Mapping an Unfriendly Subway System

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

[1] School of Information Technology and Engineering, University of Ottawa, Canada
flocchin@site.uottawa.ca
[2] Defence R&D Canada – Ottawa, Government of Canada, Canada
{matthew.kellett, peter.mason}@drdc-rddc.gc.ca
[3] School of Computer Science, Carleton University, Canada
santoro@scs.carleton.ca

Abstract. We consider a class of highly dynamic networks modelled on an urban subway system. We examine the problem of creating a map of such a subway in less than ideal conditions, where the local residents are not enthusiastic about the process and there is a limited ability to communicate amongst the mappers. More precisely, we study the problem of a team of asynchronous computational entities (the mapping agents) determining the location of black holes in a highly dynamic graph, whose edges are defined by the asynchronous movements of mobile entities (the subway carriers). We present and analyze a solution protocol. The algorithm solves the problem with the minimum number of agents possible. We also establish lower bounds on the number of carrier moves in the worst case, showing that our protocol is also move-optimal.

1 Introduction

Computer networks are not necessarily safe. They often contain dangerous elements such as computers that have undetectably crashed or network equipment that is malfunctioning or misconfigured. There is a large body of research into distributed algorithms for finding these faults, which are often referred to in the literature as black holes and black links, or, more generally, as dangerous elements (e.g., see [2–10, 12–15]). All these investigations on finding dangerous elements assume that the network itself is static and connected.

There are several classes of networks that have dynamic topologies that change as a function of time, and that might be disconnected at times. They include wireless mobile ad hoc networks where the network's topology may change dramatically over time due to the movement of the network's nodes; sensor networks where links only exist when two neighbouring sensors are awake and have power; and vehicular networks, similar to mobile ad hoc networks, where the topology changes constantly as vehicles move. Indeed there is a large amount of research on these networks (which are called *delay-tolerant*, *challenged*, *opportunistic*, *evolving*, etc.) focusing mostly on broadcasting and routing (e.g., see [1, 16–19]). At least one study [11] has looked at how to *explore* one class of these networks: periodically-varying graphs. In the periodically-varying graph

(PV graph) exploration problem, agents ride carriers between sites in the network. A link only exists between sites when a carrier is passing between them. The agents explore the network by moving from carrier to carrier when their routes meet at a site. These dynamic networks are no less prone to faults than static networks. The question is, how does one find a dangerous element in a time-varying network?

Imagine a group of tourists are visiting the unfriendly capital of Dystopia—perhaps not the best travel destination. Although the city has a subway system, there are no maps because the local population wants to limit their capital's appeal to tourists (Dystopians are grumpy by nature). The tourists want to map the subway system without the local population knowing. They agree on a location in each station where they will leave notes for each other. The problem is that there are good stations and there are bad stations. Tourists arriving at good stations can easily leave notes for each other. Tourists arriving at bad stations get lost when they try to leave notes, eventually giving up on the whole map-making process. The group wants to complete the map while minimizing the number of their group lost to the frustration of the bad stations.

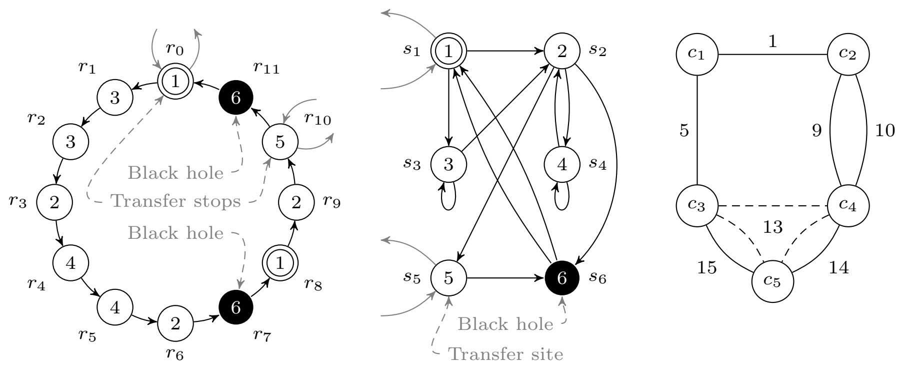
We look at black hole search in a class of time-varying network based on a similar scenario. Instead of tourists, we have agents. Instead of subway trains, we have carriers. Instead of stations, we have sites, where the bad stations are black hole sites that eliminate the agents arriving at them without leaving a discernable trace. The class of networks described by this *subway model* is much larger than the set of real subway systems. We look at the asynchronous version of the black hole search problem where the calculations of the agents and movements of the carriers take a finite but unpredictable amount of time. To measure complexity in this environment, we look at the number of carrier moves needed to complete the search. We show that our solution has a complexity $O(k \cdot n_C^2 \cdot l_R) + O(n_C \cdot l_R^2)$ carrier moves where n_C is the number of carriers and l_R is the length of the longest carrier route. We prove that the lower bound on the worst case complexity is $\Omega(k \cdot n_C^2 \cdot l_R) + \Omega(n_C \cdot l_R^2)$ carrier moves, making our solution worst-case optimal.

