Improved Bounds for Optimal Black Hole Search with a Network Map

Stefan Dobrev[1], Paola Flocchini[1], and Nicola Santoro[2]

[1] SITE, University of Ottawa
{sdobrev,flocchin}@site.uottawa.ca
[2] School of Computer Science, Carleton University
santoro@scs.carleton.ca

Abstract. A *black hole* is a harmful host that destroys incoming agents without leaving any observable trace of such a destruction. The *black hole search* problem is to unambiguously determine the location of the black hole. A team of agents, provided with a network map and executing the same protocol, solves the problem if at least one agent survives and, within finite time, knows the location of the black hole.
It is known that a team must have at least *two* agents. Interestingly, two agents with a map of the network can locate the black hole with $O(n)$ moves in many highly regular networks; however the protocols used apply only to a narrow class of networks. On the other hand, any universal solution protocol must use $\Omega(n \log n)$ moves in the worst case, regardless of the size of the team.
A universal solution protocol has been recently presented that uses a team of just *two* agents with a map of the network, and locates a black hole in at most $O(n \log n)$ moves. Thus, this protocol has both *optimal size* and *worst-case-optimal cost*. We show that this result, far from closing the research quest, can be significantly improved.
In this paper we present a *universal* protocol that allows a team of *two* agents with a network map to locate the black hole using at most $O(n + d \log d)$ moves, where d is the diameter of the network. This means that, without losing its universality and without violating the worst-case $\Omega(n \log n)$ lower bound, this algorithm allows two agents to locate a black hole with $\Theta(n)$ cost in a very large class of (possibly unstructured) networks.

Keywords: Distributed Algorithms, Distributed Computing, Mobile Agents, Harmful Host, Undetectable Failure, Size and Cost Optimal Protocols

1 Introduction

In networked systems that support autonomous *mobile agents*, a main concern is how to develop efficient agent-based *system protocols*; that is, to design protocols that will allow a team of identical simple agents to cooperatively perform (possibly complex) system tasks. Example of basic tasks are *wakeup*, *traversal*, *rendez-vous*, *election*. The coordination of the agents necessary to perform these

R. Královič and O. Sýkora (Eds.): SIROCCO 2004, LNCS 3104, pp. 111–122, 2004.
© Springer-Verlag Berlin Heidelberg 2004

tasks is not necessarily simple or easy to achieve. In fact, the computational problems related to these operations are definitely non trivial, and a great deal of theoretical research is devoted to the study of conditions for the solvability of these problems and to the discovery of efficient algorithmic solutions; e.g., see [1–8, 13].

At an abstract level, these environments can be described as a collection $\mathcal{E}$ of autonomous mobile *agents* (or *robots*) located in a graph G. The agents have computing capabilities and bounded storage, execute the same protocol, and can move from node to neighboring node. They are *asynchronous*, in the sense that every action they perform (computing, moving, etc.) takes a finite but otherwise unpredictable amount of time. Each node of the network, also called *host*, provides a storage area called *whiteboard* for incoming agents to communicate and compute, and its access is held in fair mutual exclusion. The research concern is on determining what tasks can be performed by such entities, under what conditions, and at what cost.

At a practical level, in these environments, *security* is the most pressing concern, and possibly the most difficult to address. Actually, even the most basic security issues, in spite of their practical urgency and of the amount of effort, must still be effectively addressed (for a survey, see [14]).

Among the severe security threats faced in these environments, a particularly troublesome one is a *harmful host*; that is, the presence at a network site of harmful stationary processes. This threat is acute not only in unregulated non-cooperative settings, such as Internet, but also in environments with regulated access and where agents cooperate towards common goals. In fact, a local (hardware or software) failure might render a host harmful. The problem posed by the presence of a harmful host has been intensively studied from a software engineering point of view (e.g., see [12, 15, 16]), and recently also from an algorithmic prospective [9–11].

