Searching for a Black Hole in Tree Networks

Jurek Czyzowicz[1,⋆], Dariusz Kowalski[2,3,⋆⋆],
Euripides Markou[1,⋆⋆⋆], and Andrzej Pelc[1,†]

[1] Département d'informatique, Université du Québec en Outaouais,
Hull, Québec J8X 3X7, Canada
{jurek, evripidi, pelc}@uqo.ca
[2] Max-Planck-Institut für Informatik,
Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany
darek@mpi-sb.mpg.de
[3] Instytut Informatyki, Uniwersytet Warszawski,
Banacha 2, 02-097 Warszawa, Poland

Abstract. A black hole is a highly harmful stationary process residing in a node of a network and destroying all mobile agents visiting the node, without leaving any trace. We consider the task of locating a black hole in a (partially) synchronous tree network, assuming an upper bound on the time of any edge traversal by an agent. The minimum number of agents capable to identify a black hole is two. For a given tree and given starting node we are interested in the fastest possible black hole search by two agents. For arbitrary trees we give a 5/3-approximation algorithm for this problem. We give optimal black hole search algorithms for two "extreme" classes of trees: the class of lines and the class of trees in which any internal node (including the root which is the starting node) has at least 2 children.

Keywords: algorithm, black hole, mobile agent, tree.

1 Introduction

1.1 The Background and the Problem

Security of mobile agents working in a network environment is an important issue which receives recently growing attention. Protecting agents from "host attacks", i.e., harmful items stored in nodes of the network, has become almost as urgent as protecting a host, i.e., a node of the network, from an agent's attack [8, 9]. Various methods of protecting mobile agents against malicious hosts have been discussed, e.g., in [5, 6, 7, 8, 9, 10].

⋆ Research supported in part by NSERC grant.
⋆⋆ Research supported in part by grants from KBN (4T11C04425) and UE DELIS.
⋆⋆⋆ This work was done during this author's stay at the Research Chair in Distributed Computing of the Université du Québec en Outaouais, as a postdoctoral fellow.
† Research supported in part by NSERC grant and by the Research Chair in Distributed Computing of the Université du Québec en Outaouais.

T. Higashino (Ed.): OPODIS 2004, LNCS 3544, pp. 67–80, 2005.

In this paper we consider hostile hosts of a particularly harmful nature, called *black holes* [1, 2, 3, 4]. A black hole is a stationary process residing in a node of a network and destroying all mobile agents visiting the node, without leaving any trace. Since agents cannot prevent being annihilated once they visit a black hole, the only way of protection against such processes is identifying the hostile node and avoiding further visiting it. Hence we are dealing with the issue of locating a black hole: assuming that there is at most one black hole in the network, at least one surviving agent must find the location of the black hole if it exists, or answer that there is no black hole, otherwise. The only way to locate the black hole is to visit it by at least one agent, hence, as observed in [2], at least two agents are necessary for one of them to locate the black hole and survive. Throughout the paper we assume that the number of agents is minimum possible for our task, i.e., 2, and that they start from the same node, known to be safe.

In [1, 2, 3, 4] the issue of efficient black hole search was extensively studied in many types of networks. The underlying assumption in these papers was that the network is totally asynchronous, i.e., while every edge traversal by a mobile agent takes finite time, there is no upper bound on this time. In this setting it was observed that, in order to solve the problem, the network must be 2-connected, in particular black hole search is infeasible in trees. This is because, in asynchronous networks it is impossible to distinguish a black hole from a "slow" link incident to it. Hence the only way to locate a black hole is to visit all other nodes and learn that they are safe. (In particular, it is impossible to answer the question of whether a black hole actually exists in the network, hence [1, 2, 3, 4] worked under the assumption that there is exactly one black hole and the task was to locate it.)

Totally asynchronous networks rarely occur in practice. Often a (possibly large) upper bound on the time of traversing any edge by an agent can be established. Hence it is interesting to study black hole search in such partially synchronous networks. Without loss of generality, this upper bound on edge traversal time can be normalized to 1 which yields the following definition of the time of a black hole search scheme: this is the maximum time taken by the scheme, i.e. the time under the worst-case location of the black hole (or when it does not exist in the network), assuming that all edge traversals take time 1.

