes and there is no verification or exploration work left. In the absence of link deletions, algorithm *ExploreDG-LD* is very similar in structure to the algorithm presented in [12] and solves the traditional fault-free DGE problem.

Link deletions obviously complicate the entire process, in particular the processes of building the trees and connecting them together. Some deletions—such as the deletion of an unexplored link or an inaccessible link between two black

holes—have no effect on the execution of the algorithm. On the other hand, the deletion of a tree link can have a significant effect. The trees that the agents build out from their homebases provide safe paths through the safe portion of the network, G_S. By eliminating a tree link, the adversary can cut an agent or agents off from access to the coordinating information at the root of the tree in which they are working.

In the following, we describe the algorithm from the point of view of an agent and with the help of Figures 1–7, which show the movement of the agent from the time it starts on a single work task until it finishes that task. The square numbers in each of the diagrams refer to the cases where no deletion is encountered. The circled letters in subsequent diagrams refer to cases where a deletion is encountered.

3.2 Operations without Deletions

In the absence of link deletions, algorithm *ExploreDG-LD* solves the traditional fault-free DGE problem using a logical structure similar to the algorithm presented in [12], which is however not fault-tolerant. Let us discuss the structure.

When an agent a first wakes up, it enters the *initialization* phase. It accesses the whiteboard of its homebase to see if the node has a root marker. If there is no root marker then agent a creates one. The *root marker* contains a map of the tree rooted at the node, as well as all the information needed to find verification and exploration work in the tree. In fact, every node visited by an agent has the root marker for the subtree of which it is the root, even if that subtree is only the node itself. We say that a root marker is *active* if its node has no parent; otherwise, we say that a root marker is *passive*. Only active root markers are used to coordinate work and, as we will see, a passive root marker only becomes active because of the deletion of the link to its parent in the tree.

After initialization, the agent enters the *main loop* of the algorithm and continues until the termination conditions are met. Each round, the agent looks for work in the active root marker. If the current node does not have an active root marker then the agent "grabs" the active root marker by following the parent pointers at each node until the agent finds it. The agent first looks to see if there is any verification work in the root marker. If there is no verification work then it looks for exploration work. Finally, if there no exploration work then it waits until work arrives, following the active root marker if it moves because of a merger. We describe the work of the agent starting first with exploration, then verification, and then merging.

Exploration is the work of exploring every accessible link in the network using cautious walk. Agent a in tree T with root r, as shown in Figure 1, chooses a link $[u, v]$ for exploration and takes the tree path from r to u. The agent updates all the passive root markers along the path noting that $[u, v]$ is being explored. It then uses the *cautious walk* technique to test if $[u, v]$ is safe. It marks as *dangerous* the port on u leading to $[u, v]$ and then traverses $[u, v]$ to v. If $[u, v]$ is a black link or v is a black hole then the agent is eliminated as shown in case 1 in Figure 1. The port on u remains marked as dangerous and no other agent can enter it to be eliminated.

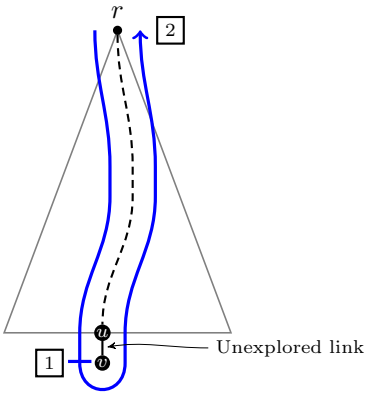

Fig. 1. Exploration of unexplored link $[u, v]$ with no deletions. Cases include encountering a black hole or black link (1), and successful return to its own root (2) (possible movement of r due to mergers is not shown).

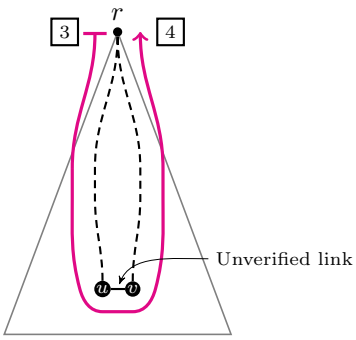

Fig. 2. Verification of internal link $[u, v]$ with no deletions. Cases include the successful verification of the link (3 , 4).