Obviously, the first step in any solution to such a problem must be to *identify*, if possible, the harmful host; i.e., to determine and report its location. Following this phase, a "rescue" activity would conceivably be initiated to deal with the destructive process resident there. Depending on the nature of the danger, the task to identify the harmful host might be difficult, if not impossible, to perform.

A particularly harmful host is a *black hole*: a host that *disposes* of visiting agents upon their arrival, leaving *no observable trace* of such a destruction. The task is to unambiguously determine and report the location of the black hole, and will be called *black hole search*. The searching agents start from the same safe site and follow the same set of rules; the task is successfully completed if, within finite time, at least one agent survives and knows the location of the black hole. Note that this type of highly harmful host is not rare; for example, in asynchronous networks, the undetectable crash failure of a site will transform that site into a black hole.

Black hole search is a non trivial problem, its difficulty aggravated by the combination of absence of any trace of destruction (outside the black hole) and asynchrony of the agents. It has been investigated focusing on identifying conditions for its solvability and determining the smallest number of agents needed

for its solution [9–11]. Some conditions are very simple; for example, the graph G must be biconnected, the team must consist of at least *two* agents, and n must be known. A complete characterization has been provided for *ring* networks [10]. Our interest is in *universal* (or *generic*) solution protocols, i.e. protocols that can be used in every biconnected network.

The efficiency of a solution protocol is measured first and foremost by the *number of agents* used by the solution. This value, called *size*, depends on many factors, including the topology of the network, the amount of *a priori* information the agents have about the network, etc. In particular, in an arbitrary network, if the topology of the network is known, *two agents* suffice ! [11]. Indeed, this surprising result can be achieved by several protocols. The second efficiency measure is the *number of moves*, called *cost*, performed by the agents. Clearly the research interest is in the design of size-optimal universal solutions (i.e., using a team of just two agents) that are also cost-efficient.

Sometimes the network has special properties that can be exploited to obtain a lower cost network-specific protocol. For example, two agents can locate a black hole with only $O(n)$ moves in a variety of highly structured interconnection networks such as *hypercubes*, square *tori* and *meshes*, *wrapped butterflies*, *star graphs* [9]. On the other hand, there are networks where $\Omega(n \log n)$ moves are required *regardless of the number of agents*; such is for example the case of ring networks [10]. The lower bound for rings implies an $\Omega(n \log n)$ lower bound on the worst case cost complexity of any *universal* protocol.

Indeed, a universal solution protocol has been recently presented that has both *optimal size* and *worst-case-optimal cost*. In fact, it uses a team of just *two* agents with a map of the network, and locates a black hole in at most $O(n \log n)$ moves [11]. Surprisingly, this result does not close the research quest.

In this paper, we show that it is possible to considerably improve the bound on cost without increasing the team size. In fact, we present a *universal* protocol, *Explore and Bypass*, that allows a team of *two* agents with a map of the network to locate a black hole with cost $O(n + d \log d)$, where d denotes the diameter of the network.

This means that, without losing its universality and without violating the worst-case $\Omega(n \log n)$ lower bound, this algorithm allows two agents to locate a black hole with $\Theta(n)$ cost in a very large class of (possibly unstructured) networks: those where $d = O(n/\log n)$.

Importantly, there are many networks with $O(n/logn)$ diameter in which the existing protocols [9, 11] fail to achieve the $O(n)$ bound. A simple example of such a network is the *wheel*, a ring with a central node connected to all ring nodes, where the central node is very slow: those protocols will require $O(n \log n)$ moves.

2 Definitions and Terminology

2.1 Framework

Let $G = (V, E)$ be a simple biconnected graph; let $n = |V|$ be the size of G and d be its diameter. At each node x, there is a distinct label (called port number)

associated to each of its incident links (or ports); let $\lambda_x(x, z)$ denote the label associated at x to the link $(x, z) \in E$, and λ_x denote the overall injective mapping at x. The set $\lambda = \{\lambda_x | x \in V\}$ of those mappings is called a *labelling* and we shall denote by (G, λ) the resulting edge-labelled graph. The nodes of G can be *anonymous* (i.e., without unique names).