2 Model

We consider a set C of n_C *carriers* that move among a set S of n_S *sites*. A carrier $c \in C$ follows a route $R(c)$ between all the sites in its *domain* $S(c) = \{s_0, s_1, \ldots, s_{n_S(c)-1}\} \subseteq S$. A carrier's *route* $R(c) = \langle r_0, r_1, \ldots, r_{l(c)-1} \rangle$ is a cyclic sequence of *stops*: after stopping at site $r_i \in S(c)$, the carrier moves to $r_{i+1} \in S(c)$, where operations on the indices are modulo $l(c) = |R(c)|$ called the *length* of the route. Carriers move *asynchronously*, taking a finite but unpredictable amount of time to move between stops. Each carrier is labelled with a distinct id and the length of its route. A route is *simple* if $n_S(c) = l(c)$, and *complex* otherwise. A *transfer site* is any site that is in the domain of two or more carriers; each transfer site is labelled with the number of carriers stopping at it. A carrier's route $R(c) = \langle r_0, r_1, \ldots, r_{l(c)-1} \rangle$ defines an edge-labelled directed multigraph $\boldsymbol{G}(c) = (S(c), \boldsymbol{E}(c), \lambda(c))$, called *carrier graph*, where there is an edge labeled $(c, i+1)$ from r_i to r_{i+1}, and the operations on indices and inside labels are modulo $l(c)$. The entire network is then represented by the edge-labelled directed multigraph $\boldsymbol{G} = (R, \boldsymbol{E}, \lambda)$, called *subway graph*, where $R = \cup_{c \in C} R(c)$, $\boldsymbol{E} = \cup_{c \in C} \boldsymbol{E}(c)$, and $\lambda = \{\lambda(c) : c \in C\}$. Associated with the subway graph is the *transfer graph* of $\boldsymbol{G}$ which we define as the edge-labelled undirected multigraph $H(\boldsymbol{G}) = (C, E_T)$ where the nodes are the carriers and, $\forall c, c' \in C, s \in S$, there is an edge between c and c' labeled s iff $s \in S(c) \cap S(c')$, i.e., s is a transfer site between c and c'. In the following, when no ambiguity arises, we will omit the edge labels in all graphs.See Figure 1c. Working in the network is a team A of k computational *agents* that start at unpredictable times from the same site, called the *homebase*. Agents can only communicate with each other using shared memory, available at each site in the form of a *whiteboard*, which is accessed in fair mutual exclusion. The agents are *asynchronous* in that they take a finite but unpredictable amount of time to perform computations at a site. All agents execute the same protocol and know the number of carriers n_C. The agents move around the network using the carriers. An agent can move from a carrier to a site (*disembark* at a stop) or from a site to a carrier (*board* a carrier), but not from one carrier to another directly. An agent on a transfer site can board any carrier stopping at it. When on a site, an agent can access the whiteboard, and can read the number of the carriers stopping at that site; furthermore, it can read the labels of any carrier stopping there. When travelling on a carrier, an agent can count the number of stops that the carrier has passed, and can decide whether or not to disembark at the next stop. Once disembarked, the agent can later board the same carrier at the same point in its route.

Among the sites there are $n_B < n_S$ *black holes*: sites that eliminate agents disembarking on them without leaving a discernable trace; black holes do not affect carriers. The *black hole search* (Bhs) problem is that of the agents determining the locations of the black holes in the subway graph. A protocol solves the Bhs problem if within finite time at least one agent survives and all the surviving agents know which *stops* are black holes. There are some basic limitations for the Bhs problem to be solvable: the transfer graph must be connected once the black holes are removed, and the homebase must be a safe site. Hence we will assume these conditions to hold. Because of asynchrony, slow computation by an agent exploring a safe stop is indistinguishable from an agent having been eliminated by a black hole stop, so, with only knowledge of n_C, termination is impossible unless each carrier has at least one safe transfer site. Hence the agents' knowledge of n_C will be assumed.