If the agent is not eliminated then it checks to see if v has been previously visited. If v has not been previously visited then a marks it as visited on its whiteboard, creates a parent pointer pointing towards u, and creates a root marker for the subtree rooted at v. It completes the cautious walk by returning to u and marking the port to $[u, v]$ as *explored*. It returns from u to r (or wherever the active root marker is) by grabbing the root marker, adding v to all the passive root markers along the way, marking $[u, v]$ for verification, and marking v's other links for exploration. If v has been previously visited then a completes the cautious walk and returns to r marking $[u, v]$ for verification in all the root markers along the way.

A safe return to r is shown as case 2 in Figure 1. Note that because of merging, which we describe below, the active root marker may have moved. In this case, the agent continues following the parent pointers up the tree, adding the information about v to each root marker it passes, until it reaches the active root marker. Its exploration is then done.

Verification is the work of determining whether every safely explored link is internal or external to the tree in which the agent is working. If an external link is found then the agent may try to merge the two trees connected by the link.

We look first at the verification of internal links as shown in Figure 2. Agent a working in tree T with root r chooses link $[u, v]$ for verification. By construction, $[u, v]$ can only be marked for verification if $u \in T$ and $[u, v]$ has already been explored. The agent needs to determine if $v \in T$ or $v \notin T$. If v is in the map of T's active root marker at r then the agent knows that $[u, v]$ is internal and it does not even have to leave r to complete the verification, which is shown as

case 3 in Figure 2. Case 3 always occurs for tree links that have been marked for verification.

If v is not in the map, it could be because $v \notin T$ or the agent from T that explored v has not yet returned to the active root marker. For an internal link, it must be the latter case. Agent a traverses to the root of v's tree T' with root r' to determine if $T \subseteq T'$ (since r may have been merged with r' during the agent's traversal). The agent traverses from r to u, across $[u, v]$, and then from v to r'. Since we can assume that the links are FIFO and we are looking at internal link verification, we know that the agent that explored v must reach the active root marker before a reaches it and a marks the link as internal on its return, which is shown as case 4 in Figure 2. Agent a's verification of $[u, v]$ is finished.

Unlike during exploration, agent a does not mark $[u, v]$ as being verified on all the passive root markers on its traversal from r to u and v to r', where $T \subseteq T'$. Instead, the agent checks to see if v is in the map of the subtree rooted at each passive root marker it passes on the path from v to r'. If it is then a marks the link as internal to that subtree. Otherwise, a does nothing. Note that all the root markers from r to u already had $[u, v]$ marked for verification by the agent that explored $[u, v]$ and the same is true for the link $[v, u]$ on the path from r' to v. As a result, if a deletion were to create a new tree below the point where both u and v are in the same subtree, the link would already be marked for verification again in the new active root marker.

We now look at the verification of external links as show in Figure 3. Agent a working in tree T with root r chooses link $[u, v]$ for verification. In the external case, node v belongs to some tree T' with root r', where $T \not\subseteq T'$. The agent traverses from r to u, across $[u, v]$, and from v to r' using parent pointers to find the active root marker in T', which is shown as case 5 in Figure 3. Along the way it marks $[v, u]$ for verification in every root marker, if it is not already there. The agent must now decide whether to merge T' with T. If $\mathrm{id}(r') > \mathrm{id}(r)$, where id() is a function that returns the id of a node's root marker, it picks up the active root marker to perform a merger. If $\mathrm{id}(r') < \mathrm{id}(r)$, it adds $[v, u]$ to the links in need of verification, if it is not already there, and begins working in tree T'. In either case, agent a's verification of $[u, v]$ is finished.

Merging is the work of adding one tree to another. Agent a has picked up the active root marker in tree T' with root r' and it traverses from r' to v. Along the way, it reverses the parent pointer to point towards the tree's new root and adjusts the maps of the passive root markers along the way to reflect the branch of the subtree lost by the reversal. It then adds $[v, u]$ as a tree link and traverses from u to the active root marker, on r or elsewhere due to mergers, adding T' and its work information to the root markers along the way, including the active root marker. The merger is then finished as shown in case 6 in Figure 3.

A reader familiar with the technique used in [12] would note two crucial differences: unlike [12], in *ExploreDG-LD* every node visited by an agent has a root marker, and there are no restrictions on the number of verifying agents for the same tree. Precisely these two factors enable the agents to cope with link failures and in particular to avoid deadlocks.