Operating in (G, λ) is a set $\mathcal{A}$ of distinct autonomous mobile agents. The agents can move from node to neighbouring node in G, have computing capabilities and bounded computational storage, obey the same set of behavioral rules (the *protocol*). The agents are *asynchronous* in the sense that every action they perform (computing, moving, etc) takes a finite but otherwise unpredictable amount of time. Initially, the agents are in the same node h, called *home base*. Each agent has a map of the labelled graph (G, λ) where the location of the home base is indicated.

Each node has a bounded amount of storage, called *whiteboard*; $O(\log n)$ bits suffice for all our algorithms. Agents communicate by reading from and writing on the whiteboards; access to a whiteboard is gained fairly in mutual exclusion.

2.2 Black Hole Search

A *black hole* is a node where resides a stationary process that destroys any agent arriving at that node; no observable trace of such a destruction will be evident to the other agents. The location of the black hole is unknown to the agents. The *Black Hole Search* problem is to find the location of the black hole. More precisely, the problem is solved if at least one agent survives, and all surviving agents know the location of the black hole.

The main measure of complexity of a solution protocol $\mathcal{P}$ is the number of agents used to locate the black hole, called the *size* of $\mathcal{P}$. Clearly, at least two agents are needed to locate the black hole. On the other hand, two agents with a map of the network suffice to locate the black hole.

The other measure of complexity of a protocol $\mathcal{P}$ is the total number of moves performed by the agents, called the *cost* of $\mathcal{P}$.

2.3 Exploration

Let T be a spanning-tree of G rooted in the home base of the agents. For each node v we define T_v to be the subtree of T rooted at v. We will slightly abuse the notation and use T_v to mean both the subtree rooted at v and the set of the nodes of this subtree. For a given T_v we define the *components* of T_v to be the connected components of the graph induced in G by the nodes of $T_v \setminus \{v\}$. Clearly, each component of T_v is a union of one or more of its subtrees (plus the connecting edges). As usual, $|S|$ means the size of the set S (we will almost exclusively talk about sets of nodes, sometimes about set of edges). For a given node v the *level* of v is defined as its distance (in T) from the root. Given two nodes v and w, $\pi_{v,w}$ denotes the unique directed path from v to w in T. For a given path π we denote by π^{-1} the reversal of this path. Given a node v and a set $S = \{e_1, e_2, \ldots, e_k\}$ of edges of T, we denote by $C^v \setminus S$ the connected component containing v obtained from T by removing the edges in S.

We say that a node is *unexplored* if it has not been visited by an agent. A node v is *explored* if all the nodes of T_v have been visited. A node v is *safe* if it has been visited, but there are *unexplored*nodes in T_v.

Similarly, each *port* can be classified as: (1) *unexplored* – no agent has yet arrived/departed via that port; (2) *dangerous* – an agent has left via this port, but none has arrived from there; (3) *used* – an agent arrived via that port and (4) *explored* – the port is *used* and all nodes in the corresponding subtree has been explored. Obviously, a *used* port does not lead to black hole; on the other hand, both *unexplored* and *dangerous* ports might lead to it.

To ensure that at least one agent survives, we will not allow an agent to leave through a *dangerous* port. To prevent the execution from stalling, we will require any *dangerous* port not leading to the black hole to be made *used* as soon as possible. This is accomplished using the following technique called *Cautious Walk* [9–11]:

Cautious Walk
Whenever an agent a leaves a node u through an *unexplored* port p (transforming it into *dangerous*) leading to a node v, upon its arrival and before proceeding somewhere else, a returns to u (transforming that port into *used*). Node u will be called the *last safe* node of a until a is back in u to make the port *used.*

If agent b reaches the last safe node u of a (i.e., a has not returned yet from v), b will perceive a as being *blocked on* (u, v) (or simply, *blocked at* v); this perception will persist until b becomes aware that a has indeed returned from v.