As in traditional mobile agent algorithms, the basic cost measure used to evaluate the *efficiency* of a Bhs solution protocol is the *size* of the team that is the number k of agents needed by the protocol. Let $\gamma(c) = |\{i : r_i \in R(c)$ is a black hole$\}|$ be the number of black holes among the stops of c; and let $\gamma(\boldsymbol{G}) = \sum_{c \in C} \gamma(c)$, called the *faulty load* of subway graph $\boldsymbol{G}$, be the total number of stops that are black holes. The *faulty load* $\gamma(\boldsymbol{G})$ of subway graph $\boldsymbol{G}$ is the number of stops that are black holes. To solve Bhs, it is obviously necessary to have more agents than the faulty load of the network, i.e. $k > \gamma$. A solution



(a) Route of c_1 ($R(c_1)$). (b) Carrier graph of c_1 ($\boldsymbol{G}(c_1)$). (c) Transfer graph $H(\boldsymbol{G})$.

Fig. 1: A route, the corresponding carrier graph, and a transfer graph (edge labels are the corresponding transfer site ids).

protocol is *agent optimal* if it solves the Bhs problem for $k = \gamma + 1$. The other cost measure is the number of *carrier moves*: when an agent is waiting for a carrier or riding on a carrier, each move made by that carrier is counted as a carrier move for the agent. A solution protocol is *move optimal* in the worst case if the total number of carrier moves incurred by all agents in solving the Bhs problem is the best possible.

3 Exploration Algorithm

In this section we present the proposed algorithm *SubwayExplore*; as we will show later, our algorithm works correctly with any number of agents $k \geq \gamma + 1$.

Informally, algorithm *SubwayExplore* works as follows. All the agents start at unpredictable times from the same site s, called the *homebase*. An agent's work involves visiting a previously unexplored stop on a carrier's route and returning, if possible, to report what was found there. Every carrier is searched from a *work site* and the work sites are organized into a logical *work tree* that is rooted in the homebase. The first agent to access the homebase's whiteboard sets up the homebase as a work site (Sect. 3.1). It and the agents awaking after it then begin to do work by visiting the stops of the carriers stopping at the homebase (Sect. 3.2). If an exploring agent finds a previously unexplored transfer site, the agent "competes" to add the transfer site to the work tree. If the agent succeeds, the transfer site becomes a work site for other carriers stopping at it and the work site from which it was discovered becomes its parent in the work tree (Sect. 3.3). When the carrier that the agent is exploring has no more unexplored stops, it tries to find another carrier with work in the work tree. It looks for work in the subtree rooted in its current work site and if there is no work available it moves to the work site's parent and tries again (Sect. 3.4). An agent terminates if it is at the homebase, there is no work, and there are n_C carriers in the work tree.

3.1 Initialization

When an agent awakes for the first time on the homebase, it tries to initialize the homebase as a work site. Only the first agent succeeds and executes Initialize Work Site. All other agents proceed directly to trying to find work.

The Initialize Work Site procedure is used to set up each work site in the work tree. The procedure takes as input the parent of the work site and the carriers to be worked on or serviced from the work site. For the homebase, the parent is *null* and the carriers to be accessed are all those stopping at s. The procedure initializes the work site's whiteboard with the information needed to find work, do work, and compete to add work. More precisely, when a work site ws is initialized, its parent is set to the work site from which it was discovered (*null* in the homebase's case) and its children are initially *null*. The carriers it will service are added to $C_{subtree}$, the set of carriers in the work tree at and below this work site. The same carriers are also added to C_{work}, the set of carriers in the subtree with unexplored stops, and C_{local}, the set of carriers serviced by this work site. For each carrier c added to C_{local}, the agent setting up the whiteboard creates three sets U_c, D_c, and E_c. The set E_c of *explored stops* is initialized with the work site at $r_0 = ws$ (r_0 is always the work site servicing the carrier). The set U_c of *unexplored stops* is initialized with the rest of the stops on the carrier's route $\{r_1, r_2, \ldots, r_{l(c)-1}\}$, which is possible because each carrier is labelled with its length as well as its id. The set D_c of *stops being explored* (and therefore potentially dangerous sites) is initially empty.