Our partially synchronous scenario makes a dramatic change to the problem of searching for a black hole. Now it is possible to use the time-out mechanism to locate the black hole in any graph, with only two agents, as follows: agents proceed along edges of a spanning tree. If they are at a safe node v, one agent goes to the adjacent node and returns, while the other agent waits at v. If after time 2 the first agent has not returned, the other one survives and knows the location of the black hole. Otherwise, the adjacent node is known to be safe and both agents can move to it. This is in fact a variant of the *cautious walk* described in [2] but combining it with the time-out mechanism makes black hole search feasible in any graph. Hence the issue is now not the feasibility but the time efficiency of black hole search, and the present paper is devoted to this problem.

Since for any network, black hole search can be done using only the edges of its spanning tree, solving the problem of fast black hole search on trees seems a natural first step. Hence in this paper we restrict attention to black hole search in tree networks using two agents, and our goal is to accomplish this task in minimum time. Clearly, in many graphs, there are more efficient black hole search schemes than those operating in a spanning tree of the graph, and the generalization of our problem to arbitrary networks remains an important and interesting open issue.

The time of a black hole search scheme should be distinguished from the time complexity of the algorithm producing such a scheme. While the first was defined above for a given input consisting of a network and a starting node, and is in fact the larger of the numbers of time units spent by the two agents, the second is the time of producing such a scheme by the algorithm. In other words, the time of the scheme is the time of walking and the time complexity of the algorithm is the time of thinking.

Constructing a fastest black hole search scheme for arbitrary trees turns out to be far from trivial. In particular, the following problem remains open. Does there exist a polynomial time algorithm which, given a tree and a starting node as input, produces a black hole search scheme working in shortest possible time for this input? Nevertheless, we show fastest schemes for some classes of trees and give a 5/3-approximation algorithm for the general case.

1.2 Our Results

For arbitrary trees we give a 5/3-approximation algorithm for the black hole search problem. More precisely, given a tree and a starting node as input, our algorithm produces a black hole search scheme whose time is at most 5/3 of the shortest possible time for this input.

We give optimal black hole search algorithms for two "extreme" classes of trees: the class of lines and the class of trees in which any internal node (including the root which is the starting node) has at least 2 children. More precisely, for every input in the respective classes these algorithms produce a black hole search scheme whose time is the shortest possible for this input.

All our algorithms work in time linear in the size of the input.

2 Model and Terminology

We consider a tree T rooted at node s which is the starting node of both agents, and is assumed to be safe (s is not a black hole). Notions of child, parent, descendant and ancestor, are meant with respect to this rooted tree. Agents have distinct labels. They can communicate only when they meet (and not, e.g., by leaving messages at nodes). We assume that there is at most one black hole in the network. This is a node which destroys any agents visiting it. A black hole search scheme (*BHS-scheme*) for the input (T, s) is a pair of sequences of edge traversals (moves) of each of the two agents, with the following properties.

- Each move takes one time unit.
- Upon completion of the scheme there is at least one surviving agent, i.e., an agent that has not visited the black hole, and this agent either knows the location of the black hole or knows that there is no black hole in the tree. The surviving agents must return to s.

The time of a black hole search scheme is the number of time units until the completion of the scheme, assuming the worst-case location of the black hole (or its absence, whichever is worse). It is easy to see that the worst case for a given scheme occurs when there is no black hole in the network or when the black hole is the last unvisited node, both cases yielding the same time. A scheme is called *fastest* for a given input if its time is the shortest possible for this input.

For any edge of a tree we define the following states:

- *unknown*, if no agent has moved yet along this edge (initial state of every edge),
- *explored*, if either the remaining agents know that there is no black hole incident to this edge, or they know which end of the edge is a black hole.

Note that in between meetings, an edge may be neither unknown nor explored. This is the case when an unknown edge has been just traversed by an agent.

Any BHS-scheme must have the following property: after a finite number of steps, at least one agent stays alive and all edges are explored (there is at most one black hole, so once the black hole has been found, all edges are explored).