3 Algorithm *Explore and Bypass*

The proposed algorithm *Explore and Bypass* ($\mathcal{E}\&\mathcal{B}$) is a rather complex protocol that uses several quite different techniques and strategies; it allows a team of just *two* agents with a map of the network to solve the Black Hole Location using at most $O(n + d \log d)$ moves.

In the following we will first describe the overall strategy. The structure of the main modules and the algorithmic techniques they employ are described next. The full description can be found in the appendix, as the handling of numerous cases does not fit in the limited space available. The overall structure and the major modules of Algorithm $\mathcal{E}\&\mathcal{B}$ are captured in Fig. 1.

In assessing the overall *cost*, our goal will be to charge the moves caused by each module to the nodes explored during the execution of that module. As most of the activities consist of traversals of unexplored nodes (and possibly a second traversal of newly explored nodes), this works quite well. The activities that cannot be easily accounted for in this way are counted as *overhead* and the cost is charged to a higher level module.

3.1 Overall Strategy

In Algorithm $\mathcal{E}\&\mathcal{B}$, the two agents, a and b, cooperatively explore the network to locate the black hole. The exploration is achieved by the two agents performing

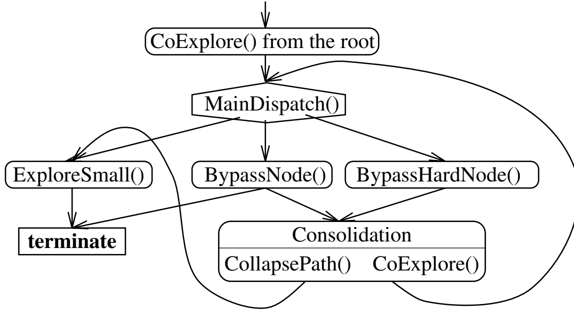

Fig. 1. The overall structure of the Algorithm $\mathcal{E}\&\mathcal{B}$.

mainly a *cooperative depth-first* traversal of a spanning-tree T of the network, using the other links as bypasses if needed. Special traversal techniques are employed in some particular cases to avoid an unnecessary increase in the overall cost of the protocol. The tree T used is a *breadth-first* spanning-tree rooted in the home base of the agents; this choice is made for efficiency reasons that will become apparent later.

The algorithm starts with both agents cooperatively exploring the tree T (module CoExplore) without passing through non-T edges. Eventually, no more nodes can be explored without using non-T edges: an agent a arrives at a node u and discovers that the only non-explored edge is a *dangerous* edge (u, v) (thus, b is blocked at v). When this occurs, there are several possible actions that a may take.

The decision on which action a will take (procedure MainDispatch()) depends on the structure of the set U of still unexplored nodes: (1) If U is small ($|U| \leq d$), a will execute procedure ExploreSmall(). (2) If the largest unexplored subtree T_m hanging from v is not too big ($|U \setminus T_m| \geq d$), a will execute procedure BypassNode(); otherwise (3) a will execute procedure BypassHardNode(). Agent a informs b about the decided action by leaving a message in the whiteboard at u.

If agent b returns to u, it will discover a's decision, and coordinate its activity with a. The goal is to re-establish a situation similar to the initial one, where both agents cooperatively explore a single (sub)tree (and eventually one of them perceives the other as blocked on a link). Although the goal might not be achieved, the agents will always return to a configuration when MainDispatch() can be applied again:

Property UVW: An agent (say, b) is blocked on (u, v), and the other agent (say, a) is in u. Either the only *unexplored* nodes are the nodes of T_v, or there exists a neighbour w of v such that the *unexplored* nodes are the nodes of $T_v \setminus T_w$ (if v is a child of u and w is a child of v) or $T_v \setminus T_u$ (if u is a child of v and v is a child of w)).