3.2 Do Work

We now discuss how the agents do their exploration of unexplored stops (procedure Do Work shown in Algorithm 1). To limit the number of agents eliminated by black holes, we use a technique similar to the cautious walk technique used in static networks. Consider an agent a on the work site ws of a carrier c that still has unexplored stops, i.e. $U_c \neq \emptyset$. The agent does the following. It chooses an unexplored stop $r \in U_c$ for exploration, removes r from U_c, and adds it to the set D_c of stops being explored. It then takes c to r and disembarks. If the agent survives, it returns to ws using the same carrier c and disembarks. The agent can make the trip back to ws because it knows the index of r and $l(c)$ and can therefore calculate the number of stops between r and ws. At ws, it removes r from D_c and adds it to the set E_c of explored stops. At this point, the agent also adds the site id and any other information of interest. If r is a transfer site and a is the first to visit it (its whiteboard is blank), then, before returning to ws, the agent proceeds as follows. It records on r's whiteboard all the carriers that pass by r, including their id and lengths of their route. It initializes two sets in its own memory: the set of *new carriers* initially containing all the carriers stopping

Algorithm 1 Do work

```
    Agent $a$ is working on carrier $c$ from work site $ws$.
 1: procedure DO WORK(carrier $c$)
 2:     while $U_c \neq \emptyset$ do
 3:         choose a stop $r$ from $U_c$
 4:         $U_c \leftarrow U_c \setminus \{r\}$          ▷ Remove $r$ from the set of unexplored stops
 5:         $D_c \leftarrow D_c \cup \{r\}$          ▷ Add $r$ to the set of stops being explored
 6:         take $c$ to $r$ and disembark
            ▷ If not eliminated by black hole
 7:         if $r$ is a transfer site $\wedge$ whiteboard is blank then
 8:             $a.newC \leftarrow \emptyset$          ▷ Initialize agent's set of new carriers
 9:             $a.existingC \leftarrow \emptyset$          ▷ Initialize agent's set of existing carriers
10:             for each carrier $c$ stopping at $r$ do
11:                 record $c$ on whiteboard
12:                 $a.newC \leftarrow a.newC \cup \{c\}$ ▷ Add carrier to agent's set of new carriers
13:             end for
14:         end if
15:         take $c$ to $ws$ and disembark
16:         $D_c \leftarrow D_c \setminus \{r\}$          ▷ Remove $r$ from the set of stops being explored
17:         $E_c \leftarrow E_c \cup \{r\}$          ▷ Add $r$ to the set of explored stops
18:         if $r$ was a transfer site then
19:             COMPETE TO ADD WORK
20:         end if
21:     end while
22: end procedure
```

at r; and the set of *existing carriers*, initially empty. These sets are used in the next procedure that we discuss: competing to add work.

3.3 Compete to Add Work

When an agent a discovers that a stop r is an unvisited transfer site, that stop is a potential new work site for the other carriers stopping at it. There is a problem, however: other agents may have independently discovered some or all of those carriers stopping at r. To ensure that each carrier has only one associated work site in the work tree, in our algorithm agent a must compete with all those other agents before it can add r as the new work site in the tree for those carriers. We use $C_{subtree}$ on the work sites in the work tree to decide the competition (if any).

Let us describe the actions that agent a performs; let a have just finished exploring r on carrier c_{ws} from work site ws and found that r is a new transfer site. The agent has a set of *new carriers* that initially contains all the carriers stopping at r, a set of *existing carriers* that is initially empty, and is currently on its work site ws. The agent walks up the work tree from ws to s checking the set of new carriers against $C_{subtree}$ on each work site. If a new carrier is not in $C_{subtree}$, the agent adds it. If a new carrier is in $C_{subtree}$, the agent moves it to the set of existing carriers. The agent continues until it reaches s or its set of new carriers is empty. The agent then walks down the work tree to ws. It adds each carrier in its set of new carriers to C_{work} on each work site on the way down to ws. For each carrier in its set of existing carriers, it removes the carrier from $C_{subtree}$ on the work site if it was the agent that added it. When it reaches ws, it removes the existing carriers and if there are no new carriers, it continues its work on c_{ws}. If there are new carriers, the agent adds r as a child of ws and goes to r. At r, the agent initializes it as a work site using the INITIALIZE WORK SITE procedure with ws as its parent and the set of new carriers as its carriers. The agent then returns to ws and continues its work on c_{ws}. The procedure COMPETE TO ADD WORK, shown in Algorithm 2, ensures the following properties:

