Almost-Optimal Deterministic Treasure Hunt in Arbitrary Graphs

Sébastien Bouchard
Univ. Bordeaux, Bordeaux INP, CNRS, LaBRI, UMR5800, F-33400 Talence, France

Yoann Dieudonné
MIS Lab., Université de Picardie Jules Verne, Amiens, France

Arnaud Labourel
Aix Marseille Univ, Université de Toulon, CNRS, LIS, Marseille, France

Andrzej Pelc
Département d'informatique, Université du Québec en Outaouais, Gatineau, Canada

Abstract

A mobile agent navigating along edges of a simple connected graph, either finite or countably infinite, has to find an inert target (treasure) hidden in one of the nodes. This task is known as treasure hunt. The agent has no *a priori* knowledge of the graph, of the location of the treasure or of the initial distance to it. The cost of a treasure hunt algorithm is the worst-case number of edge traversals performed by the agent until finding the treasure. Awerbuch, Betke, Rivest and Singh [3] considered graph exploration and treasure hunt for finite graphs in a restricted model where the agent has a fuel tank that can be replenished only at the starting node s. The size of the tank is $B = 2(1+\alpha)r$, for some positive real constant α, where r, called the radius of the graph, is the maximum distance from s to any other node. The tank of size B allows the agent to make at most $\lfloor B \rfloor$ edge traversals between two consecutive visits at node s.

Let $e(d)$ be the number of edges whose at least one extremity is at distance less than d from s. Awerbuch, Betke, Rivest and Singh [3] conjectured that it is impossible to find a treasure hidden in a node at distance at most d at cost nearly linear in $e(d)$. We first design a deterministic treasure hunt algorithm working in the model without any restrictions on the moves of the agent at cost $\mathcal{O}(e(d) \log d)$, and then show how to modify this algorithm to work in the model from [3] with the same complexity. Thus we refute the above twenty-year-old conjecture. We observe that no treasure hunt algorithm can beat cost $\Theta(e(d))$ for all graphs and thus our algorithms are also almost optimal.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases treasure hunt, graph, mobile agent

Digital Object Identifier 10.4230/LIPIcs.ICALP.2021.36

Category Track A: Algorithms, Complexity and Games

Related Version *Full Version*: `https://arxiv.org/abs/2010.14916`

Funding *Andrzej Pelc*: Partially supported by NSERC discovery grant 2018-03899 and by the Research Chair in Distributed Computing at the Université du Québec en Outaouais.

1 Introduction

The background. A mobile agent has to find an inert target (treasure) in some environment that can be a network modeled by a graph or a terrain in the plane. This task, known as *treasure hunt*, has important applications when the environment is dangerous for humans. When a miner is lost in a contaminated mine, it may have to be found by a robot, and the length of the robot's trajectory should be as short as possible, in order to minimize rescuing time. In this example, a graph models the corridors of the mine with nodes representing crossings. Another application of treasure hunt in graphs is searching for a data item in a communication network modeled by a graph.

The models and the problem. We consider a simple connected undirected locally finite graph $\mathcal{G} = (V_{\mathcal{G}}, E_{\mathcal{G}})$, i.e., a graph with nodes of finite degrees. Such a graph can be either finite or countably infinite. A mobile agent (robot) starts at a node s of $\mathcal{G}$, called the *source node*, and moves along its edges. The maximum distance of any node from s is denoted by r and called the *radius* of the graph (the radius of countably infinite graphs is infinite). We make the same assumption as in [3] that the agent has unbounded memory and can recognize already visited nodes and traversed edges. This is formalized as follows. Nodes of $\mathcal{G}$ have distinct labels that are positive integers. Each edge has ports at both of its extremities. Ports corresponding to edges incident to a node of degree δ are numbered $0, 1, \dots, \delta - 1$ in an arbitrary way. At the beginning, the agent situated at node s sees its degree. The agent executes a deterministic algorithm: at each step, it selects a port number on the basis of currently available information, and traverses the corresponding edge. When the agent enters the adjacent node, it learns its label, its degree, and the incoming port number. Each node of $V_{\mathcal{G}}$ will be identified with its label, and each edge of $E_{\mathcal{G}}$ will be identified as the quadruple (v, w, p, q), where $v < w$ are labels of the edge extremities, p is its port number at node v and q is its port number at node w.

The above simple model will be called *unrestricted.* However, some authors imposed additional restrictions, in the case when the graph is finite. The authors of [3] used a restriction of moves of the agent that we will call the *fuel-restricted model.* They assumed that the agent has a fuel tank that can be replenished only at the starting node s of the agent. The size of the tank is $B = 2(1+\alpha)r$, for some positive real constant α, where r is the radius of the graph. The tank of size B allows the agent to make at most $\lfloor B \rfloor$ edge traversals between two consecutive visits at node s. The restriction used in [11] was of a different kind. We will call it the *rope-restricted model.* It was assumed in [11] that the agent is *tethered*, i.e., attached to s by a rope that it unwinds by a length 1 with every forward edge traversal and rewinds by a length of 1 with every backward edge traversal. The rope is infinitely extendible but has to satisfy the following constraint: the segment of the rope unwinded by the agent must never be longer than $L = (1 + \alpha)r$, for some positive real constant α. Hence the agent is forced to match every forward edge traversal of an edge with a backward edge traversal, rewinding the rope, in a first-in last-out stack order.

The task of treasure hunt, in any of the above three models, is defined as follows. An adversary hides the treasure in some node of the underlying graph $\mathcal{G}$. The agent has no *a priori* knowledge of the graph, of the location of the treasure or of the initial distance to it, and has to find the treasure. The *cost* of a treasure hunt algorithm is the worst-case number of edge traversals performed by the agent until finding the treasure.

In order to state our problem we need the notion of a *ball.* Given a non-negative integer k, a graph G, and a node u, the ball $B_k(G, u)$ is defined as the subgraph $K = (V_K, E_K)$ of G, where V_K is the set of all nodes at distance at most k from u in G, and E_K is the set of all edges of G whose at least one extremity is at distance smaller than k from u in G. (Thus $B_k(G, u)$ is the subgraph of G induced by nodes at distance at most k without edges joining nodes at distance exactly k from u in G). The number of edges in ball $B_k(\mathcal{G}, s)$, where s is the source node, will be denoted by $e(k, \mathcal{G})$. Whenever the graph $\mathcal{G}$ is clear from the context, we will write $e(k)$ instead of $e(k, \mathcal{G})$.

The main problem considered in this paper is inspired by the following conjecture of Awerbuch, Betke, Rivest and Singh [3], formulated for their fuel-restricted model:

Is it possible (we conjecture not) to find a treasure in time nearly linear in the number of those vertices and edges whose distance to the source is less than or equal to that of the treasure? [1]

Our results. Our main result refutes the above twenty-year-old conjecture. Let d be any integer such that $1 < d \leq r$, where r is the radius of the underlying graph $\mathcal{G}$. We first design a deterministic treasure hunt algorithm working in the unrestricted model and always finding a treasure located at distance at most d from the source node, at cost $\mathcal{O}(e(d)\log d)$. We then show how to modify this algorithm to work in the fuel-restricted and rope-restricted models with the same complexity. Since $d \leq e(d)$, the cost of our algorithms differs from $e(d)$ only by a logarithmic factor, and hence it is nearly linear in $e(d)$, contrary to the conjecture. Due to the ignorance of the agent concerning the graph in which it operates, it can be easily shown that no treasure hunt algorithm can beat cost $\Theta(e(d))$ for all graphs and thus our algorithms are also almost optimal. The main difficulty is to design the algorithm for the unrestricted model. This algorithm is then suitably modified for each of the two restricted models.

Solving the problem of treasure hunt at a cost quasi-linear in $e(d)$ required to respect two fundamental principles, whose joint implementation seemed precarious in the light of the existing literature.

The first one is a *prudence principle.* It consists in never getting "for too long" beyond the unknown distance d in order to guarantee a cost that depends on $e(d)$. This can be ideally achieved by emulating BFS. However, since in such an emulation the agent must physically move from one node to the next, it may be forced to traverse $\Omega(e(d)^2)$ edges before finding the treasure, in some graphs. In particular, this could be the case when $\mathcal{G}$ is an infinite line.