Summarizing, the two agents perform a cooperative depth-first traversal of a breadth-first spanning-tree of the network, until an agent finds the other blocked

on a link and all unexplored nodes are on the other side of that link. In this case, the agent will either execute ExploreSmall() (a procedure that leads the protocol to terminate without any other call to any other procedure), or will bypass the blocked agent using either BypassNode() or BypassHardNode(); every time the latter cases occur, the level (i.e., the distance from the root) in T of the node being bypassed will become larger, so eventually the ExploreSmall() action will be chosen.

3.2 Cooperative Exploration of a Tree

Procedure CoExplore() is the principal exploration procedure. The algorithm starts with both agents executing CoExplore(x) from the home base x. It is a *cooperative depth-first* traversal in which each agent avoids the part being explored by the other (for as long as possible).

When executing CoExplore(Node u), the agent, say a, determines if there is an unexplored subtree T_v of u. If so, it (recursively) explores T_v using cautious walk. That is, it will: mark the port from u to v as *dangerous*; go to v; (if it survives) mark the port from v to u as *safe*, return to u, mark the port from u to v as *safe*, and check for messages – if any, execute the corresponding code, otherwise return to v, recursively execute CoExplore(v), and finally mark the port from u to v as *explored*. This process is repeated as long as there is an unexplored subtree of u.

There are now two cases. If all incident edges are *explored*, the tree T_u does not contain the black hole; in this case, the recursive call ends. Otherwise, there is only one non-explored (i.e., *safe* or *dangerous*) link left, the one leading to the subtree where the other agent is still working. Let (u, z) be such a link. If (u, z) is *safe*, a goes to z to join the other agent in exploring T_z, executing CoExplore(z). If (u, z) is *dangerous*, the co-exploration cannot continue, and the agent executes MainDispatch(u, z, nil) to decide which action to take.

The cost of CoExplore() is linear in the number of nodes explored, as it is essentially the traversal of a tree.

3.3 Exploring a Small Forest

When an agent finds the other blocked at a node, the agent executes Procedure ExploreSmall() if there are at most d *unexplored* nodes left; i.e. $|U| \leq d$. The main idea is that the unexplored part is so small that it can be efficiently handled by executing the existing $O(n \log n)$ algorithm of [11] on U. However, that algorithm requires that the graph is biconnected, while U might not be. The main task of ExploreSmall() is therefore to explore U in such a way that, eventually, the part U' of U still left unexplored satisfies the following property:

Property Small: There is a path π of length $O(d)$ in $G \setminus U'$ such that adding π to U' results in a biconnected graph U''.

The description of how to obtain U' from U using $O(n)$ moves is rather technical and can be found in the appendix. When only U' is left unexplored, the path π is identified and the algorithm from [11] is applied to U''.

Since $|U| \leq d$, $U' \subseteq U$ and $|\pi| \in O(d)$, also $|U''| \in O(d)$ and the complexity of applying this algorithm to U'' is $O(d \log d)$. In other words, ExploreSmall() requires at most $O(n + d \log d)$ moves in total by the two agents.

3.4 Exploring a Forest of Small Components

If agent a finds agent b blocked at a node v, a must be able to bypass v to continue the exploration. BypassNode() is the branch taken in MainDispatch() if the largest connected component of $U \setminus \{v\}$ is not too big (i.e. there are at least d other nodes in U).

As a result of BypassNode(), either v is identified as the black hole, or all unexplored nodes will be located in a single subtree hanging from v; in the latter case, either PropertyUVW holds or ExploreSmall() is called. We will now describe its modules in some details.

Main Action – Procedures BypassNode() and ReleasedB(). When agent a finds agent b blocked on the link (u, v), a must be able to bypass v to continue the exploration. BypassNode() is the branch taken in MainDispatch() if the largest connected component of $U \setminus \{v\}$ is not too big (i.e. there are at least d other nodes in U); if/when b returns to u, it executes procedure ReleasedB().