Lemma 1 *All new work is reported to the root.*

Lemma 2 *If a new carrier is discovered, it is added to the work tree within finite time.*

Lemma 3 *A carrier is always serviced from a single work site.*

3.4 Find Work

Now that we have seen work being done and new work added to the tree, it is easy to discuss how an agent a finds work. When a work site is initialized, its parent is set to the work site from which it was discovered (*null* in the homebase's case) and its children are initially *null*. As mentioned before, each work site has a set C_{work} that contains the carriers in its subtree with unexplored stops.

If C_{work} on the current work site is not empty, an agent a looking for work chooses a carrier c and walks down the tree until it reaches the work site ws servicing c or it finds that c is no longer in C_{work}. Assume that agent a reaches ws without finding c missing from C_{work}. Then a works on c until it is either eliminated by a black hole or U_c is empty. If the agent survives and is the first agent to discover that U_c is empty, it walks up the tree from ws to s removing c from C_{work} along the way. So, it is possible for an agent descending to do work on c to find out before it reaches ws that the work on c is finished. In that case, the agent starts over trying to find work.

If agent a looking for work finds that C_{work} at the current work site is empty, it moves to the work site's parent and tries again. If it reaches the root without finding work but the termination condition is not met (there are fewer than n_C carriers in the work tree), the agent waits (loops) until new work arrives or the termination condition is finally met. The procedure FIND WORK, shown in Algorithm 3, ensures the following property:

Lemma 4 *Within finite time, an agent looking for work either finds it or waits on the root.*

Algorithm 2 Compete to Add Work

Agent a has found a new transfer site r while exploring carrier c_{ws} from work site ws and is competing to add it to the work tree with ws as r's parent.

```
 1: procedure Compete to Add Work
        ▷ Walk up tree
 2:     repeat
 3:         take the appropriate carrier to $parent$ and disembark
 4:         for $c \in a.newC$ do
 5:             if $c \in C_{subtree}$ then
 6:                 $a.newC \leftarrow a.newC \setminus \{c\}$          ▷ Remove from set of new carriers
 7:                 $a.existingC \leftarrow a.existingC \cup \{c\}$    ▷ Add to set of existing carriers
 8:             else
 9:                 $C_{subtree} \leftarrow C_{subtree} \cup \{c\}$
10:             end if
11:         end for
12:     until (on $s$) $\vee$ ($a.newC = \emptyset$)
        ▷ Walk down tree
13:     while not on $ws$ do
14:         for $c \in a.newC$ do
15:             $C_{work} \leftarrow C_{work} \cup \{c\}$          ▷ Add new carriers with work in subtree
16:         end for
17:         for $c \in a.existingC$ do
18:             if $a$ added $c$ to $C_{subtree}$ then
19:                 $C_{subtree} \leftarrow C_{subtree} \setminus \{c\}$      ▷ Remove carrier from subtree set
20:             end if
21:         end for
22:         take appropriate carrier to $child$ in direction of $ws$ and disembark
23:     end while
        ▷ Remove any existing carriers on $ws$
24:     for $c \in a.existingC$ do
25:         if $a$ added $c$ to $C_{subtree}$ then
26:             $C_{subtree} \leftarrow C_{subtree} \setminus \{c\}$          ▷ Remove carrier from subtree set
27:         end if
28:     end for
        ▷ Add any new carriers to the tree with $r$ as their work site
29:     if $a.newC \neq \emptyset$ then
30:         $children \leftarrow children \cup \{r\}$
31:         for $c \in a.newC$ do
32:             $C_{work} \leftarrow C_{work} \cup \{c\}$
33:         end for
34:         take carrier $c_{ws}$ to $r$ and disembark
35:         Initialize Work Site($ws$, $a.newC$)
36:         take carrier $c_{ws}$ to $ws$ and disembark
37:     end if
38:     Do Work($c_{ws}$)                                ▷ Keep working on original carrier
39: end procedure
```