The second principle is what we could call an *efficiency principle.* It consists in getting a cost that is asymptotically close to the number of edges of the subgraph that has been explored till finding the treasure, if the treasure is far away. This can be ideally achieved using the treasure hunt algorithm of [11], the cost of which is linear in the number of edges of the explored subgraph. However, using this algorithm, the agent may go for too long beyond the unknown distance d and consequently the cost of treasure hunt could not be upper bounded by any function of $e(d)$. The key challenge overcome by our work was combining these two principles within the same algorithm. It is precisely the combination of prudence with efficiency that finally made possible the design of an almost-optimal treasure hunt algorithm.

Due to lack of space, the proofs of several results are omitted.

Related work. The task of treasure hunt, i.e., finding an inert target hidden in some environment, has been studied for over fifty years [5, 6, 7]. The environment where the target is hidden may be a graph or a plane, and the search may be deterministic or randomized. The book [1] surveys both treasure hunt and the related rendezvous problem, where the target and the searching agent are both mobile and they cooperate to meet. This book is concerned mostly with randomized search strategies. In [23, 26] the authors studied relations between treasure hunt and rendezvous in graphs. The authors of [4] studied the task of treasure hunt on the line and in the grid, and initiated the study of the task of searching for an unknown line in the plane. This research was continued, e.g., in [16, 21].

Several papers considered treasure hunt in the plane, see surveys [14, 15]. In [20], the author designs an optimal algorithm to sweep the plane in order to locate an unknown fixed target, where locating means getting the agent originating at point O to a point P such that the target is in the segment OP. In [13], the authors generalized the search problem in the

[1] Here time is what we call cost, i.e., the worst-case number of edge traversals until finding the treasure.

plane to the case of several searchers. Efficient treasure hunt in the plane, under complete ignorance of the searching agent, was studied in [24]. Treasure hunt on the line (called the cow-path problem [17]) has been also generalized to the environment consisting of multiple rays originating at a single point [2, 10, 22, 25].

In [12], the authors considered treasure hunt in several classes of graphs including trees. Treasure hunt in trees was studied in [8, 9, 18]. In [8, 9], the authors considered complete b-ary trees, and in [18], treasure hunt was studied in symmetric trees, with possibly multiple treasures.

In [19, 23], treasure hunt in graphs was considered under the advice paradigm, where a given number of bits of advice can be given to the agent, and the issue is to minimize this number of bits. The impact of different types of knowledge on the efficiency of the treasure hunt problem restricted to symmetric trees was studied in [18].

The two papers closest to the present work are [3, 11]. Both of them are mainly interested in exploration of finite unknown graphs but they both get interesting corollaries for the treasure hunt problem. [3] adopts the fuel-restricted model and [11] adopts the rope-restricted model. In [3], the authors get a treasure hunt algorithm working at cost $O(E + V^{1+o(1)})$, where E (resp. V) is the number of edges (resp. nodes) in a ball $B_\Delta(\mathcal{G}, s)$, with $\Delta \leq d + o(d)$, if the treasure is at distance at most d from the starting node of the agent. Since $e(\Delta)$ may be a lot larger than $e(d)$, this does not permit to bound the cost of the algorithm by any function of $e(d)$. This impossibility may be the reason for their conjecture that we refute in this paper. In [11], the authors design, for any constant $0 < \alpha < 1$, a treasure hunt algorithm whose cost is linear in $e((1 + \alpha)d)$. Again, since $e((1 + \alpha)d)$ may be much larger than $e(d)$, this does not permit to bound the cost of the algorithm by any function of $e(d)$.

2 Preliminaries

In this section we introduce some conventions, definitions and procedures that will be used to describe and analyze our algorithm.

Consider any graph $H = (V_H, E_H) \subseteq \mathcal{G}$. If H is finite, its size i.e., its number of edges is denoted by $|H|$. A graph is said to be *empty* if it contains no node. In the rest of this section, we assume that H is not empty.

Let u and v be two (not necessarily distinct) nodes of H. We say that a sequence of i integers $(x_1, x_2, \dots, x_i)$ is a *path* (of length i) in H from node u to v iff (1) $i = 0$ and $u = v$, or (2) there exists an edge e in H between node u and a node w of H such that the port number of edge e at node u is x_1 and $(x_2, \dots, x_i)$ is a path from node w to v in H. The lexicocraphically smallest shortest path from node u to v in H, if any, is denoted by $P_H(u, v)$, and the length of this path is denoted by $|P_H(u, v)|$. The distance between u and v in H is denoted by $d_H(u, v)$ and is equal to $|P_H(u, v)|$ if $P_H(u, v)$ exists, ∞ otherwise. If H is finite and connected, the eccentricity $\epsilon_H(u)$ of node u is defined as $\max_{w \in V_H} d_H(u, w)$. The degree of u in H will be denoted by $deg_H(u)$, or simply by $deg(u)$ if $H = \mathcal{G}$. We say that node u is *incomplete* (resp. *complete*) in H if $deg_H(u) < deg(u)$ (resp. $deg_H(u) = deg(u)$). We also say that a port p is free at node u in H, if $p \leq deg(u) - 1$ and there is no edge $(u, *, p, *)$ or $(*, u, *, p)$ in E_H.

We will often need to handle subgraphs of $\mathcal{G}$ through union and intersection operations. More precisely, given two subgraphs $H' = (V_{H'}, E_{H'})$ and $H'' = (V_{H''}, E_{H''})$ of $\mathcal{G}$, the union of (resp. the intersection of) H' and H'' is denoted by $H' \sqcup H''$ (resp. $H' \sqcap H''$) and is equal to $(V_{H'} \cup V_{H''}, E_{H'} \cup E_{H''})$ (resp. $(V_{H'} \cap V_{H''}, E_{H'} \cap E_{H''})$).

We define the boundary of a ball $B_f(\mathcal{G}, s)$, where s is the source node, as the set of nodes of $B_f(\mathcal{G}, s)$ that are incomplete in $B_f(\mathcal{G}, s)$.

To design our algorithm, we will also make use of three basic routines presented below. The first routine is `MoveTo`(H, v). Assuming that the agent currently occupies a node w of H and $P_H(w, v)$ exists, this routine moves the agent from node w to node v by following path $P_H(w, v)$. The second routine is `IncompleteNodes`(v, H, l) where l is a positive integer. This routine returns the set of all nodes w of H such that $d_H(v, w) \leq l$ and w is incomplete in H. The third routine is `Nodes`$(\mathcal{S})$, where $\mathcal{S}$ is a finite set of finite subgraphs of $\mathcal{G}$. This routine returns the union of all nodes in all subgraphs from $\mathcal{S}$.

Given an execution $\mathcal{E}$ of a series of instructions, the cost of $\mathcal{E}$ is the number of edge traversals performed by the agent during $\mathcal{E}$.

We will use the following convention. The agent will sometimes need to use Depth First Search traversal of graphs (not performed physically, but performed as a computation in the memory of the agent). Such a traversal depends on the order in which edges incident to a given node are traversed for the first time. We fix this order as the increasing order of port numbers at the given node. In this way the traversal is unambiguous, and we call it DFS.

3 Intuition

The purpose of this section is to sketch an intuitive overview of our algorithm that allows to find the treasure at an almost-optimal cost in the unrestricted model. To this end and to simplify the discussion, we will assume that the underlying graph $\mathcal{G}$ is countably infinite with nodes of finite degrees. We will rely on the notion of *largest explored ball*. By "largest explored ball", at a given phase of treasure hunt, we mean the ball $B_f(\mathcal{G}, s)$ where f is the largest integer such that each edge of $B_f(\mathcal{G}, s)$ has been traversed at least once. This largest integer f is the radius of the largest explored ball.

At a high level, our algorithm works in phases $i = 1, 2, 3, \ldots$ and immediately stops as soon as the treasure is found. At the beginning of phase i, the agent is located at node s and the radius of the largest explored ball is equal to f_i. The goal for the agent is to terminate the phase at node s while satisfying at least one of the following three conditions unless, of course, the treasure has been found before.

- *Condition 1.* The agent has entirely explored ball $B_{f_i+1}(\mathcal{G}, s)$, $e(f_i + 1) \geq 2e(f_i)$ and the cost of the phase is $\mathcal{O}(e(f_i + 1))$.
- *Condition 2.* The agent has entirely explored ball $B_{2f_i}(\mathcal{G}, s)$, $f_i \geq 1$ and the cost of the phase is $\mathcal{O}(e(f_i))$.
- *Condition 3.* The agent has entirely explored ball $B_{f_i+k}(\mathcal{G}, s)$ for some positive integer k, $e(f_i + k + 1) \geq 2e(f_i)$, $f_i \geq 2$, and the cost of the phase is $\mathcal{O}(e(f_i) \log f_i)$.