The main idea comes from observing that 1) each unexplored component can be reached from u by a safe bypass path β (otherwise v would disconnect G) and 2) $|\beta| \in O(d)$ (from the structure of the explored and unexplored parts).

The trees (i.e., the connected components) in the forest $U \setminus \{v\}$ must be reached by a to be explored. To control the amount of moves to reach each of those trees, the exploration of the trees will be done in order and with a different technique, depending on a the size of the tree. More precisely:

1. Agent a first explores the largest components (i.e., the ones containing a subtree of size at least d); the cost of reaching any such tree is $O(d)$, hence it can be "charged" to the cost of exploring that tree (increasing the multiplicative constant but not the order of magnitude of that cost).
2. After all these components have been explored (and assuming agent b is still blocked) a proceeds to explore the smaller ones according to the following strategy:
   a traverses the whole *explored* part in one sweep and whenever it finds a link leading to an unexplored component C, a explores C, then returns and continues with the traversal. If b is still blocked, the only unexplored node left is v that is identified as the black hole.

What makes the situation complex is the fact that b could become unblocked at any time; the algorithm must be able to handle correctly and efficiently every configuration this fact might cause. This goal is achieved by carefully exploring the components using a special technique (procedure ExploreComponent) described in the next section.

The activity of agent b, when (if) it becomes unblocked and finds the message <BypassNode>, is prescribed by procedure ReleasedB(): when (if) it is released,

b follows the trace of a, until it reaches the last safe node visited by a; b then leaves there a message, telling a to finish its component, and goes on exploring the remaining unexplored subtrees of v. If a finishes its part, we are exactly in the situation where the agents continue the co-exploration of T_v as if no bypassing had happened. However, if b finishes first while a is still working on its component, the structure of the remaining unexplored part can be quite complex. In this case, the unexplored part must be consolidated so that PropertyUVW holds and MainDispatch() can be applied. This is accomplished by procedure CollapsePath() described later.

ExploreComponent. To simplify the consolidation (without increasing the complexity), the components are explored in a special manner: Consider the *component graph* H where nodes correspond to the subtrees of component C, and edges correspond to connections (in G) between subtrees. The component graph H is "visited" using a depth-first traversal, where "visiting a node" in H means fully exploring the corresponding subtree before going to the next one.

Such an approach guarantees the following property to hold:

Property P1: At any moment there is only one partially explored subtree $T_{v'}$ of component C.

This property will allow b, when unblocked, to tell a to finish only the current subtree $T_{v'}$, not the whole component C.

Consider now the partially explored subtree $T_{v'}$. Let y be the node in which a entered $T_{v'}$ and let $\pi_{yv'}$ be the path from y to the root v'. After agent a enters $T_{v'}$, it performs a DF-traversal of $T_{v'}$ starting from y and satisfying the following restriction: If an edge (h, k) belongs to $\pi_{yv'}$, it will be the last edge explored from h. Such an approach guarantees that the unexplored part of $T_{v'}$ has the following structure:

Property P2: Let w be the last explored node on the path from y to the root v', and let w' be the next node (if $w \neq v'$) on this path. The path from w' to v', together with all subtrees hanging from this path, is unexplored. If a is on the link from w to w', then all subtrees hanging from w are explored; otherwise, a subtree of w contains a and is partially explored, while the other subtrees are either fully explored, or unexplored.

Notice that, if b has done all its work and a is still in $T_{v'}$, property P2 tells us that the unexplored part consists just of a path and the trees hanging from this path. In order to reach a configuration where PropertyUVW is finally satisfied (and, thus MainDispatch() can be called again), this path is explored using the procedure CollapsePath() described in the next section.

As each subtree is traversed at most twice, the complexity of ExploreComponent() is linear in the number of explored nodes.