Algorithm 3 Find work

Agent a is looking for work in the work tree. The agent knows n_C, the number of carriers, which is needed for termination. Let ws be the current work site.

```
 1: procedure FIND WORK
        ▷ Main loop
 2:     while (not on $s$) $\vee$ ($C_{work} \neq \emptyset$) $\vee$ ($|C_{subtree}| < n_C$) do
            ▷ Choose carrier to work on and go there
 3:         if $C_{work} \neq \emptyset$ then
 4:             choose carrier $c$ from $C_{work}$
 5:             while ($c \notin C_{local}$) $\wedge$ ($c \in C_{work}$) do
 6:                 take appropriate carrier to child in direction of $c$ and disembark
 7:             end while
 8:             if $c \in C_{local}$ then                          ▷ On the work site servicing $c$
 9:                 DO WORK($c$)
10:                 if $c \in C_{work}$ then         ▷ The first agent to find no work left on $c$
11:                     while not on $s$ do
12:                         $C_{work} \leftarrow C_{work} \setminus \{c\}$
13:                         take appropriate carrier to parent and disembark
14:                     end while
15:                     $C_{work} \leftarrow C_{work} \setminus \{c\}$
16:                 end if
17:             end if
18:         else                                                      ▷ No work in subtree
19:             take appropriate carrier to parent and disembark
20:         end if
21:     end while
22: end procedure
```

3.5 Correctness

Let us now prove the correctness of algorithm *SubwayExplore*. To do so, we need to establish some additional properties of the Algorithm:

Lemma 5 *Let $r_i \in R(c)$ be a black hole. At most one agent is eliminated by stopping at r_i when riding c.*

Lemma 6 *There is at least one agent alive at all times before termination.*

Lemma 7 *An agent that undertakes work completes it within finite time.*

Lemma 8 *If there is work available, an agent eventually does it.*

Lemma 9 *All carriers are eventually added to the tree.*

We can now state the correctness of our algorithm:

Theorem 1 *Protocol* SubwayExplore *correctly and in finite time solves the mapping problem with $k \geq \gamma(\boldsymbol{G}) + 1$ agents in any subway graph $\boldsymbol{G}$.*

4 Bounds and Optimality

We now analyze the costs of our algorithm, establish lower bounds on the complexity of the problem and prove that they are tight, showing the optimality of our protocol.

Theorem 2 *The algorithm solves black hole search in a connected dangerous asynchronous subway graph in $O(k \cdot n_C^2 \cdot l_R + n_C \cdot l_R^2)$ carrier moves in the worst case.*

We now establish some *lower bounds* on the worst case complexity of any protocol using the minimal number of agents.

Theorem 3 *For any α, β, γ, where $\alpha, \beta > 2$ and $1 < \gamma < 2\alpha\beta$, there exists a simple subway graph $\boldsymbol{G}$ with α carriers with maximum route length β and faulty load γ in which every agent-optimal subway mapping protocol $\mathcal{P}$ requires $\Omega(\alpha^2 \cdot \beta \cdot \gamma)$ carrier moves in the worst case. This result holds even if the topology of $\boldsymbol{G}$ is known to the agents.*

Proof. Consider a subway graph $\boldsymbol{G}$ whose transfer graph is a line graph; all α routes are simple and have the same length β; there exists a unique transfer stop between neighbouring carriers in the line graph; no transfer site is a black hole, and the number of black holes is γ. The agents have all this information, but do not know the order of the carriers in the line.

Let $\mathcal{P}$ be a subway mapping protocol that always correctly solves the problem within finite time with the minimal number of agents $k = \gamma + 1$. Consider an adversary $\mathcal{A}$ playing against the protocol $\mathcal{P}$. The power of the adversary is the following: 1) it can choose which stops are transfers and which are black holes; 2) it can "block" a site being explored by an agent (i.e., delay the agent exploring the stop) for an arbitrary (but finite) amount of time; 3) it can choose the order of the carriers in the line graph. The order of the carrier will be revealed to the agents incrementally, with each revelation consistent with all previous ones; at the end the entire order must be known to the surviving agents.

Let the agents start at the homebase on carrier c_1. Let $q = \lceil \frac{k-2}{\beta-2} \rceil$. Assume that the system is in the following configuration, which we shall call *Flip(i)*, for some $i \geq 1$: (1) carrier c_1 is connected to c_2, and carrier c_j ($j < i$) is connected to c_{j+2}; (2) all stops of carriers $c_1, c_2, \ldots, c_i$ have been explored, except the transfer stop r_{i+1}, leading from carrier c_{i-1} to carrier c_{i+1}, and the stop r_{i+2} on carrier c_{i+1}, which are currently being explored and are blocked by the adversary; and (3) all agents, except the ones blocked at stops r_{i+1} and r_{i+2}, are on carrier c_i. See Fig. 2. If the system is in configuration *Flip(i)*, with $i < \alpha - q$, the adversary operates as follows.