The *explored territory* at step t of a BHS-scheme is the set of explored edges. At the beginning of a BHS-scheme the explored territory is empty. We say that a *meeting* occurs in node v at step t when the agents meet at node v and exchange information which *strictly increases* the explored territory. Node v is called a *meeting point*.

In any step of a BHS-scheme, an agent can traverse an edge or wait in a node. Also the two agents can meet. If at step t a meeting occurs, then the explored territory at step t is defined as the explored territory *after* the meeting. The sequence of steps of a BHS-scheme between two consecutive meetings is called a *phase*.

3 Preliminary Results

Lemma 1. *In a BHS-scheme, an unexplored edge cannot be traversed by both agents.*

Hence in a BHS-scheme, an edge can be explored only in the following way: an agent traverses this edge and then a meeting is scheduled. Whether it occurs or not (in the latter case the agent vanished in the black hole) the edge becomes explored.

Lemma 2. *During a phase of a BHS-scheme an agent can traverse at most* one *unexplored edge.*

Therefore an unknown edge could be explored in the next phase only if it is adjacent to the explored territory. The explored territory increases only at scheduled meeting points.

Lemma 3. *At the end of each phase, the explored territory is increased by one or two edges.*

We define a *1-phase* to be a phase in which exactly one edge is explored. Similarly, we define a *2-phase* to be a phase in which exactly two edges are explored. In view of Lemma 3, every phase is either a 1-phase or a 2-phase.

Lemma 4. *Let v be a meeting point at step t in a BHS-scheme. Then at least one of the following holds: $v = s$ or v is an endpoint of an edge which was already explored at step $t - 1$.*

Hence an agent which traversed an unexplored edge must return to the explored territory in order to go to the meeting point. A corollary of Lemmas 1, 2 and 4 is that at any step of a BHS-scheme the explored territory is connected.

A node p is called a *limit* of the explored territory at step t if it is incident both to an explored and to an unexplored edge.

A way of exploring exactly one edge in a phase is the following: one of the agents walks through the explored territory to its limit p, while the other agent walks through the explored territory to p, traverses an unknown edge and returns to p. If we assume that both agents are at a limit p of the explored territory at step t and (p, u) is an unknown edge towards node v, we define the following procedure:

probe(v): one agent traverses edge (p, u) (which is towards node v) and returns to node p to meet the other agent who waits. If they do not meet at step $t + 2$ then the black hole has been found.

We also define a procedure that the two agents could follow to explore two new edges in a phase. Suppose that the two agents reside at node m at step t. Let $p_1, ..., p_i$ be the limits of the explored territory at that step. Each of the unknown edges which could be explored in the following phase has to be incident to a node from the set $\{p_1, ..., p_i\}$. Let the two selected unknown edges for exploration be (k, p_k) and (l, p_l), $p_k, p_l \in \{p_1, ..., p_i\}$ (possibly $p_k = p_l$). We assume that node m belongs to the path $< k, l >$. The definition of the procedure is the following:

split(k, l): One of the agents traverses the path from node m to node k and returns towards node p_l. The other traverses the path from node m to node l and returns towards node p_k. Let $dist(l, k)$ denote the number of edges in the path from node k to node l. If they do not meet at step $t + dist(l, k)$ then the black hole has been found.

4 Black Hole Search in a Line

In this section we construct an optimal black hole search algorithm for lines, with linear time complexity. A line is a graph $L = (V, E)$, where $V = \{0, ..., n\}$

and $E = \{[i, i+1] : i = 0, 1, ..., n-1\}$. 0 and n are called endpoints of the line. The starting node is denoted by s, while a and b denote the distances between s and the endpoints of the line, with $a \leq b$, hence $a + b = n$. We assume $b > 0$, otherwise the line consists of a single node. We call *right* the direction from s towards the closer endpoint and *left* the other direction.