Actually, the conditions we really seek to meet in our algorithm are a little more intricate than those presented above, because we needed stronger technical requirements to refute the conjecture of Awerbuch, Betke, Rivest and Singh [3]. However, this would add an unnecessary level of complexity to understand the intuition, hence we omit these technical details here.

Before seeing how we implement our strategy, let us briefly examine why it permits us to get a cost quasi-linear in $e(d)$. Since $f_1 = 0$ and the radius of the largest explored ball increases by at least one during each phase in which the treasure is not found, the agent necessarily finds the treasure by the end of some phase $\lambda \leq d$, and $f_i < f_\lambda < d$ for every $1 \leq i < \lambda$. During each phase satisfying Condition 1, the size of the largest explored ball at least doubles, which means that the total cost of these phases is upper bounded by twice the worst-case cost of the last phase satisfying Condition 1 i.e., $\mathcal{O}(e(f_\lambda+1))$. Concerning the phases fulfilling Condition 2, their number is at most $\mathcal{O}(\log(f_\lambda+1))$ and the cost of each of them cannot be more than $\mathcal{O}(e(f_\lambda))$, which implies that their total cost is $\mathcal{O}(e(f_\lambda)\log(f_\lambda+1))$. It remains to consider the case of the phases satisfying Condition 3. Given such a phase i, we have the guarantee that the size of the largest explored ball at least doubles between the beginning of phase i and the end of phase $i+1$, provided phase $i+1$ exists and is not prematurely interrupted by the discovery of the treasure. Indeed, at the end of phase i, the agent has at least entirely explored ball $B_{f_i+k}(\mathcal{G},s)$ for some positive integer k and $e(f_i+k+1) \geq 2e(f_i)$, while at the end of the (not prematurely interrupted) phase $i+1$ the agent has at least entirely explored ball $B_{f_{i+1}+1}(\mathcal{G},s)$ with $f_{i+1} \geq f_i+k$. Using this, it can be shown that the total cost of the phases satisfying Condition 3 is at most four times the worst-case cost of the last phase satisfying this condition i.e., $\mathcal{O}(e(f_\lambda)\log(f_\lambda+1))$. Given that the last phase λ can be viewed as a truncated phase that should have normally satisfied one of the three conditions, our sketch of analysis leads to the conclusion that the cost incurred by the agent till the discovery of the treasure is in $\mathcal{O}(e(f_\lambda+1)\log(f_\lambda+1))$, which is $\mathcal{O}(e(d)\log d)$ and is in line with our expectations.

Having justified the pertinence of such a strategy, we can turn our attention to its implementation. To do so, we need to introduce a technical building block, which we call `GlobalExpansion`(l,m) and to which we will go back at the end of this section to give additional details. Always executed from the source node s, it is a function that returns a boolean and whose two input parameters are positive integers except m that may be sometimes equal to the special symbol $\perp$. Assuming that $B_f(\mathcal{G},s)$ is the largest explored ball, the execution of `GlobalExpansion`$(l,\perp)$ permits the agent to traverse all the edges of $B_{f+l}(\mathcal{G},s)$ that are outside of $B_f(\mathcal{G},s)$ before coming back to node s. Under the same assumption, the execution of `GlobalExpansion`(l,m), when m is a positive integer, consists for the agent in acting as if m was $\perp$ but with the following extra requirement: as soon as more than m distinct edges outside of $B_f(\mathcal{G},s)$ have been traversed during the execution of the function, the agent backtracks to node s and aborts this execution. If m is $\perp$ or at least large enough to avoid an aborted execution, the agent ends up exploring $B_{f+l}(\mathcal{G},s)$ and the function returns true. Otherwise, the function returns false. It should be stressed that all of this is made while guaranteeing two properties. The first one is that the agent is always in $B_{f+2l-1}(\mathcal{G},s)$ during the execution of `GlobalExpansion`(l,m). The second is that the cost of the execution of `GlobalExpansion`(l,m) is $\mathcal{O}(e(f+2l-1))$ (resp. $\mathcal{O}(\min\{e(f)+m, e(f+2l-1)\})$) when $m=\perp$ (resp. $m\neq\perp$). Both these properties will turn out to be crucial to ensure a proper design of the phases. Finally, even if by chance the agent could explore a larger ball, we will assume for the ease of our intuitive explanations that $B_{f+l}(\mathcal{G},s)$ (resp. $B_f(\mathcal{G},s)$) is the largest ball explored by the agent at the end of `GlobalExpansion`(l,m) in the case where the returned value is true (resp. false).

Let us consider a phase i of our algorithm and, in order not to burden the text with a lot of “unless the treasure is found”, let us assume that the treasure will not be found by the end of it. Phase i is made of at most three successive attempts, each of them aiming at fulfilling at least one of the three conditions described earlier, with the help of our building block. In the first attempt, the agent executes `GlobalExpansion`$(1,\perp)$ from node s, the cost of which is $\mathcal{O}(e(f_i+1))$. At the end of this execution, the agent is at node s and $B_{f_i+1}(\mathcal{G},s)$ has been entirely explored by the agent. If $e(f_i+1) \geq 2e(f_i)$ or $f_i \leq 1$, the first attempt is a success as Condition 1 or Condition 2 is verified, and the agent directly switches to phase $i+1$. Otherwise, the attempt is a failure, but we can nonetheless observe that the cost incurred because of the attempt is just $\mathcal{O}(e(f_i))$ because $e(f_i+1) < 2e(f_i)$.

If the first attempt has failed, the agent starts the second attempt of phase i that consists of an execution of function `GlobalExpansion`$(f_i - 1, e(f_i))$. The hope here is to expand by a distance of $f_i - 1$ the radius of the largest explored ball, which is $B_{f_i+1}(\mathcal{G}, s)$. According to the properties of `GlobalExpansion` and the fact that $e(f_i + 1) < 2e(f_i)$, the cost of this execution, and thus of the second attempt, is $\mathcal{O}(e(f_i))$. If `GlobalExpansion`$(f_i - 1, e(f_i))$ returns true, then at the end of the second attempt, the radius of the largest explored ball is $2f_i$. Hence, the cost of the first two attempts being equal to $\mathcal{O}(e(f_i))$ and f_i being at least 2, Condition 2 is satisfied and the agent starts phase $i + 1$ without making the third attempt.

On the other hand, if `GlobalExpansion`$(f_i - 1, e(f_i))$ returns false, it is a different story. Indeed, the largest explored ball is still only $B_{f_i+1}(\mathcal{G}, s)$ and we cannot ensure the fulfillment of Condition 1 or Condition 2. This is exactly where Condition 3 comes into the picture. In order to remedy the failures of the two previous attempts, the agent will start a third and last attempt which consists of a dichotomic process that is described in Algorithm 1. At the end of this process, Condition 3 is guaranteed to be satisfied.

Algorithm 1 Third attempt.

```
1 $floor := f_i + 1$; $ceil := 3f_i - 2$; $l := \lfloor \frac{ceil - floor}{2} \rfloor$;
2 while $l \geq 1$ and $|B_{floor}(\mathcal{G}, s)| < 2e(f_i)$ do
3     $success :=$ GlobalExpansion$(l, e(f_i))$;
4     if $success = true$ then
5         $floor := floor + l$; $l := \lfloor \frac{ceil - floor}{2} \rfloor$;
6     else
7         $ceil := floor + 2l - 1$; $l := \lfloor \frac{l}{2} \rfloor$;
```

In order to better understand why we can get such a guarantee, let us take a look at the properties that are satisfied during the third attempt and at its end.

Since the execution of `GlobalExpansion`$(f_i - 1, e(f_i))$ returned false, the agent has explored at least $e(f_i)$ distinct edges outside of ball $B_{f_i+1}(\mathcal{G}, s)$ during the second attempt. Moreover, during this execution, the agent was always in $B_{3f_i-2}(\mathcal{G}, s)$ according to the properties of `GlobalExpansion`. As a result, in view of line 1 of Algorithm 1, we necessarily have the following feature before the execution of the while loop of Algorithm 1: $B_{floor}(\mathcal{G}, s)$ is the largest explored ball and $e(ceil) \geq 2e(f_i)$. Actually, by carefully examining the pseudocode of the while loop and using again the properties of `GlobalExpansion`, it can be inductively proven that this feature is a loop invariant. Alone, this loop invariant is not enough to bring the sought guarantee, but as highlighted below, it is of precious help to do the job.