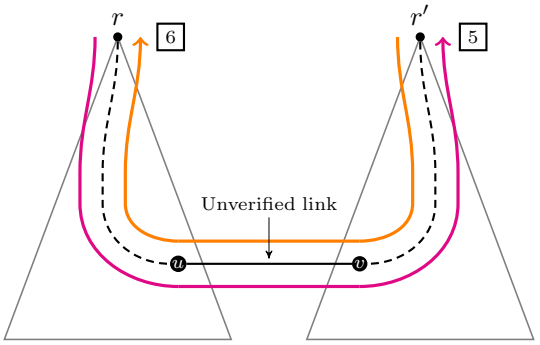

Fig. 3. Verification of external link $[u, v]$ and merging across link $[v, u]$ with no deletions. Cases include successfully verifying the link is external (5), and successfully merging the two trees (6) (possible movement of r or r' due to mergers is not shown).

3.3 Operations with Deletions

There are certain types of deletion that an agent never encounters or which have very little effect on the agent. For instance, the deletion of an inaccessible link or the deletion of a link between two unvisited nodes would have no effect on the agent's execution of the algorithm. The deletion of a known non-tree link is worth noting in the agent's map if it passes by it, but it does not have an effect on the actions taken by the agent. As a result, we are only concerned with the deletion of a tree link or of a link being explored or verified. When we discover such a deletion, we work around it while taking steps to repair the damage. The actions taken often depend more on what was deleted than on the work being performed by the agents, so many of the deletion handling cases overlap. We look at the actions taken by the agent in each case and note any task specific differences.

As a consequence of how we have defined work—the exploration or verification of a link or the merging of one root marker with another—it is only possible for an agent to encounter two deletions during a single piece of work. It can encounter a single deletion either on its way away or back towards the root of its own tree. It can only encounter two deletions if it encounters one on the way away from the root of its own tree and another on the way back. If an agent is unable to return to its own root, it simply starts working wherever it is. The cases presented below take into account both possibilities, one deletion or two, where necessary. In each case, we describe what the agent does when its work is "interrupted" by a deletion.

The cases cover the following work. Let r, r', and r'' be the roots of trees T, T', and T'', respectively. For exploration, an agent a is exploring link $[u, v]$, where $u \in T$. For internal verification, an agent a is verifying link $[u, v]$, where $u \in T$, $v \in T'$, and $T \subseteq T'$. For external verification, an agent a is verifying link $[u, v]$, where $u \in T$, $v \in T'$, and $T \neq T'$. For merging, an agent a is merging

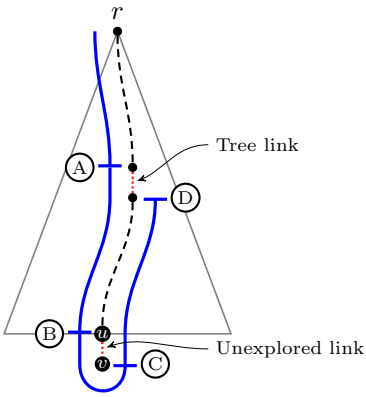

Fig. 4. Exploration of unexplored link $[u, v]$ with one deletion. Cases include the deletion of a tree link (Ⓐ, Ⓓ) and the deletion of the unexplored link, (Ⓑ, Ⓒ).

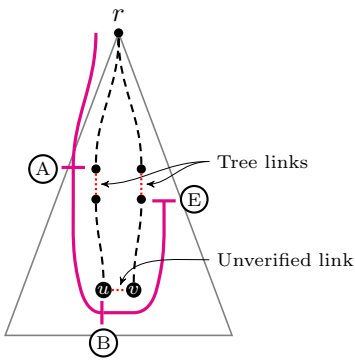

Fig. 5. Verification of internal link $[u, v]$ with one deletion. Cases include the deletion of a tree link (Ⓐ, Ⓓ, Ⓔ), and the deletion of the unverified link (Ⓑ).

across link $[v, u]$, where $v \in T'$ or $v \in T''$ depending on the number of deletions, $u \in T$, $T'' \subset T'$, and $T \neq T'$. Let $[x, y]$ be a tree link on the path from the root to the link being worked on, where x is closest to the root and y is closest to the link.

Case Ⓐ in Figures 4 to 6 applies to an exploring or verifying agent that finds a tree link $[x, y]$ deleted on the path from r to u. The agent returns from x to the active root marker deleting the subtree rooted in x from the map of every root marker along the way. The agent then looks for new work.