Theorem 1. *The time of any BHS-scheme on the line is at least:*

- $4n - 2$, *when* $a = 0$
- $\sum_{i=1}^{a} 2i$, *when* $1 \leq a = b \leq 5$
- $4n - 6$, *when* $a = 1 < b$
- $4n - 10$, *when* $a = 2 < b$ *or* $a = 3 < b$
- $4n - 8$, *when* $a = 4 < b$ *or* $a = 5 < b$ *or* $a \geq 6$

We will now give an optimal algorithm to solve the black hole search problem for the line (i.e. an algorithm which produces a fastest BHS-scheme for any line). Suppose that both agents reside at the same node m. The algorithm uses procedures *probe*, *split* and the following ones:

- **walk(k):** both agents go 1 step towards node k.
- **walk-and-probe(v):**
  while the position of the agents is not adjacent to node v **do**
    walk(v);
    probe(v)
- **return(s):**
  repeat walk(s) **until** all remaining agents are at s

The high-level description of Algorithm Line is the following:

- **case** $a = 0$**:** the two agents explore the line by probing left of s and return
- **case** $1 \leq a = b \leq 5$**:** the two agents explore the line by repeated splits
- **case** $a = 1 < b$**:** the two agents first do a split and then explore the rest of the line by probing left and return
- **case** $a = 2 < b$**:** the two agents first do a split, then explore all edges left of s except one by probing, and finally explore the last two edges by a split
- **case** $3 \leq a < b$ **or** $a \geq 5$**:** the two agents first do two splits, then explore all edges left of s except one by probing. They explore the last left edge together with an edge right of s by a split and finally explore the remaining edges (if any) which are right of s by probing and return

The precise formulation of the algorithm is given as Algorithm 1. The time complexity of the algorithm is linear.

Theorem 2. *Algorithm Line produces a fastest BHS-scheme for any line.*

The proofs of the results of this section are omitted due to lack of space and will appear in the full version of the paper.

Algorithm 1. Algorithm Line

```
case a = 0
        probe(0);
        walk-and-probe(0);
case 1 ≤ a = b ≤ 5
        for i := 1 to a
            split(s − i, s + i);
case a = 1 < b
        split(s − 1, s + 1);
        walk-and-probe(0);
case a = 2 < b
        split(s − 1, s + 1);
        walk-and-probe(1);
        split(0, s + 2);
case a = 3 < b
        split(s − 1, s + 1);
        split(s − 2, s + 2);
        walk(s − 1);
        walk-and-probe(1);
        split(0, s + 3);
case 4 ≤ a < b OR a ≥ 6
        split(s − 1, s + 1);
        split(s − 2, s + 2);
        walk(s − 1);
        walk-and-probe(1);
        split(0, s + 3);
        walk(s + 2);
        walk-and-probe(n);
return(s)
```

5 Black Hole Search in a Tree

In this section we study the problem of black hole search in trees.

Consider a tree T rooted at the starting node s. If e is an edge, $e = (u, v)$ means that v is the child of u. Let $e = (u, v)$ be an edge of the tree. Consider the following coloring which creates a partition of the edges of the tree. This partition will be used in the analysis of our algorithms.

- assign red color to edge e if node v has at least two descendants,
- assign green color to edge e if v is a leaf and exactly one of the following holds: $u = s$ or the edge (t, u) is a red edge (where t is the parent of u),
- assign blue color to edge e if it has none of the above properties

Let $e = (u, v)$ and $e' = (v, z)$ be two blue edges such that v is the unique child of u and z is a leaf and the unique child of v. We call the set of these two edges a *branch*. The set of all branches of blue edges with upper node u is called a *block*.

Lemma 5. *In any BHS-scheme, the following holds: a green edge has to be traversed by the agents at least* 2 *times, a red edge has to be traversed at least* 6 *times and a branch of blue edges requires a total of at least* 6 *traversals.*

Proof. By Lemma 1 any edge has to be traversed 2 times by one agent to become explored. In particular a green edge needs 2 traversals.

Consider a red edge $e = (u, v)$. Let l be the number of descendants of node v. In view of Lemmas 1 and 2, if during any phase after exploration edge e is traversed always by only one agent then at least $2l \geq 4$ additional traversals are required (an agent has to traverse e two times for every descendant of v). If there is at least one phase after exploration of e where the edge is traversed by both agents then at least 4 additional traversals of e are required for the exploration of the edges with upper node v (both agents traverse e and return). Thus the total minimum number of traversals is 6.