The number of iterations of the while loop can be shown to be $\mathcal{O}(\log f_i)$. Furthermore, at the beginning of each iteration, $B_{floor}(\mathcal{G}, s)$ has size smaller than $2e(f_i)$ in view of the condition of the while loop, and is the largest explored ball in view of the loop invariant. Hence, according to the cost property of `GlobalExpansion`, each execution of `GlobalExpansion`$(l, e(f_i))$ costs at most $\mathcal{O}(e(f_i))$ like the previous two attempts, which gives a total cost of $\mathcal{O}(e(f_i) \log f_i)$ of the whole phase. This corresponds exactly to the target value of Condition 3. Along with this, at the end of the while loop, the size of $B_{floor}(\mathcal{G}, s)$ is at least $2e(f_i)$, or $l < 1$. In the first case, we immediately have $e(floor + 1) \geq 2e(f_i)$, while in the second case it can be shown that $ceil \leq floor + 1$. This, combined with the fact that $e(ceil)$ is always at least $2e(f_i)$ (by the loop invariant) and the fact that $floor$ is always at least $f_i + 1$, allows us to show the last missing piece of the puzzle, which is precisely this: when Algorithm 1 terminates, ball $B_{f_i+k}(\mathcal{G}, s)$ is entirely explored and $e(f_i + k + 1) \geq 2e(f_i)$ for some integer $k \geq 1$.

To conclude with the intuitive explanations, let us give, as promised, some more insight concerning the building block `GlobalExpansion`(l, m). At first glance, one might think that `GlobalExpansion` could be directly derived from the exploration algorithm `CFX`(v, r, α) of [11], which permits to explore a ball $B_r(\mathcal{G}, v)$ at a cost of $\mathcal{O}\left(\frac{|B_{(1+\alpha)r}(\mathcal{G},v)|}{\alpha}\right)$ for any given real $\alpha > 0$ (this corresponds to a cost of $\mathcal{O}\left(\frac{e((1+\alpha)r)}{\alpha}\right)$ when $v = s$) provided $\alpha r \geq 1$. Indeed, the task of `GlobalExpansion`(l, m) that consists in expanding the radius f of the largest explored ball by a distance l in the case where m is appropriately set, can be done with `CFX`$(s, f + l, \alpha)$. However, in this case we want the cost of this expansion to be $\mathcal{O}(e(f + 2l - 1))$, which is an important property of our strategy. This cannot be guaranteed using `CFX`$(s, f + l, \alpha)$ because, in order to get a cost depending on $e(f + 2l - 1)$, we would have to set α to a value lower than $\frac{l-1}{f+l}$, which cannot lead to a cost that is linear in $e(f + 2l - 1)$, as $\frac{l-1}{f+l}$ can be arbitrarily small. True, during the design we could have been "less demanding" about some of the properties of `GlobalExpansion`(l, m), but not significantly enough to permit the use of `CFX`$(s, f + l, \alpha)$ without spoiling the validity or the cost complexity of our strategy. Another solution that may come to mind would be to apply `CFX`(v, l, α) from each node v located on the boundary of the largest explored ball $B_f(\mathcal{G}, s)$. Visiting each node of the boundary can be done in $\mathcal{O}(e(f))$. Hence, this solution looks attractive because by setting α to $\frac{1}{2}$ or less (which overcomes the above problem of the arbitrarily small value) and provided the zones explored by the different executions of `CFX` do not overlap, we would get a cost that is linear in $e(f + 2l - 1)$. The bad news is that there may be overlaps. Of course, some overlaps can be easily avoided, especially those appearing within $B_f(\mathcal{G}, v)$, but some others cannot without running the risk of missing some nodes of $B_{f+l}(\mathcal{G}, s)$ that are outside of $B_f(\mathcal{G}, s)$. These "necessary overlaps" may be pernicious and may occur in a way that prevents us from guaranteeing a cost of $\mathcal{O}(e(f + 2l - 1))$.

So, what did we do? Although it was not possible to use `CFX` as a black box, we managed to tailor `GlobalExpansion` by adapting to our needs an elegant algorithmic technique used in `CFX`. Through a set of judiciously pruned trees spanning some already explored area, it allowed us to satisfy the desired cost property of `GlobalExpansion` by controlling and amortizing efficiently the number of times the same edges are traversed. The technique in question is detailed in the next section that presents the pseudocode of our treasure hunt algorithm.

4 Algorithm

Solving the treasure hunt problem in the unrestricted model can be done by executing Algorithm `TreasureHunt`(x) described below in Algorithm 2 and by interrupting it as soon as the treasure is found.

Algorithm 2 `TreasureHunt`(x).

```
v := the current node;
$\mathcal{M}$ := ({v}, ∅); /* $\mathcal{M}$ is a global variable     */
repeat
    Search(x);
```

The input parameter x is a positive real constant. It is a technical ingredient that will have an impact on the maximal distance at which the agent can be from node s. In our present context, parameter x does not really matter and it can be fixed as *any* positive real constant. In fact, it will show its full significance in Section 6 that is dedicated to the same problem in restricted models: there, we will reuse `TreasureHunt`(x) in a context where x will have to be carefully chosen. The variable $\mathcal{M}$ in line 2 of Algorithm 2 is a global variable that will always correspond to some explored subgraph of $\mathcal{G}$. For this reason, it will recurrently appear in most of the pseudocodes of the functions described thereafter.

As the reader can see, the execution of Algorithm `TreasureHunt`(x) essentially consists of a series of executions of procedure `Search`(x), whose pseudocode is described in Algorithm 3: these executions correspond to what we called "phases" in our intuitive explanations of Section 3. Procedure `Search`(x) should be seen as the organizer of our solution. At the beginning of each call to `Search`(x), $\mathcal{M}$ is some explored ball $B_f(\mathcal{G}, s)$ and the goal of the call is to make this ball grow while satisfying some conditions. These conditions, whose simplified version we gave at the beginning of Section 3, are formally described in Lemma 4.

Algorithm 3 `Search`(x).

```
1  $v :=$ the current node; $m := |\mathcal{M}|$;
2  $floor := \epsilon_{\mathcal{M}}(v)$; $ceil := \lfloor (1+x) \cdot floor \rfloor$;
3  $success :=$ GlobalExpansion$(1, \perp)$;
4  $floor := floor + 1$; $i := 0$; $l := \lfloor \frac{ceil - floor}{2} \rfloor$;
5  while $l \geq 1$ and $|\mathcal{M}| < 2m$ and ($i \neq 1$ or $success = false$) do
6      $success :=$ GlobalExpansion$(l, m)$;
7      if $success = true$ then
8          $floor := floor + l$; $l := \lfloor \frac{ceil - floor}{2} \rfloor$;
9      else
10         $ceil := floor + 2l - 1$; $l := \lfloor \frac{l}{2} \rfloor$;
11     $\mathcal{M} := B_{floor}(\mathcal{M}, v)$;
12     $i := i + 1$;
```

Although there are some technical differences, we can discern, throughout the lines of Algorithm 3, the three attempts outlined in Section 3 that rely on function `GlobalExpansion`. Roughly speaking, line 3 of Algorithm 3 relates to the first attempt, the first iteration of the while loop of Algorithm 3 relates to the second attempt, and the other iterations relate to the third attempt.

The pseudocode of function `GlobalExpansion`(l, m) is given by Algorithm 4. It has primarily the same specifications as those given in Section 3 except that we did not implement the case where $m = \perp$ and $l \geq 2$ as it was not necessary for our purpose. Hence, the function precisely handles the case where $l = 1$ and $m = \perp$, and the case where $l \geq 1$ and $m \neq \perp$. The general scheme of the function is as follows. At the beginning, the agent knows a ball $B_f(\mathcal{G}, s)$ that is stored in variable $\mathcal{M}$ and the objective is to expand the radius of this ball by a distance l, without exploring more than m edges outside of $B_f(\mathcal{G}, s)$, if $m \neq \perp$. To do this, the agent visits the nodes $L[1], L[2], \ldots$ (stored in the array L) of the boundary of $B_f(\mathcal{G}, s)$ and executes from these nodes function `CDFS` (described in Algorithm 5 and whose name stands for Constrained DFS) or function `LocalExpansion` (described in Algorithm 6) depending on the initial values of l and m. Each of these executions, which starts and ends at the same node, locally contributes to the global expansion of the ball. In the case where

$m \neq \perp$, variable b of Algorithm 4 is updated with the return value of the two aforementioned functions, and corresponds at each stage to the remaining number of new edges the agent is authorized to traverse outside of $B_f(\mathcal{G}, s)$. If b becomes negative before the end of the while loop of Algorithm 4, the objective of expansion is simply not reached. Note that, in order to avoid that the moves from one node of the boundary of $B_f(\mathcal{G}, s)$ to the next get too costly, they are made according to a precise order that results from the definition of L given in line 2 of Algorithm 4.