Case Ⓑ in Figures 4 to 6 applies to an exploring or verifying agent that finds that the link $[u, v]$ it is meant to explore or verify has been deleted. The agent returns from u to the active root marker marking the deletion on the map of every root marker along the way. The agent then looks for new work.

Case Ⓒ in Figure 4 applies to an exploring agent that finds that $[u, v]$ has been deleted after it has traversed it during the first step of its cautious walk. If v is a new node, the agent creates a root marker and begins working in the new tree rooted at v. If v has already been visited, the agent traverses from v to the active root marker in v's tree. The agent then looks for new work.

Case Ⓓ in Figures 4 and 7 applies to an exploring agent that finds a tree link $[y, x]$ deleted on the path to r. The agent starts working for the new active root marker at y.

Case Ⓔ in Figures 5 and 6 applies to a verifying agent that finds a tree link $[y, x]$ deleted on the path from v to the root of v's tree. If $\mathrm{id}(y) > \mathrm{id}(r)$ then the agent picks up y's root marker and merges it. If $\mathrm{id}(y) < \mathrm{id}(r)$ then the agent adds $[v, u]$ to the links to be verified, if it is not already there, and starts working for the new active root marker at y.

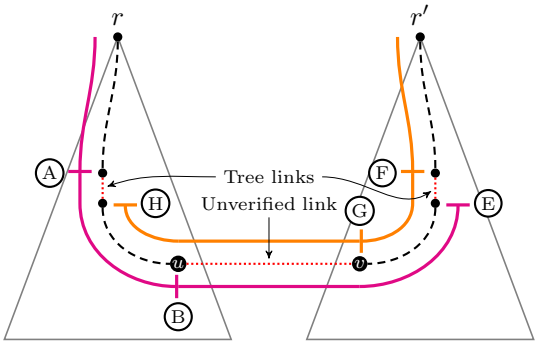

Fig. 6. Verification of external link $[u, v]$ and merging across link $[v, u]$ with one deletion. Cases for verifying (Ⓐ, Ⓑ, Ⓔ) and merging (Ⓕ, Ⓖ, Ⓗ) include the deletion of a tree link (Ⓐ, Ⓔ, Ⓕ, Ⓗ) and deletion of the unverified link, (Ⓑ, Ⓖ).

Case Ⓕ in Figures 6 and 7 applies to a merging agent that finds a tree link $[x, y]$ deleted on the path to v. The agent deletes the subtree rooted in y from the now active root marker on x and starts working there.

Case Ⓖ in Figures 6 and 7 applies to a merging agent that finds the link $[v, u]$ deleted. The agent starts working for the active root marker at v.

Case Ⓗ in Figures 6 and 7 applies to a merging agent that finds a tree link $[y, x]$ deleted on the path from u to r. The agent starts working for the new active root marker at y.

4 Correctness and Complexity

4.1 Absence of Failures

We first prove that, in absence of failures, algorithm *ExploreDG-LD* is a correct solution to the DGE problem. The proof follows a series of lemmas.

Lemma 1. *In the absence of deletions, an agent that is verifying will finish verifying within finite time.*

Lemma 2. *In the absence of deletions, an agent that is exploring will finish exploring within finite time.*

From Lemmas 1 and 2 it follows that:

Lemma 3. *In the absence of deletions, at any time, there is at least one agent alive that is not waiting.*

Lemma 4. *In the absence of deletions, every link in $E \backslash E_I$ is eventually explored and those in G_S are also verified.*

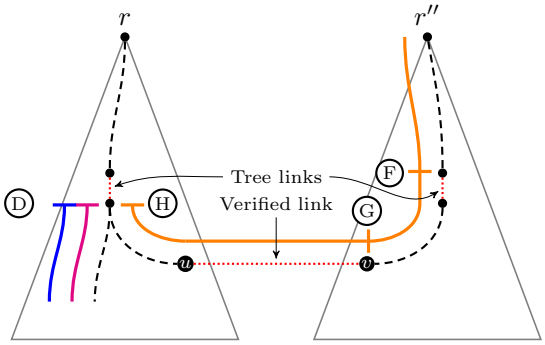

Fig. 7. Agent encounters second deletion during exploration, verification, or merging. Cases include, for returning exploring and verifying agents, the deletion of a tree link (Ⓓ), and, for merging agents, the deletion of tree links (Ⓕ, Ⓗ) and the deletion of the verified link (Ⓖ).