CollapsePath. Procedure CollapsePath() is called when all subtrees of v except one have been explored. The remaining unexplored nodes form a path π_e with

unexplored subtrees hanging from this path. The agents divide the remaining unexplored path π_e into two disjoint parts with about equal number of nodes (the nodes of the hanging unexplored subtrees are counted as well), and each agent goes on exploring its part. Note that there is a safe bypass path β of length $O(d)$ connecting the opposite ends of π_e : if π_e is in the first subtree T' of component C, we use as bypass β the path that agent a had used to reach C; otherwise, β passes through the (already explored) subtree T'' from which the entry point of T' was reached (note that T'' is connected to the already explored node v).

The agent that finishes first uses β to reach the other one and split the workload again. This is repeated until there is a single unexplored node in the path or the number of unexplored nodes becomes less then d. In the first case, PropertyUVW is satisfied, in the second case ExploreSmall() is directly executed.

It may happen (because the agents are also exploring the subtrees hanging from the path) that the whole path is explored. In such a case, both agents end up co-exploring a single subtree hanging from this path and eventually one of them will become blocked, creating a configuration corresponding to the first case of PropertyUVW.

Each round of halving the unexplored path involves traversing a bypass of length $O(d)$. However, as the number of unexplored nodes is about halved at each round, and CollapsePath() terminates when there are less then d nodes left, the number of rounds with less than d newly explored nodes is $O(1)$. In other words, the overhead of of CollapsePath() is only $O(d)$.

3.5 Exploring a Forest with a Large Component

When agent a must bypass agent b blocked at a node v, if $U \setminus \{v\}$ has one overwhelmingly large subtree T_m, then BypassNode() may be inefficient: It can happen that an overhead of $O(d)$ is incurred, but the one remaining partially explored subtree of v is T_m itself but only $o(d)$ new nodes have been explored. Since in the worst case, this can happen $O(d)$ times, the total overhead would be an unacceptable $O(d^2)$.

To avoid such an overhead, we must make sure there is enough progress; in this way, while the overhead is not avoided, it has no chance to accumulate. This is done by procedure BypassHardNode().

Let x be the node dividing $T|_U = U \cap T$ (the tree induced in T by the nodes of U) into components of size at most $|U|/2$. BypassHardNode() uses the path $\pi_{v,x}$ from v to x as the pivot, i.e. after BypassHardNode() there is either a single unexplored subtree of $T|_U$ hanging from the fully explored $\pi_{v,x}$, or there is one unexplored node $w \in \pi_{v,x}$ and the subtrees hanging from it are unexplored.

In this way, we are ensured that the number of unexplored nodes is at least halved and MainDispatch() can be applied again. Moreover, the number of newly explored nodes is at least $d/2$, i.e. there are enough of them to distribute the $O(d)$ overhead among them.

There is a major problem with this approach: while all components of $U \setminus \{v\}$ are reachable from the already explored nodes, the same is not true for the

components of $U \setminus \pi_{v,x}$. We must make sure that at least some components (or the path itself) are reachable.

To achieve that, the idea of BypassHardNode() is to proceed along the path from v to x, taking as long a jump ahead as possible: in other words, a goes to the furthest reachable subtree $T_{w'}$ hanging from $\pi_{v,x}$ (w is the parent if w') and explores from there (first $T_{w'}$, then the path from w to x and the hanging subtrees), while b (after/if becoming unblocked) will take care of the path (and hanging subtrees) between v and w. If a finishes first, the remaining unexplored path between v and w will be collapsed and the number of unexplored nodes is at least halved. If b finishes first, there are several possible cases but two principal outcomes:

(1) The unexplored nodes are limited to the subtree $T_{w'}$. The corresponding unexplored path is collapsed (procedure CollapsPath() is called).

(2) We are essentially in the same situation as we started, but we have moved along the line from v to x to at least w. The process (bypassing) is repeated until x is reached and/or the unexplored nodes are limited to a subtree (and possibly its root) hanging from the path (thus, Collapsepath() can be called to terminate).

In case 2) a new bypass path will be needed to reach a smaller area of unexplored nodes. The crucial observation is that the bypass does not go below w, otherwise T'_w would not have been the furthest reachable subtree. This means that the bypass of iteration $i > 1$ uses only nodes explored in the previous iteration.