A branch of 2 blue edges can be traversed in the following ways. 2 traversals are required for the exploration of the upper edge of the branch. If during any phase after exploration of the upper edge, this edge is traversed always by only one agent then at least 4 additional edge traversals on this branch are required. If there is at least one phase after exploration of the upper edge when this edge is traversed by both agents then at least 6 additional edge traversals on this branch are required (both agents traverse the upper edge, then one of them explores the lower edge and finally they return). Therefore the total minimum number of traversals on each branch is 6.

Lemma 6. *Any BHS-scheme requires at least* 3, 1 *and* 3b *time units for the traversals of a red edge, a green edge and a block of* b *branches of blue edges, respectively.*

5.1 An Optimal Algorithm for a Family of Trees

Consider the family $\mathcal{T}$ of rooted trees with the following property: any internal node of a tree in $\mathcal{T}$ (including the root) has at least 2 children. Trees in $\mathcal{T}$ will be called *bushy trees*.

Let T be a bushy tree with root s and let u be an internal node of T. The *heaviest child* $v = H(u)$ of u is defined as a child v of u such that the subtree $T(v)$ rooted at v (which is also a bushy tree) has a maximum height among all subtrees rooted at children of u. The *lightest* child $v' = L(u)$ of u is defined as a child v' of u such that the subtree $T(v')$ rooted at v' has a minimum height among all subtrees rooted in a child of u. Ties are broken arbitrarily. Notice that $H(u)$ and $L(u)$ can be computed for all nodes u in linear time.

The high-level description of Algorithm Bushy-Tree is the following. Let m be the meeting point of the two agents after a phase (initially $m = s$).

- Explore any pair of unknown edges (m, x), (m, y) with upper node m by executing procedure $split(x, y)$, leaving edge $(m, L(m))$ last.
- If there is one unknown edge with upper node m (which must be $(m, L(m))$) explore this edge together with another unknown edge (if any) again using

procedure *split*. If edge $(m, L(m))$ is the last unknown edge in the tree, explore it by executing procedure *probe*$(L(m))$.
- If all edges with upper node m are explored, explore similarly as before any unknown edges incident to the children of m and to ancestors of m.

Below we give the precise formulation of the algorithm.

Algorithm. Bushy-Tree
special-explore(s)

Procedure special-explore(v)
for every pair of unknown edges $(v, x), (v, y)$ with upper node v **do**
split(x, y), so that edge $(v, L(v))$ is explored last
end for
if every edge is explored **then**
repeat walk(s) **until** (all remaining agents are at s)
else
case 1: every edge incident to v has been explored
$next$:= relocate(v);
special-explore($next$);
case 2: there is an unknown edge (v, z) incident to v
(must be $z = L(v)$ *)*
explore-only-child$(v, next)$;
special-explore($next$);
end if

Function relocate(v) takes as input the current node v where both agents reside and returns the new location of the two agents. If there is an unknown edge incident to a child of v then the agents go to that child. Otherwise the two agents go to the parent of v.

Function relocate(v)
case 1.1: $\exists$ an unknown edge incident to $w \in children(v)$
walk(w);
relocate := w
case 1.2: every edge incident to any child of v is explored
let t be the parent of v;
walk(t);
relocate := t

Procedure explore-only-child$(v, next)$ takes as input the current node v where both agents reside and returns the new meeting point after the exploration of edge $(v, L(v))$. The description of the procedure is the following:

- If there is an unknown edge incident to a child w of v, $w \neq L(v)$, then the agents explore edge $(w, H(w))$ together with edge $(v, L(v))$ by *split*$(H(w), L(v))$. The new meeting point is w.
- If every edge incident to any child w of v, different from $L(v)$, is explored and edge $(v, L(v))$ is not the last unknown edge in the tree, then find the deepest ancestor a of v with unknown edges whose upper node is a descendant of a; the agents explore edge $(D(a), H(D(a)))$ (where $D(a)$ is the closest descendant of a with incident unknown edges), together with edge $(v, L(v))$, by *split*$(H(D(a)), L(v))$; the new meeting point is $D(a)$.
- If edge $(v, L(v))$ is the last unknown edge in the tree then explore it by calling *probe*$(L(v))$; the new meeting point is v.