Lemma 5. *In the absence of deletions, the tree of a verifying agent that becomes an exploring agent is merged within finite time.*

Proof. When a verifying agent verifies an external link and finds the tree on the other side has a lower id than its own tree, it starts working for that tree. Let a_2 be the verifying agent from tree T_2 with root r_2 that is verifying edge $[v_2, u_1]$ between T_2 and T_1 and has arrived on root r_1. Let $\mathrm{id}(r_1) < \mathrm{id}(r_2)$. Since T_1 has a lower id root marker, a_2 marks the edge it was verifying, but in the opposite direction, $[u_1, v_2]$, for verification in T_1's root marker, and then starts working there. Without loss of generality, assume that all other agents are currently working on exploration and there are no other links to be verified. Agent a_2 immediately chooses to verify $[u_1, v_2]$ and returns to r_2. Since T_2 has a higher id root marker, agent a_2 picks it up and returns to r_1 to merge T_2's root marker with T_1's. Hence the lemma follows.

While it is possible that other agents are already trying to merge the two trees, we know that the mergers are only performed by agents from lower id trees, so we still have a finite number of mergers that can take place and no cycle of mergers can emerge.

We can now prove the following.

Theorem 1. *Algorithm* ExploreDG-LD *correctly solves the* DGE *problem in the absence of failures.*

4.2 Dynamic Link Failures

We now prove that algorithm *ExploreDG-LD* is also a correct solution to the DGE-LD problem. That is

Theorem 2. *Algorithm* ExploreDG-LD *correctly and within finite time solves the* DGE-LD *problem by constructing a rooted spanning tree of G_S, marking all safe edges as such, and marking all ports in G_S leading to a black hole or to a black edge as dangerous. The total number of moves by the agents is at most $O(k^2 \cdot n_S + n_S \cdot m + k \cdot n_S \cdot D)$.*

To prove the theorem, we prove the correctness of algorithm *ExploreDG-LD* by showing that, while deletions can slow the algorithm, they cannot stop the algorithm from making progress.

We start by noting that because of our use of cautious walk, a team size of $k = f + 1$, where $f = |F_B| + 2|E_B \setminus (E_I \cup F_B)|$ is the number of links incident on safe nodes whose traversal will cause an agent to be teliminated, and the assumption that deletions do not eliminate agents, we can say that at any time, there is always one agent alive.

We next look at work that is aborted as the result of deletion. We say that a work task—exploration, verification, or merging—is *aborted* if the agent is unable to reach the link being explored, verified, or merged over, respectively.

Lemma 6. *Within finite time, an agent completes aborted work.*

Note that in the case of exploration and verification, the active root marker may have moved farther away due to a merger or closer due to a deletion.

We now show that there is always at least one edge leading out of any tree or subtree explored by the agents that cannot be deleted by the adversary.

Let G_E be the explored portion of the graph. Let $G_{SD} = (V_{SD}, E_{SD})$ be the safe portion of the graph G_S after the adversary has performed all its deletions. Let $T = (V_T, E_T)$ be any tree or portion thereof, where $V_T \neq \emptyset$, built by the agents during the algorithm. We define a tree *border* edge as any safe edge connecting two safe nodes where one end is in the tree and one is not in the tree. Let the set of tree border edges be $E_{TF} = \{e \in E_S : e = [u, v] \wedge u \in V_T \wedge v \notin V_T\}$. For any such tree or subtree thereof that covers less than all the nodes, we show that there is a tree border edge that cannot be deleted.

Lemma 7. *For any tree or subtree $T \subseteq G_E$, where $V_T \subset V_S$, there is a safe edge $e \in E_{TF}$ such that $e \in E_{SD}$.*

Proof. By contradiction, assume that no such edge $e \in E_{TF} \cap E_{SD}$ exists. Since we have assumed that no edge is in E_{TF} and E_{SD}, the adversary deletes the all edges in E_{TF}. Since $V_T \subset V_S$, T cannot span the entire safe portion of the graph. As a result, the deletions of all the edges in E_{TF} disconnects T from the rest of the graph, contradicting our assumption that deletions do not disconnect the safe portion of the graph.