(1) The adversary unblocks r_{i+1}, the transfer site leading to carrier c_{i+1}. At this point, all $k-1$ unblocked agents (including the $k-2$ currently on c_i) must move to r_{i+1} to explore c_{i+1} without waiting for the agent blocked at r_{i+2} to come back. To see that all must go within finite time, assume by contradiction

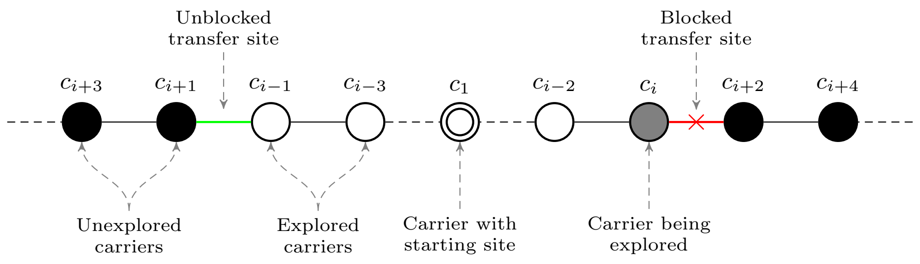


Fig. 2: Transfer graph in the lowerbound proof

that only $1 \leq k' \leq k-2$ agents go to explore c_{i+1} within finite time, while the others never go to r_{i+1}. In this case, the adversary first reveals the order of the carriers in the line graph by assigning carrier c_j to be connected to c_{j+1} for $\alpha > j > i$. Then the adversary chooses the following stops to be black holes: r_{i+2}, the first k' non-transfer stops visited by the k' agents, and other $k-k'-2$ non-transfer stops arbitrarily chosen in those carriers. Notice this can be done because, since $q = \lceil \frac{k-2}{\beta-2} \rceil$, the number of non-transfer stops among these carriers is $q(l-2)+1 \geq k-1$. Thus all k' agents will enter a black hole. Since none of the other agents will ever go to c_{i+1}, the mapping will never be completed. Hence, within finite time all $k-1$ non blocked agents must go to r_{i+1}, with a total cost of $O(k \cdot i \cdot \beta)$ carrier moves.

(2) The adversary blocks each stop of c_{i+1} being explored, until $k-1$ stops are being explored. At that point, it unblocks all those stops except one, r_{i+3}. Furthermore, it makes r_{i+2} the transfer stop leading to carrier c_{i+2}.

Notice that after these operations, the system is precisely in configuration *Flip(i+1)*. Further observe now that, from the initial configuration, when all agents are at the homebase and the protocol starts, the adversary can create configuration *Flip(0)* by simply blocking the first two stops of c_1 being explored, and making one of them the transfer to c_2. In other words, within finite time, the adversary can create configuration *Flip(0)*; it can then transform configuration *Flip(i)* into *Flip(i+1)*, until configuration *Flip*$(\alpha-q-1)$ is reached. At this point the adversary reveals the entire graph as follows: it unblocks $r_{\alpha-q+1}$, the transfer site leading to carrier $c_{\alpha-q+1}$; it assigns carrier c_j to be connected to c_{j+1} for $\alpha > j > \alpha - q$; finally it chooses $k-1$ non-transfer stops of these carriers to be black holes; notice that they can be chosen because, since $q = \lceil \frac{k-2}{\beta-2} \rceil$, the number of non-transfer stops among these carriers is $q(l-2)+1 \geq k-1$.

The transformation from *Flip(i)* into *Flip(i+1)* costs the solution protocol $\mathcal{P}$ at least $\Omega(k \cdot i \cdot \beta)$ carrier moves, and this is done for $1 \leq i \leq \alpha - q$; since $\alpha(l-2) \geq (k-2)$ it follows that $\alpha - q = \alpha - \lceil \frac{k-2}{\beta-2} \rceil \geq \alpha - \frac{k-2}{\beta-2} \geq \frac{\alpha}{2}$; hence, $\sum_{1 \leq i \leq \alpha-q} i = O(\alpha^2)$. In other words, the adversary can force any solution protocol to use $\Omega(\alpha^2 \cdot \beta \cdot \gamma)$ carrier moves.