Procedure explore-only-child$(v, next)$

case 2.1: there is an unknown edge incident to $w \in children(v)$, $w \neq L(v)$
split($L(v)$,$H(w)$);
$next := w$

case 2.2: every edge incident to any $w \in children(v)$, $w \neq L(v)$ is explored
($L(v)$ must be a leaf *)*

case 2.2.1: there are at least 2 unknown edges left
let a be the deepest ancestor of v such that:
$D(a) :=$ the closest descendant of a with incident unknown edges;
split($H(D(a))$, $L(v)$);
$next := D(a)$

case 2.2.2: there is only 1 unknown edge left
probe($L(v)$);
$next := v$

Notice that all edges of the tree (except possibly the last one if the number of edges is odd) are explored by calling procedure split. Observe that in any bushy tree, there are only *red* and *green* edges. By definition, in every red edge $e_r = (u_r, v_r)$, node v_r has at least two children and every leaf of the tree is an endpoint of a green edge $e_g = (u_g, v_g)$. Also u_g has at least two children.

Since all values $H(u)$ and $L(u)$ can be computed in linear time it is easy to see that time complexity of Algorithm Bushy-Tree is linear.

Theorem 3. *Algorithm Bushy-Tree produces a fastest BHS-scheme for any bushy tree.*

Sketch of the proof: The scheme produced by Algorithm Bushy-Tree traverses any red edge 6 times and any green edge 2 times. Moreover every phase is a 2-phase (i.e. the two agents traverse edges in parallel), except possibly the last phase (in the case when the number of edges is odd), and no agent waits in any 2-phase.

5.2 An Approximation Algorithm for Trees

In this section we give an approximation algorithm with ratio $\frac{5}{3}$ for the black hole search problem, working for arbitrary trees (i.e. an algorithm which produces a BHS-scheme whose time is at most 5/3 of the shortest possible time, for every input).

The high-level description of Algorithm Tree is the following. Let v be the meeting point of the two agents after a phase (initially $v = s$); the edges with upper node v are explored by calling procedure split until either all such edges are explored or there is at most one remaining unknown edge incident to v, which is explored by calling procedure probe; this is repeated for any child of v. The precise formulation of the algorithm is given below. Apart from procedures split and probe it uses function relocate defined in the previous section. The time-complexity of Algorithm Tree is linear.

Algorithm. Tree
explore(s)

Procedure explore(v)
for every pair of unknown edges $(v, x), (v, y)$ incident to v **do**
split(x, y);
end for
if there is only one remaining unknown edge (v, z) incident to v **then**
probe(z);
end if
if every edge is explored **then**
repeat walk(s) **until** both agents are at s
else
next := relocate(v);
explore(*next*)
end if

Lemma 7. *Let u be a node which is neither a leaf nor a middle of a branch of blue edges. Let d be the down degree of u. Let β be the number of branches of blue edges with upper node u, ρ the number of red edges with upper node u and γ the number of green edges with upper node u. Algorithm Tree spends at most $d + 4\beta + 2\rho$ time units if d is even, and $d + 1 + 4\beta + 2\rho$ time units if d is odd for the traversals of all the above edges.*

Theorem 4. *Algorithm Tree achieves $\frac{5}{3}$ approximation ratio.*

Proof. If the tree consists of a single edge, then the ratio is one. Otherwise, suppose that the tree has k nodes $u_1, u_2, ..., u_k$ such that $\forall u_i \exists v_j$ $(e_{ij} = (u_i, v_j))$ is a red edge, a green edge or an upper blue edge in a branch of blue edges. In

any case, $\forall u_i \neq s$ u_i has at least two descendants, hence (u'_i, u_i) is a red edge. Thus there are at least $k-1$ red edges in the tree. Let d_i: $i = 1, ..., k$ be the down degree of u_i. Suppose that d_i: $i = 1, ..., l$ is odd and d_i: $i = l+1, ..., k$ is even. Let β_i be the number of branches of blue edges with upper node u_i, ρ_i the number of red edges with upper node u_i and γ_i the number of green edges with upper node u_i. We have $d_i = \beta_i + \rho_i + \gamma_i$.