We now focus on work in the trees created by the agents. Each tree has an active root marker. We show that, with one exception, all tree border edges are marked for work in the tree's active root marker, being worked on, or become internal within finite time. The one exception comes as the result of the deletion of a tree link. For the links that become internal, we assume that there is an agent

available to do the work that leads to this outcome. We show later that such an agent eventually becomes available.

When the adversary deletes a tree link, it creates a subtree in the tree where the deletion takes place, even if that subtree is only a single node. We call a subtree created in this way a *deletion subtree* and the tree to which the deleted edge currently belongs the *original tree*.

Let T_O be the original tree and T_S be the deletion subtree created by the deletion of tree edge e_D. Let E_{OS} be the tree border edges, if any, that connect T_O to T_S and E_{SO} be the same edges in the opposite direction. Let $E_{OS} = E_{SO} = \emptyset$, if the tree is not an original tree or a deletion subtree, respectively.

Lemma 8. *For any tree $T \subseteq G_E$ with an active root marker, where $V_T \subset V_S$, all edges $e \in E_{TF} \setminus E_{OS}$ are marked for work in T's active root marker, being worked on, or are marked as internal within finite time.*

We now consider what happens to the links in E_{OS} . The reason that they are not marked for work is because they are marked internal to T_O even though they are now external due to the deletion of e_D. By contrast, the links of E_{SO}, which are the same links as E_{OS} but in the opposite direction, are automatically marked for verification in T_S's active root marker. By construction, when an agent verifies an internal link, it only marks the link as internal in the passive root markers on its path if both ends are in the subtree rooted in the passive root marker; otherwise, the link remains marked for verification. The set E_{SO} are exactly those links in T_S that are internal to T_O before the deletion but were marked for verification in T_S's passive root marker when it became active.

We now prove that the links in E_{OS} are guaranteed to be marked for work in T_O's active root marker if there is previously reported work in T_S, there is an agent working in T_O, and there are no other connections to T_S; otherwise, there is no guarantee they are marked for work.

Lemma 9. *Let $e_W \in T_S \setminus E_{SO}$ be marked for work in T_S and let T_S have no agents working for it. Let an agent a in T_O choose to work on e_W. Within finite time, the links in E_{OS} are marked for work in the active root marker of T_O.*

This result suggests that there is two circumstances when a deletion subtree is never detected: the subtree contains no agents and no work except for the links in E_{SO}, or the subtree contains agents exploring frontier or black links and no other work except for the links in E_{SO}. We say such a deletion subtree is *empty*.

Lemma 10. *The links E_{OS} leading to an empty deletion subtree are never marked for work.*

These *empty* trees do not affect the correctness of the algorithm.

Lemma 11. *The failure to detect a deletion subtree with no other work than E_{SO} does not affect the correctness of the algorithm.*

The same is true of the deletion of non-tree links, although they can and do affect the complexity of the algorithm.

Lemma 12. *The deletion of a non-tree link does not stop the agent from completing its current work.*

We now need to deal with the assumption in Lemma 9 that there must be an agent in T_O that chooses to work on $e_W \in T_S$ in order to guarantee that the links in E_{OS} are marked for work and in Lemma 8 that there is an agent available to verify $[v, u]$ in the deletion subtree T_i, where $i \geq 1$. We show that before termination there must always be an agent working in the network and because that agent must eventually work on edges that cannot be deleted, every tree is eventually worked on.

Lemma 13. *At any time before termination, at least one agent is performing work.*

Corollary 1. *An agent is eventually available to do the work in T_O described in Lemma 9 and in deletion subtree T_i, where $i \geq 1$, described in Lemma 8.*

We can now prove the correctness of the algorithm.

Lemma 14. *Within finite time, all accessible links have been visited and all surviving agents terminate.*

Lemma 15. *After at most $O(k^2 \cdot n_S + n_S \cdot m + k \cdot n_S \cdot D)$ moves, all agents that are still alive terminate.*

The proof of the main result, Theorem 2, now follows.