Algorithm 4 `GlobalExpansion`(l, m).

```
1  $v :=$ the current node;
2  $L :=$ the array containing all the nodes of the boundary of $\mathcal{M}$ sorted in the order of
     the first visit through the DFS traversal of $\mathcal{M}$ from node $v$;
3  $T :=$ the tree produced by the DFS traversal of $\mathcal{M}$ from node $v$;
4  $i := 1$; $b := m$; $\mathcal{T} := \emptyset$; /* $\mathcal{T}$ is a global variable */
5  while $i \leq |L|$ and ($b \geq 0$ or $b = \perp$) do
6      MoveTo($T, L[i]$);
7      if $l = 1$ then
8          if $b = \perp$ then
               /* We run CDFS($1, deg(L[i])$) without using its return value. */
9              $(*, *) :=$CDFS($1, deg(L[i])$);
10         else
               /* We run CDFS($1, b$) without using the second term of its
                  return value. */
11             $(b, *) :=$CDFS($1, b$);
12     else
13         $b :=$ LocalExpansion($l, b$);
14     $i := i + 1$;
15 MoveTo($T, v$);
16 return the logical value of "$b \geq 0$ or $b = \perp$";
```

As one can see in lines 9 and 11 of Algorithm 4, the implementation of the case $l = 1$ in Algorithm 4 directly relies on function `CDFS`. We will see below that this function is also involved in the trickier case where $l \geq 2$ and $m \neq \perp$ through the calls to function `LocalExpansion`. Function `CDFS`(l, b) permits the agent to perform a depth-first search in the zone that does not belong to $\mathcal{M}$ when it starts executing it. During the execution of this function $\mathcal{M}$ grows, augmented with the edges that are traversed by the agent. The two input parameters $l \geq 1$ and $b \geq 0$ are integers that bring constraints to the execution of the depth-first search. The first indicates the limit depth of the search, while the second indicates an upper bound on the number of distinct edges the agent can traverse during the search: when this bound is violated, the agent stops the search and goes back to the node it occupied at the beginning of the search. The return value of `CDFS`(l, b) is a couple (n, T). The first term n is an integer such that $b - n$ is the number of distinct edges that have been traversed during the execution of `CDFS`(l, b). If the bound b has been respected then $n \geq 0$, otherwise $n = -1$. Concerning the second term T of the return value, it simply corresponds to the resulting DFS tree of the execution of `CDFS`(l, b). If $n \geq 0$ and v is the occupied node at the start of `CDFS`(l, b), then for every node u such that $d_T(u, v) < l$, u is complete in $\mathcal{M}$ at the end of `CDFS`(l, b). Note that in the particular case where $l = 1$ and $m = \perp$ in Algorithm 4, the second argument of each call to CDFS is always set to the degree of the node from which the function is executed (cf. line 9 of Algorithm 4) in order to ensure that this node becomes complete in $\mathcal{M}$ at the end of the call.

Algorithm 5 CDFS(l, b).

```
v := the current node; T := ({v}, ∅); bound := b;
if l > 0 then
    Mark node v;
    while node v is incomplete in M and bound ≥ 0 do
        pt_1 := the smallest free port at node v in M;
        Take port pt_1;
        w := the current node;
        pt_2 := the port by which the agent has just entered node w;
        if v < w then
            K := ({v, w}, {(v, w, pt_1, pt_2)});
        else
            K := ({v, w}, {(w, v, pt_2, pt_1)});
        M := M ⊔ K; bound := bound − 1;
        if w is not marked then
            (bound, T') := CDFS(l − 1, bound);
            T := T ⊔ T' ⊔ K;
        Take port pt_2;
    Unmark node v;
return (bound, T);
```

The case where $l \geq 2$ and $m \neq \perp$ in Algorithm 4 relies on function LocalExpansion. It is exactly here that we make use of the algorithmic technique of [11] mentioned at the end of Section 3, which is based on a set of adequately pruned trees. In our solution, this set corresponds to the variable $\mathcal{T}$. It is a global variable like $\mathcal{M}$ and it is initialized to $\emptyset$ at the beginning of each call to GlobalExpansion (cf. line 4 of Algorithm 4). Let us consider the ith call LE_i to LocalExpansion(l, b) made from node $L[i]$ during an execution of GlobalExpansion(l, m). At the end of LE_i, the return value of LocalExpansion(l, b) is an integer $n \geq -1$ such that $b - n$ is the number of distinct edges that have been traversed during LE_i and that were not in $\mathcal{M}$ at the start of LE_i. Besides, in the case where $n \geq 0$, at the end of LE_i we can guarantee that for each incomplete node u of $\mathcal{M}$, $d_{\mathcal{M}}(L[i], u) > l$ or u is one of the last $|L| - i$ nodes of L (i.e., a node of L from which the agent has not yet executed LocalExpansion(l, b)).

To see the algorithmic technique in question at work, let us focus on an iteration I of the first while loop of Algorithm 6 occuring in LE_i. This iteration starts at node $L[i]$ and we can show that at the beginning of I, we necessarily have the following properties.

- $\mathcal{T}$ is a set of node disjoint trees that are all subgraphs of $\mathcal{M}$.
- For each tree Tr of $\mathcal{T}$, $|Tr| \geq \lfloor \frac{l}{8} \rfloor$ if Tr contains a node different from $L[i]$.
- Every incomplete node of $\mathcal{M}$ belongs to a tree of $\mathcal{T}$ or is one of the last $|L| - i$ nodes of L.

Algorithm 6 LocalExpansion(l, b).

```
1  $bound := b$; $v :=$ the current node;
2  if $v$ is incomplete in $\mathcal{M}$ and no tree of $\mathcal{T}$ contains node $v$ then
3      $\mathcal{T} := \mathcal{T} \cup \{(\{v\}, \emptyset)\}$;
4  while IncompleteNodes($v, \mathcal{M}, l$) $\cap$ Nodes($\mathcal{T}$) $\neq \emptyset$ and $bound \geq 0$ do
5      $u :=$ the node with the smallest label in IncompleteNodes($v, \mathcal{M}, l$) $\cap$ Nodes($\mathcal{T}$);
6      MoveTo($\mathcal{M}, u$);
7      Prune($l$);
8      $bound :=$ Explore($l, bound$);
9      Remove from $\mathcal{T}$ every tree for which all the nodes are complete in $\mathcal{M}$;
10     while there are two trees $T$ and $T'$ in $\mathcal{T}$ having a common node do
11         $T'' :=$ the spanning tree produced by the BFS traversal of $T \sqcup T'$ from the
             node having the smallest label in $T \sqcup T'$;
12         $\mathcal{T} := (\mathcal{T} \setminus \{T, T'\}) \cup \{T''\}$;
13     Execute in the reverse order all the edge traversals that have been made since the
         beginning of the current iteration of the while loop;
14 return $bound$;
```

Let us examine what happens during iteration I. At the beginning of I, the agent follows a path of length at most l from node $L[i]$ to a node u that is incomplete in $\mathcal{M}$ (cf. line 5 of Algorithm 6). By the first and third properties and the condition at line 4 of Algorithm 6, node u belongs to a unique tree $T_u \subseteq \mathcal{G}$ of $\mathcal{T}$. Once the agent occupies node u, the tree T_u is pruned via the procedure Prune(l) at line 7 of Algorithm 6. The pseudocode of procedure Prune is detailed in Algorithm 7.

Algorithm 7 Prune(l).

```
1  $v :=$ the current node;
2  $T_v :=$ the tree of $\mathcal{T}$ containing node $v$;
3  $\mathcal{T} := \mathcal{T} \setminus \{T_v\}$;
4  Root $T_v$ at node $v$;
5  foreach node $u$ of $T_v$ such that $d_{T_v}(u, v) = \max\{1, \lfloor \frac{l}{4} \rfloor\}$ do
6      $T_u :=$ the subtree of $T_v$ rooted at $u$;
7      if $\epsilon_{T_u}(u) \geq \lfloor \frac{l}{4} \rfloor - 1$ then
8          $\mathcal{T} := \mathcal{T} \cup \{T_u\}$;
9          Remove from $T_v$ all nodes that belong to $T_u$ and all edges that are incident to
             a node of $T_u$;
10 $\mathcal{T} := \mathcal{T} \cup \{T_v\}$;
```

In the context of iteration I, the pruning operation will transform T_u into a tree T'_u such that $\epsilon_{T'_u}(u) \leq \lfloor \frac{l}{2} \rfloor - 1$, while preserving the three properties listed above: this offers two important advantages to which we will return at the end of this section. Once the pruning is done, the agent applies function Explore($l, bound$), whose pseudocode is given in Algorithm 8.