According to Lemma 6, any BHS-scheme must spend at least $3\beta_i + 3\rho_i + \gamma_i$ time units on the traversals of all red edges, green edges and branches of blue edges with upper node u_i. Hence in view of Lemma 7 the ratio between the time of our scheme and the fastest possible scheme is at most:

$$\frac{\sum_{i=1}^{l}(d_i + 1 + 4\beta_i + 2\rho_i) + \sum_{i=l+1}^{k}(d_i + 4\beta_i + 2\rho_i)}{\sum_{i=1}^{k}(3\beta_i + 3\rho_i + \gamma_i)} = \frac{\sum_{i=1}^{k}(5\beta_i + 3\rho_i + \gamma_i) + l}{\sum_{i=1}^{k}(3\beta_i + 3\rho_i + \gamma_i)}$$

The above ratio is $\leq \frac{5}{3}$ when $3l \leq 6\sum_{i=1}^{k}\rho_i + 2\sum_{i=1}^{k}\gamma_i$. Since $\sum_{i=1}^{k}\rho_i \geq k-1$, this ratio is lower or equal to $\frac{5}{3}$ when

$$6(k-1) + 2\sum_{i=1}^{k}\gamma_i \geq 3l \tag{1}$$

If $k-1 \geq l$ (i.e. there is at least one node of even down degree) then inequality (1) is true.

If $k-1 < l$ it means that $l = k$. This is the situation when every vertex u_i has an odd lower degree. If $k \geq 2$, inequality (1) still holds. If $k = 1$ then there is no red edge ($u_1 = s$). As long as there are at least two green edges, inequality (1) is true. Otherwise one of the following holds:

- The tree consists of a block of β_1 branches of blue edges where β_1 is even, and one green edge. In this case the total number of edges in the tree is odd. Hence, in any BHS-scheme at least one edge must be explored in a 1-phase. We prove that any BHS-scheme has to spend at least $3\beta_1 + 2$ time units for all the traversals. According to Lemma 5 the total number of traversals needed is at least $6\beta_1 + 2$. At least 2 of the traversals are done during a 1-phase and require at least 2 time units. Therefore the time needed in this case is at least $\frac{6\beta_1}{2} + 2 = 3\beta_1 + 2$.
  According to Lemma 7, the scheme produced by Algorithm Tree uses $d_1 + 1 + 4\beta_1 = 5\beta_1 + 2$ time units. Thus the ratio is at most $\frac{5\beta_1+2}{3\beta_1+2} \leq \frac{5}{3}$.
- The tree consists of a block of β_1 branches of blue edges where β_1 is odd. If $\beta_1 = 1$ then the ratio is one. Otherwise we prove that any BHS-scheme has to spend in this case at least $3\beta_1 + 1$ time units for all traversals.
  - If there is an edge in a branch which has been traversed by both agents during a phase then the total number of edge traversals in that branch is 8. Therefore in view of Lemma 5, the total number of traversals is at least $6(\beta_1 - 1) + 8$ and the time needed is at least $\frac{6\beta_1+2}{2} = 3\beta_1 + 1$.

- Otherwise, if there is at least one edge that has been explored during a 1-phase then the total number of traversals done during *2-phases* is at most $6\beta_1 - 2$ by Lemma 5, while there are 2 traversals done in a 1-phase which requires 2 time units. Therefore the time needed is at least $\frac{6\beta_1-2}{2} + 2 = 3\beta_1 + 1$.
- The remaining case is that every edge is explored during a 2-phase and there is no edge which has been traversed by both agents during a phase. Since the number of upper edges in branches is odd, there must be a 2-phase ϕ during which an upper edge of a branch is explored together with a lower edge of another branch. The time needed for this phase is at least 4 time units since both agents cannot traverse the same edge. In view of Lemma 5 the total number of traversals in every phase except ϕ is at least $6(\beta_1 - 2) + 2 + 4$ (there is a branch on which only 2 traversals are done and a branch on which only 4 traversals are done). Hence the time needed in this case is at least $\frac{6\beta_1-6}{2} + 4 = 3\beta_1 + 1$.