References

[1] Balamohan, B., Flocchini, P., Miri, A., Santoro, N.: Time Optimal Algorithms for Black Hole Search in Rings. In: Wu, W., Daescu, O. (eds.) COCOA 2010, Part II. LNCS, vol. 6509, pp. 58–71. Springer, Heidelberg (2010)

[2] Balamohan, B., Flocchini, P., Miri, A., Santoro, N.: Improving the Optimal Bounds for Black Hole Search in Rings. In: Kosowski, A., Yamashita, M. (eds.) SIROCCO 2011. LNCS, vol. 6796, pp. 198–209. Springer, Heidelberg (2011)

[3] Chalopin, J., Das, S., Labourel, A., Markou, E.: Tight Bounds for Scattered Black Hole Search in a Ring. In: Kosowski, A., Yamashita, M. (eds.) SIROCCO 2011. LNCS, vol. 6796, pp. 186–197. Springer, Heidelberg (2011)

[4] Chalopin, J., Das, S., Santoro, N.: Rendezvous of Mobile Agents in Unknown Graphs with Faulty Links. In: Pelc, A. (ed.) DISC 2007. LNCS, vol. 4731, pp. 108–122. Springer, Heidelberg (2007)

[5] Cooper, C., Klasing, R., Radzik, T.: Searching for Black-Hole Faults in a Network Using Multiple Agents. In: Shvartsman, M.M.A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 320–332. Springer, Heidelberg (2006)

[6] Czyzowicz, J., Dobrev, S., Královič, R., Miklík, S., Pardubská, D.: Black Hole Search in Directed Graphs. In: Kutten, S., Žerovnik, J. (eds.) SIROCCO 2009. LNCS, vol. 5869, pp. 182–194. Springer, Heidelberg (2010)

[7] Czyzowicz, J., Kowalski, D., Markou, E., Pelc, A.: Complexity of searching for a black hole. Fund. Inform. 71(2,3), 229–242 (2006)

[8] Dobrev, S., Flocchini, P., Královič, R., Ružička, P., Prencipe, G., Santoro, N.: Black hole search in common interconnection networks. Networks 47(2), 61–71 (2006)
[9] Dobrev, S., Flocchini, P., Prencipe, G., Santoro, N.: Searching for a black hole in arbitrary networks: Optimal mobile agents protocols. Distrib. Comput. 19(1), 1–19 (2006)
[10] Dobrev, S., Flocchini, P., Prencipe, G., Santoro, N.: Mobile search for a black hole in an anonymous ring. Algorithmica 48(1), 67–90 (2007)
[11] Flocchini, P., Ilcinkas, D., Santoro, N.: Ping pong in dangerous graphs: Optimal black hole search with pebbles. Algorithmica 62(3–4), 1006–1033 (2012)
[12] Flocchini, P., Kellett, M., Mason, P.C., Santoro, N.: Map construction and exploration by mobile agents scattered in a dangerous network. In: Proceedings of IPDPS (2009)
[13] Flocchini, P., Kellett, M., Mason, P.C., Santoro, N.: Searching for black holes in subways. Theory Comput. 50(1), 158–184 (2012)
[14] Glaus, P.: Locating a Black Hole without the Knowledge of Incoming Link. In: Dolev, S. (ed.) ALGOSENSORS 2009. LNCS, vol. 5804, pp. 128–138. Springer, Heidelberg (2009)
[15] Klasing, R., Markou, E., Radzik, T., Sarracco, F.: Hardness and approximation results for black hole search in arbitrary networks. Comput. Sci. 384(2-3), 201–221 (2007)
[16] Klasing, R., Markou, E., Radzik, T., Sarracco, F.: Approximation bounds for black hole search problems. Networks 52(4), 216–226 (2008)
[17] Kosowski, A., Navarra, A., Pinotti, C.M.: Synchronous black hole search in directed graphs. Theor. Comput. Sci. 412(41), 5752–5759 (2011)
[18] Královič, R., Miklík, S.: Periodic Data Retrieval Problem in Rings Containing a Malicious Host. In: Patt-Shamir, B., Ekim, T. (eds.) SIROCCO 2010. LNCS, vol. 6058, pp. 157–167. Springer, Heidelberg (2010)
[19] Miklík, S.: Exploration in faulty networks. Ph.d. thesis, Faculty of Mathematics, Physics, and Informaticcs, Comenius University, Bratislava, Slovakia (2010)
[20] Shannon, C.: Presentation of a maze-solving machine. In: Proceedings of the 8th Conference of the Josiah Macy Jr. Foundation (Cybernetics), pp. 173–180 (1951)
[21] Shi, W.: Black Hole Search with Tokens in Interconnected Networks. In: Guerraoui, R., Petit, F. (eds.) SSS 2009. LNCS, vol. 5873, pp. 670–682. Springer, Heidelberg (2009)