Algorithm 8 `Explore`(l, b).

```
bound := b; i := 1; v := the current node;
T := the tree of $\mathcal{T}$ containing node v;
V := array containing all the nodes of T sorted in the order of the first visit through
  the DFS traversal of T from node v;
while $i \leq |V|$ and $bound \geq 0$ do
    MoveTo$(T, V[i])$;
    if node $V[i]$ is incomplete in $\mathcal{M}$ then
        $(bound, T')$ := CDFS$(\lfloor \frac{l}{2} \rfloor, bound)$;
        $\mathcal{T} := \mathcal{T} \cup \{T'\}$;
return bound;
```

In the pseudocodes of `LocalExpansion` and of `Explore`, variable *bound* corresponds at any stage to the number of remaining edges the agent is authorized to traverse outside of $B_f(\mathcal{G}, s)$. In the context of iteration I, function `Explore`$(l, bound)$ permits the agent to explore tree T'_u and to execute function `CDFS`$(\lfloor \frac{l}{2} \rfloor, bound)$ from the nodes of T'_u that are incomplete in $\mathcal{M}$, as long as variable *bound* remains non-negative. These executions of `CDFS` occuring during the exploration of T'_u create in turn trees that are added to $\mathcal{T}$ (cf. line 8 of Algorithm 8) and that contain the new incomplete nodes of $\mathcal{M}$. If the return value of function `Explore`$(l, bound)$ is non-negative, we can show that all the nodes of T'_u have become complete in $\mathcal{M}$. Under the same condition, we will also guarantee that each tree Tr, which has been added to $\mathcal{T}$ during the execution of function `Explore`, contains an incomplete node only if $|Tr| \geq \lfloor \frac{l}{8} \rfloor$. Both these guarantees combined with lines 9 to 12 of Algorithm 6 will allow us to show that our three properties will be satisfied for the next iteration I', if any, even if it occurs in another call to `LocalExpansion` (in the same execution of `GlobalExpansion`(l, m)). In particular, this is made possible by the fact that $\mathcal{T}$ is never reset between the calls to `LocalExpansion` during the execution of the while loop of Algorithm 4.

To fully appreciate the process accomplished during I, we need to come back to the two aforementioned advantages that are brought by the pruning operation. The first advantage concerns the height of T'_u. The fact that $\epsilon_{T'_u}(u) \leq \lfloor \frac{l}{2} \rfloor - 1$ is a key element to control the maximal distance between the agent and node s. Without this, the agent could go too far from node s and we would not be able to guarantee that the agent explores only edges of $B_{f+2l-1}(\mathcal{G}, s)$ during the execution of `GlobalExpansion`(l, m) (which is a crucial property as pointed out in Section 3). The second advantage concerns the size of T'_u. The pruning operation preserves the second property, and thus (1) T'_u corresponds to a tree containing only node $L[i]$ or (2) $|T'_u| \geq \lfloor \frac{l}{8} \rfloor$. This implies that the cost resulting from the moves of line 6 of Algorithm 6 and line 5 of Algorithm 8 is linear in the size of T'_u. Besides, if *bound* is still non-negative at the end of `Explore`$(l, bound)$, all the nodes of T'_u have become complete (it is in particular the case for node u) and the tree is removed from $\mathcal{T}$ through line 9 of Algorithm 6. After this removal, no edge of T'_u will be an edge of another tree of $\mathcal{T}$ till the end of the execution of `GlobalExpansion`(l, m). As a result, if the return value of `Explore`$(l, bound)$ is non-negative in I, we can associate the moves of line 6 of Algorithm 6 and line 5 of Algorithm 8 to at least one node that becomes complete during I and to at least $\lfloor \frac{l}{8} \rfloor$ edges that will no longer be edges of any tree of $\mathcal{T}$ till the end of the execution of `GlobalExpansion`(l, m). In our analysis, this association will enable us to amortize efficiently the number of times the agent retraverses the edges that have been already explored during any previous iteration of the considered while loop. This will be a decisive argument to show the cost of $\mathcal{O}(e(f) + m)$ for the execution of `GlobalExpansion`(l, m) in the case where $l \geq 2$ and $m \neq \perp$.

5 Correctness and complexity analysis

In this section, we give a sketch of the proof of correctness and of complexity of Algorithm `TreasureHunt`(x) in the unrestricted model. `TreasureHunt`(x) is an exploration algorithm that can be executed also if there is no treasure in $\mathcal{G}$. We first establish several exploration properties of our algorithm or of its components assuming that there is no treasure in $\mathcal{G}$. In fact, this assumption concerns all the lemmas (and only them) of this section. After the series of lemmas, we show the main result of this section, namely Theorem 6, which specifies that our algorithm allows to find the treasure at a cost quasi-linear in $e(d)$.

Throughout the proof of correctness, we will often have to consider the value of the global variable $\mathcal{M}$ before or after some executions. To this end, we introduce the following convention: given an execution $\mathcal{E}$ of Algorithm `TreasureHunt`(x) or some part of it, we denote by $M_1(\mathcal{E})$ the value of $\mathcal{M}$ at the beginning of $\mathcal{E}$ and by $M_2(\mathcal{E})$ the value of $\mathcal{M}$ at the end of $\mathcal{E}$.

We start by giving two lemmas concerning the function `CDFS`(l, b). They list some properties that are useful to prove Lemma 3. They are direct consequences of Algorithm 5 and can be easily proved by induction on l.

▶ **Lemma 1.** *Consider an execution $\mathcal{E}$ of function* `CDFS`(l, b) *from a node u of $\mathcal{G}$ where $l \geq 1$ and $b \geq 0$ are integers. Assume that $M_1(\mathcal{E}) \subseteq \mathcal{G}$. Execution $\mathcal{E}$ terminates at node u, and the agent always knows a path of length at most l from node u to its current node during $\mathcal{E}$.*

▶ **Lemma 2.** *Consider an execution $\mathcal{E}$ of function* `CDFS`(l, b) *from a node u of $\mathcal{G}$ where $l \geq 1$ and $b \geq 0$ are integers. Assume that $M_1(\mathcal{E}) \subseteq \mathcal{G}$. Function* `CDFS`$(l, b)$ *returns a couple (i, Tr) such that the following properties are satisfied.*

- *Let G be the subgraph of $\mathcal{G}$ that has been explored during $\mathcal{E}$. $G \subseteq B_l(\mathcal{G}, u)$, $|M_1(\mathcal{E}) \sqcap G| = 0$, $M_1(\mathcal{E}) \sqcup G = M_2(\mathcal{E})$, Tr is a spanning tree of G and $i = b - |G| \geq -1$.*
- *The cost of $\mathcal{E}$ is $2|G|$ and $\epsilon_{Tr}(u) \leq l$.*
- *If $i \geq 0$ then for every node v of Tr such that $d_{Tr}(u, v) < l$, v is complete in $M_2(\mathcal{E})$. If $i = -1$, then there exists a node v of Tr such that $d_{Tr}(u, v) \leq l - 1$ and v is incomplete in $M_2(\mathcal{E})$.*

The following lemma establishes the properties of function `GlobalExpansion`(l, m). It is prerequisite to prove Lemma 4 that concerns procedure `Search`(x).

▶ **Lemma 3.** *Consider an execution $\mathcal{E}$ of function* `GlobalExpansion`(l, m) *from the source node s, where l is a positive integer and m is either a positive integer or $\perp$. Assume that $M_1(\mathcal{E}) = B_f(\mathcal{G}, s)$ for some integer $f \geq 0$.*

- *if $m \neq \perp$, or $m = \perp$ and $l = 1$, then $\mathcal{E}$ terminates at node s and during $\mathcal{E}$ the agent always knows a path in $\mathcal{G}$ of length at most $f + 2l - 1$ from node s to its current node.*
- *If $m = \perp$ and $l = 1$ then the cost of $\mathcal{E}$ is $\mathcal{O}(e(f + 1))$ and $B_{f+1}(\mathcal{G}, s) = M_2(\mathcal{E})$.*
- *If $m \neq \perp$ and function* `GlobalExpansion`(l, m) *returns true (resp. false) then $B_{f+l}(\mathcal{G}, s) \subseteq M_2(\mathcal{E})$ (resp. $B_f(\mathcal{G}, s) \subseteq M_2(\mathcal{E})$ and $e(f + 2l - 1) > e(f) + m$) and the cost of $\mathcal{E}$ is $\mathcal{O}(e(f) + m)$.*

Below is the lemma establishing the properties of procedure `Search`(x).