According to Lemma 7, the time of the scheme produced by Algorithm Tree is $d_1 + 1 + 4\beta_1 = 5\beta_1 + 1$ time units. Thus in all three cases the ratio is at most $\frac{5\beta_1+1}{3\beta_1+1} \leq \frac{5}{3}$.

Notice that there exists a family of trees in which the approximation ratio achieved by Algorithm Tree is exactly 5/3. This family includes all trees which consist of an even number β of branches of blue edges. According to Lemma 7, the time of the scheme produced by Algorithm Tree is $\beta + 4\beta = 5\beta$ for such a tree, while the fastest BHS-scheme for this tree requires exactly 3β time units (for example, all upper edges are explored two by two by calling procedure split and then all lower edges are explored in the same way).

6 Conclusion

We presented algorithms for the black hole search problem on trees. For arbitrary trees we gave a 5/3-approximation algorithm, and for two classes of trees (lines and trees all of whose internal nodes have at least 2 children) we gave optimal algorithms, i.e., methods of constructing a shortest possible black hole search scheme for any input in the class. The time complexity of all our algorithms is linear in the size of the input.

It remains open if there exists a polynomial time algorithm to construct a fastest black hole search scheme for an arbitrary tree. More generally, we do not know if the problem is polynomial for arbitrary graphs. We conjecture that the answer to the latter question is negative. Hence it seems interesting to find good approximation algorithms for the black hole search problem on arbitrary graphs. It should be noted that a trivial scheme, proceeding along any spanning tree of the graph using walk-and-probe and returning to the starting node, provides a 4-approximation algorithm for this problem.

References

1. S. Dobrev, P. Flocchini, R. Kralovic, G. Prencipe, P. Ruzicka, N. Santoro, Black hole search by mobile agents in hypercubes and related networks, Proc. of Symposium on Principles of Distributed Systems (OPODIS 2002), 171-182.
2. S. Dobrev, P. Flocchini, G. Prencipe, N. Santoro, Mobile agents searching for a black hole in an anonymous ring, Proc. of 15th International Symposium on Distributed Computing, (DISC 2001), 166-179.
3. S. Dobrev, P. Flocchini, G. Prencipe, N. Santoro, Searching for a black hole in arbitrary networks: Optimal Mobile Agents Protocols, Proc. 21st ACM Symposium on Principles of Distributed Computing (PODC 2002), 153-161.
4. S. Dobrev, P. Flocchini, G. Prencipe, N. Santoro, Multiple agents rendezvous on a ring in spite of a black hole, Proc. Symposium on Principles of Distributed Systems (OPODIS 2003).
5. F. Hohl, Time limited black box security: Protecting mobile agents from malicious hosts, Proc. Conf. on Mobile Agent Security (1998), LNCS 1419, 92-113.
6. F. Hohl, A framework to protect mobile agents by using reference states, Proc. 20th Int. Conf. on Distributed Computing Systems (ICDCS 2000), 410-417.
7. S. Ng, K. Cheung, Protecting mobile agents against malicious hosts by intention of spreading, Proc. Int. Conf. on Parallel and Distributed Processing and Applications (PDPTA'99), 725-729.
8. T. Sander, C.F. Tschudin, Protecting mobile agents against malicious hosts, Proc. Conf. on Mobile Agent Security (1998), LNCS 1419, 44-60.
9. K. Schelderup, J. Ines, Mobile agent security – issues and directions, Proc. 6th Int. Conf. on Intelligence and Services in Networks, LNCS 1597 (1999), 155-167.
10. J. Vitek, G. Castagna, Mobile computations and hostile hosts, in: Mobile Objects, D. Tsichritzis, Ed., University of Geneva, 1999, 241-261.