Theorem 4 *For any α, β, γ, where $\alpha, \beta > 2$ and $1 < \gamma < \beta - 1$, there exists a simple subway graph $\boldsymbol{G}$ with α carriers with maximum route length β and faulty* *load γ in which every subway mapping protocol $\mathcal{P}$ requires $\Omega(\alpha \cdot \beta^2)$ carrier moves in the worst case. This result holds even if the topology of $\boldsymbol{G}$ is known to the agents,*

Theorem 5 *Protocol* SubwayExplore *is agent-optimal and move-optimal.*

References

1. B. Bui-Xuan, A. Ferreira, and A. Jarry. Computing shortest, fastest, and foremost journeys in dynamic networks. *Int. J. Found. Comp. Sci.*, 14(2):267–285, 2003.
2. J. Chalopin, S. Das, and N. Santoro. Rendezvous of mobile agents in unknown graphs with faulty links. In *DISC 2007*, pages 108–122. Springer, 2007.
3. C. Cooper, R. Klasing, and T. Radzik. Searching for black-hole faults in a network using multiple agents. In *OPODIS 2006*, pages 320–332. Springer, 2006.
4. C. Cooper, R. Klasing, and T. Radzik. Locating and repairing faults in a network with mobile agents. *Theoretical Computer Science*, to appear, 2010.
5. J. Czyzowicz, D. Kowalski, E. Markou, and A. Pelc. Complexity of searching for a black hole. *Fundamenta Informaticae*, 71(2,3):229–242, 2006.
6. J. Czyzowicz, D. Kowalski, E. Markou, and A. Pelc. Searching for a black hole in synchronous tree networks. *Combin. Probab. Comput.*, 16(4):595–619, July 2007.
7. S. Dobrev, P. Flocchini, G. Prencipe, and N. Santoro. Searching for a black hole in arbitrary networks. *Distibuted Computing*, 19(1):1–19, 2006.
8. S. Dobrev, P. Flocchini, G. Prencipe, and N. Santoro. Mobile search for a black hole in an anonymous ring. *Algorithmica*, 48(1):67–90, 2007.
9. P. Flocchini, D. Ilcinkas, and N. Santoro. Ping pong in dangerous graphs: Optimal black hole search with pure tokens. In *DISC 2008*, 227–241. Springer, 2008.
10. P. Flocchini, M. Kellett, P. Mason, and N. Santoro. Map construction and exploration by mobile agents scattered in a dangerous network. In *IPDPS 2009*, 1-10, 2009.
11. P. Flocchini, B. Mans, and N. Santoro. Exploration of periodically varying graphs. In *ISAAC 2009*, volume 5878 of *LNSC*, pages 534–543. Springer, 2009.
12. P. Glaus. Locating a black hole without the knowledge of incoming links. In *ALGOSENSORS 2009*, volume 5804 of *LNCS*, pages 128–138. Springer, 2009.
13. R. Klasing, E. Markou, T. Radzik, and F. Sarracco. Hardness and approximation results for black hole search in arbitrary networks. *Theoretical Computer Science*, 384(2–3):201–221, October 2007.
14. R. Klasing, E. Markou, T. Radzik, and F. Sarracco. Approximation bounds for black hole search problems. *Networks*, 54(4):216–226, 2008.
15. A. Kosowski, A. Navarra, and M. C. Pinotti. Synchronization helps robots to detect black holes in directed graphs. In *OPODIS 2009*, 86–98. Springer, 2009.
16. C. Liu and J. Wu. Scalable routing in cyclic mobile networks. *IEEE Trans. Parallel Distrib. Syst.*, 20(9):1325–1338, 2009.
17. R. O'Dell and R. Wattenhofer. Information dissemination in highly dynamic graphs. In *2005 Joint Work. on Foundations of Mobile Computing*, 104–110, 2005.
18. X. Zhang, J. Kurose, B. Levine, D. Towsley, and H. Zhang. Study of a bus-based disruption-tolerant network. In *13th Int. Conf. on Mobile Computing and Networking*, 206, 2007.
19. Z. Zhang. Routing in intermittently connected mobile ad hoc networks and delay tolerant networks: Overview and challenges. *IEEE Communications Surveys & Tutorials*, 8(1):24–37, 2006.