As each bypass path (except in the last iteration, when it is used in CollapsePath()) is used only a constant number of times, the overhead of iteration $i > 1$ can be charged to iteration $i - 1$. We are left with the overhead $O(d)$ for the first iteration and for the possible CollapsePath() in the last iteration. Since BypassHardNode() has explored at least $d/2$ nodes, this overhead can be distributed among them.

4 Analysis

Due to space constraints, we omit correctness and complexity analysis of Algorithm $\mathcal{E}\&\mathcal{B}$. Full proofs can be found in the technical report.

References

1. S. Alpern. The Rendezvous search problem. SIAM J. of Control and Optimization 33, 673 - 683, 1995.
2. E. Arkin, M. Bender, S. Fekete, and J. Mitchell. The freeze-tag problem: how to wake up a swarm of robots. In 13th ACM-SIAM Symposium on Discrete Algorithms (SODA '02), pages 568–577, 2002.
3. B. Awerbuch, M. Betke, and M. Singh. Piecemeal graph learning by a mobile robot. Information and Computation 152, 155–172, 1999.
4. L. Barrière, P. Flocchini, P. Fraigniaud, and N. Santoro. Capture of an intruder by mobile agents. In 14th ACM-SIAM Symp. on Parallel Algorithms and Architectures (SPAA '02), 200-209, 2002.

5. L. Barrière, P. Flocchini, P. Fraigniaud, and N. Santoro. Election and rendezvous in fully anonymous systems with sense of direction. In 10th Colloquium on Structural Information and Communication complexity (SIROCCO '03), 17-32, 2003.
6. X. Deng and C. H. Papadimitriou. Exploring an unknown graph. J. of Graph Theory, 32:265-297, 1999.
7. A. Dessmark, P. Fraigniaud and A. Pelc. Deterministic rendezvous in graphs. In 11th European Symposium on Algorithms (ESA'03) 2003.
8. K. Diks, P. Fraigniaud, E. Kranakis, and A. Pelc. Tree exploration with little memory. In 13th ACM-SIAM Symposium on Discrete Algorithms (SODA '02), 588–597, 2002.
9. S. Dobrev, P. Flocchini, R. Král'ovič, G. Prencipe, P. Ružička, and N. Santoro. Optimal search for a black hole in common interconnection networks. In Proc. of Symposium on Principles of Distributed Systems (OPODIS'02), 2002.
10. S. Dobrev, P. Flocchini, G. Prencipe, and N. Santoro. Mobile agents searching for a black hole in an anonymous ring. In Proc. of 15th Int. Symp. on Distributed Computing (DISC'01), 166-179, 2001.
11. S. Dobrev, P. Flocchini, G. Prencipe, and N. Santoro. Searching for a black hole in arbitrary networks: Optimal mobile agent protocols. In Proc. of 21st ACM Symposium on Principles of Distributed Computing (PODC'02), 153-162, 2002.
12. F. Hohl. A Model of attacks of malicious hosts against mobile agents. In ECOOP Workshop on Distributed Object Security and 4th Workshop on Mobile Object Systems, LNCS 1603, 105-120, 1998.
13. E. Kranakis, D. Krizanc, N. Santoro, and C. Sawchuk. Mobile agent rendezvous in a ring. In 23rd International Conference on Distributed Computing Systems (ICDCS'03), 2003.
14. R. Oppliger. Security issues related to mobile code and agent-based systems. Computer Communications 22, 12, 1165-1170, 1999.
15. T. Sander and C. F. Tschudin. Protecting mobile agents against malicious hosts. In Conf on Mobile Agent Security, LNCS 1419, 44–60, 1998.
16. J. Vitek and G. Castagna. Mobile computations and hostile hosts. In D. Tsichritzis, editor, Mobile Objects, 241-261. University of Geneva, 1999.