▶ **Lemma 4.** *Consider an execution $\mathcal{E}$ of procedure* `Search`(x) *from the source node s, for any real constant $x > 0$. Assume that $M_1(\mathcal{E}) = B_f(\mathcal{G}, s)$ for some integer $f \geq 0$.*

- *The execution terminates at node s and during the execution the agent always knows a path in $\mathcal{G}$ of length at most $\max\{f + 1, \lfloor (1 + x)f \rfloor\}$ from node s to its current node.*
- *There exists an integer $f' > f$ such that $M_2(\mathcal{E}) = B_{f'}(\mathcal{G}, s)$ and at least one of the following properties holds:*

1. *The cost of $\mathcal{E}$ is $\mathcal{O}(e(f+1))$ and $xf < 3$.*
2. *The cost of $\mathcal{E}$ is $\mathcal{O}(e(f))$ and $f' > (1+\frac{x}{3})f$.*
3. *The cost of $\mathcal{E}$ is $\mathcal{O}(e(f)\log(f+2))$ and $e(f'+1) \geq 2e(f)$.*
4. *The cost of $\mathcal{E}$ is $\mathcal{O}(e(f+1))$ and $e(f+1) \geq 2e(f)$.*

If we put aside the initial assignments of lines 1 and 2 in Algorithm 2, the execution of procedure `TreasureHunt`(x) from the source node s in $\mathcal{G}$ can be viewed as a sequence of consecutive executions of procedure `Search`(x): the ith execution of `Search`(x) in this sequence will be denoted by S_i.

The following lemma is a small technical observation concerning the execution of `TreasureHunt`(x) from the source node s. Since, at the beginning of this execution, variable $\mathcal{M}$ is equal to $B_0(\mathcal{G}, s)$, the lemma can be easily proved by induction on i using Lemma 4.

▶ **Lemma 5.** *Consider an execution of procedure* `TreasureHunt`(x) *from the source node s, for any real constant $x > 0$. For every integer $i \geq 1$, S_i starts and ends at node s, and there are two integers $f_{i+1} > f_i \geq i-1$ such that $M_1(S_i) = B_{f_i}(\mathcal{G}, s)$ and $M_2(S_i) = B_{f_{i+1}}(\mathcal{G}, s)$.*

Using Lemmas 4 and 5, we can prove the main result of this section that is stated in the following theorem.

▶ **Theorem 6.** *Consider a graph $\mathcal{G}$ of unknown radius r in which a treasure is located at an unknown distance at most $1 < d \leq r$ from the starting node s of an agent. For any real constant $x > 0$, procedure* `TreasureHunt`(x) *allows the agent to find the treasure at cost $\mathcal{O}(e(d)\log d)$.*

6 Treasure hunt with restrictions

Theorem 6 holds for the task of treasure hunt without any restrictions on the moves of the agent, for all locally finite graphs, both finite and infinite. In this section we show how to modify our treasure hunt algorithm to make it work under the fuel-restricted and the rope-restricted models for finite graphs.

Strictly speaking, the fuel-restricted model was defined in [3] assuming that both the constant $\alpha > 0$ and the radius r were known to the agent. On the other hand, the rope-restricted model was defined in [11] for any known constant $\alpha > 0$ and for unknown radius r. We will show that, for each of these restrictive models and for any known constant $\alpha > 0$, we can design a treasure hunt algorithm with the promised efficiency even when r is unknown. To this end, we need to modify the restriction of the fuel-restricted model from [3], avoiding to reveal r to the agent by showing it the size of the tank. We fix a positive constant α, known to the agent, and we proceed as follows. For the restricted tank case from [3], we assume that at any visit of s the agent can put as much fuel in the tank as it wants, but we show that if the (unknown) radius of the graph is r then the tank is never filled to more than $B = 2(1+\alpha)r$. The formalization of the rope-restricted model corresponds to its definition in [11]. Recall that the agent is attached at s by an infinitely extendible rope that it unwinds by a length 1 with every forward edge traversal and rewinds by a length of 1 with every backward edge traversal. Whenever the agent completely backtracks to s, the unwinded segment of the rope is of length 0. We show that if the (unknown) radius of the graph is r then the initial segment of the rope unwinded by the agent executing our algorithm will never be longer than $L = (1+\alpha)r$.

▶ **Theorem 7.** *Consider a graph $\mathcal{G}$ of unknown radius r in which a treasure is located at an unknown distance at most $1 < d \leq r$ from the starting node s of the agent. For any positive constant α, procedure* `TreasureHunt`$(\frac{\alpha}{2})$ *can be transformed into a procedure allowing the agent to find the treasure at cost $\mathcal{O}(e(d) \log d)$ in the rope-restricted model (resp. fuel-restricted model) without ever using a segment of the rope longer than $(1+\alpha)r$ (resp. without filling the tank to more than $2(1+\alpha)r$ at any visit of s).*

Proof. The execution of procedure `TreasureHunt`$(\frac{\alpha}{2})$ from node s corresponds to a sequence $S = (S_1, S_2, \dots, S_{|S|})$ of executions of `Search`$(\frac{\alpha}{2})$, in which the $|S|$th execution of `Search`$(\frac{\alpha}{2})$ is interrupted prematurely because of the discovery of the treasure.

We denote by G_0 the graph consisting only of node s, and for every $1 \leq i \leq |S|$, we denote by G_i the subgraph of $\mathcal{G}$ that has been explored from the beginning of S_1 to the end of S_i. For every $1 \leq i \leq |S|$, the cost of S_i will be denoted by c_i.

According to Lemma 5, for every $1 \leq i \leq |S|$, S_i starts and ends at node s (except $S_{|S|}$ that ends at the node containing the treasure), there is an integer $f_i \geq 0$ such that $M_1(S_i) = B_{f_i}(\mathcal{G}, s)$ and if $i < |S|$, $M_2(S_i) = M_1(S_{i+1})$. Moreover, the value of $\mathcal{M}$ is always a subgraph of $\mathcal{G}$ whose nodes and edges have been all explored by the agent, and thus, for every $1 \leq i \leq |S|$, $B_{f_i}(\mathcal{G}, s)$ is a subgraph of G_{i-1}, f_i is unique and $f_i < d$ (or otherwise the treasure would have been found before the start of S_i which leads to a contradiction with the existence of this execution). Hence, from the fact that $d \leq r$, we get the following claim.

▷ Claim 8. For every $1 \leq i \leq |S|$, $\max\{f_i + 1, \lfloor (1+\alpha) f_i \rfloor\} \leq (1+\alpha) r$

First, we describe a new algorithm $\mathcal{A}$ that permits to find the treasure, in the model without constraints, with asymptotically the same cost as that of `TreasureHunt`$(\frac{\alpha}{2})$. This new algorithm consists in executing `TreasureHunt`$(\frac{\alpha}{2})$ with some changes in order to guarantee an extra property that will be important for our purpose. More precisely, an execution of $\mathcal{A}$ from node s is a sequence of executions $(S'_1, S'_2, \dots, S'_{|S|})$ in which each S'_i has cost $\mathcal{O}(c_i)$ and corresponds to an emulation of execution S_i. In particular, for every $1 \leq i \leq |S|$, S'_i starts and ends at node s (except $S'_{|S|}$ that ends at the node containing the treasure), $M_1(S'_i) = M_1(S_i) = B_{f_i}(\mathcal{G}, s)$, and at the end of S'_i, G_i has been entirely explored. Obviously, all of this would not be interesting without the additional crucial property brought by S'_i that will be called the *frequent return property* and that is the following. Let Sk be the stack initially empty in which we push (resp. pop) the last traversed edge if it corresponds to a forward (resp. backward) edge traversal. During S'_i, the size of Sk is 0 at least once during any block of $2\max\{f_i + 1, \lfloor (1+\alpha) f_i \rfloor\}$ consecutive edge traversals, and is never greater than $\max\{f_i + 1, \lfloor (1+\alpha) f_i \rfloor\}$. Moreover, at the beginning of S'_i, the size of Sk is 0, and if $i < |S|$, it is also 0 at the end of S'_i.

Note that Algorithm $\mathcal{A}$ is a solution with the desired cost in the rope-restricted model, that will never use a segment of the rope longer than $(1+\alpha)r$, as for all $1 \leq i \leq |S|$, we have $\max\{f_i + 1, \lfloor (1+\alpha) f_i \rfloor\} \leq (1+\alpha) r$ according to Claim 8. By requiring the agent, each time the size of Sk is 0 in S'_i, to refuel its tank up to the limit of $2 \cdot \max\{f_i + 1, \lfloor (1+\alpha) f_i \rfloor\}$ (when the size of Sk is 0, the agent is at node s), we also get our objective with algorithm $\mathcal{A}$ in the fuel-restricted model, as the agent never runs out of fuel and $2 \cdot \max\{f_i + 1, \lfloor (1+\alpha) f_i \rfloor\} \leq 2(1+\alpha) r$.

Let us describe how we can construct our emulations while ensuring the features mentioned above. Consider the emulation S'_i of S_i. Assume that at the beginning of S'_i, G_{i-1} has been entirely explored, the size of Sk is 0 and $M_1(S'_i) = M_1(S_i) = B_{f_i}(\mathcal{G}, s)$. These assumptions are trivially satisfied if $i = 1$. We will show below that, at the end of S'_i, G_i is entirely explored and if $i < |S|$ the size of Sk is 0. We will also show that if $i < |S|$ then $M_1(S'_{i+1}) = B_{f_{i+1}}(\mathcal{G}, s)$. We consider two cases.

The first case is when $\alpha f_i \geq 2$. We assume for simplicity that the number of edge traversals in S_i is a positive multiple of $\lfloor \frac{\alpha f_i}{2} \rfloor$. As we will explain in detail, in this case the agent executes S_i but interrupts it after each block of $\lfloor \frac{\alpha f_i}{2} \rfloor$ edge traversals, except the last one, to make a "return trip" to node s before resuming S_i from where it was interrupted. The goal of these return trips is to satisfy the frequent return property. Once the agent has executed all instructions of S_i, it is either at the node containing the treasure or at node s. In the first case, we know that $i = |S|$ and S'_i is simply over. In the second case $i < |S|$, but we do not have the guarantee that the size of Sk is 0. Hence, if the agent occupies node s once it has executed all instructions of S_i, it then finishes S'_i with what we call a *close period* in which it executes in the reverse order some of the last edge traversals so that the size of Sk becomes 0 at the end of S'_i.

Denote by v_k the node in which the kth interruption occurs, and by P_k the path of length at most $\lfloor (1 + \frac{\alpha}{2}) f_i \rfloor$ from node s to v_k that is known by the agent when the interruption occurs. Note that P_k necessarily exists in view of Lemma 4, of the initial assumptions concerning S'_i and of the fact that no edge traversal of S_i has been skipped before the kth interruption. Also note that if there are several paths that can play the role of P_k, we simply choose the lexicographically smallest shortest path among them.

Each interruption is composed of two parts. In the first interruption, the first part consists in backtracking to node s by executing in the reverse order the last $\lfloor \frac{\alpha f_i}{2} \rfloor$ edge traversals. The second part consists in going back to node v_1 using path P_1 to resume S_i. For the kth interruption with $k > 1$, the first part consists in backtracking to node s by executing in the reverse order the last $|P_{k-1}| + \lfloor \frac{\alpha f_i}{2} \rfloor$ edge traversals, and the second part consists in going back to node v_k using path P_k to resume S_i. Finally, the close period simply consists in backtracking to node s by executing in the reverse order the last $\lfloor \frac{\alpha f_i}{2} \rfloor$ edge traversals if S_i is made of only one block of $\lfloor \frac{\alpha f_i}{2} \rfloor$ edge traversals. Otherwise, it consists in backtracking to node s by executing in the reverse order the last $|P_{k^*-1}| + \lfloor \frac{\alpha f_i}{2} \rfloor$ edge traversals where $k^* = \frac{c_i}{\lfloor \frac{\alpha f_i}{2} \rfloor}$ is the number of blocks of $\lfloor \frac{\alpha f_i}{2} \rfloor$ edge traversals in S_i.

It follows by induction on the number of interruptions that the size of Sk is 0 at the end of the first part of each interruption. Using this, the fact that Sk is empty at the beginning of S'_i and the fact that for every $1 < k \leq \frac{c_i}{\lfloor \frac{\alpha f_i}{2} \rfloor}$, $|P_{k-1}| + \lfloor \frac{\alpha f_i}{2} \rfloor \leq \lfloor (1 + \alpha) f_i \rfloor$, it follows that the frequent return property is satisfied during S'_i.

Moreover, it follows from the above explanation that at the end of S'_i, G_i is entirely explored and the agent is at the node containing the treasure, if $i = |S|$. If $i < |S|$, it also follows that the size of Sk is 0 at the beginning of the next emulation S'_{i+1}, and $M_1(S'_{i+1}) = M_1(S_{i+1}) = B_{f_{i+1}}(\mathcal{G}, s)$ because $M_2(S'_i) = M_2(S_i) = M_1(S_{i+1})$. Finally, concerning the cost of S'_i observe that the number of interruptions is $\frac{c_i}{\lfloor \frac{\alpha f_i}{2} \rfloor} - 1$ and during each interruption as well as during the close period the agent makes at most $2\lfloor (1+\alpha) f_i \rfloor$ edge traversals. The cost of S'_i is then upper bounded by $c_i + \frac{c_i}{\lfloor \frac{\alpha f_i}{2} \rfloor} 2\lfloor (1+\alpha) f_i \rfloor \leq (1 + \frac{2(1+\alpha) f_i}{\lfloor \frac{\alpha f_i}{2} \rfloor}) c_i$. If $2 \leq \alpha f_i < 4$, then $\frac{2}{\alpha} \leq f_i < \frac{4}{\alpha}$, which implies that the cost is at most $(1 + \frac{8(1+\alpha)}{\alpha}) c_i$. Otherwise, $\alpha f_i \geq 4$ and the cost is then upper bounded by $(1 + \frac{2(1+\alpha) f_i}{\frac{\alpha f_i}{2} - 1}) c_i \leq (1 + \frac{2(1+\alpha)}{\frac{\alpha}{2} - \frac{1}{f_i}}) c_i$ which is also at most $(1 + \frac{8(1+\alpha)}{\alpha}) c_i$, as $\frac{1}{f_i} \leq \frac{\alpha}{4}$. Hence, the cost of S'_i is $\mathcal{O}(c_i)$ as α is a constant, which concludes the first case.

The second case is when $\alpha f_i < 2$. Here, we could not apply the same strategy as that of the first case because we have $\lfloor \frac{\alpha f_i}{2} \rfloor = 0$. Consequently, we adopt a slightly different strategy in which the agent executes S_i but interrupts it before each of its edge traversals. As explained in detail below, the kth interruption either consists of a return trip to node s before resuming S_i and making the kth edge traversal of S_i, or it consists in going to the node the agent should occupy at the end of the kth edge traversal of S_i but without taking the corresponding edge: the agent then resumes S_i as if it had just performed the kth edge traversal of S_i (essentially it just makes some computations before interrupting again S_i for the next edge traversal, if any). We will show that the latter situation will occur only when the "skipped edge" has already been traversed before by the agent. Once S_i has been entirely processed, S'_i is simply over if the agent is located at the node containing the treasure. Otherwise, the agent is at node s and $i < |S|$. In this case, it executes (similarly as in the previous case) a close period in order to guarantee that the size of Sk is 0 at the end of S'_i.

Let us first focus on the interruptions. We denote by $(u_1, u_2, u_3, \dots, u_{c_i+1})$ the sequence (with repetitions), in the chronological order, of the nodes that are visited during S_i, and by $(e_1, e_2, e_3, \dots, e_{c_i})$ the sequence (with repetitions), in the chronological order, of the edges that are traversed during S_i. Consider the kth interruption occuring at node u_k just before the kth edge traversal of S_i and assume that at the beginning of this interruption, the property $H(k)$, consisting of the following three conditions, is satisfied:

- The agent has made $D_k \leq f_i + 1$ edge traversals since the last time when Sk was empty (this could be the current time).
- The sequence of edges $(e_1, e_2, \dots, e_{k-1})$ has been previously explored by the agent.
- The size of Sk has been 0 at least once during any previous block of $2(f_i + 1)$ consecutive edge traversals and has never been greater than $f_i + 1$.

Note that at the beginning of the first interruption, property $H(1)$ immediately holds. We will show below that property $H(k + 1)$ is satisfied at the beginning of the $(k + 1)$th interruption, if any.

In the kth interruption, the agent first checks whether it knows a path of length at most f_i from node s to node u_k. If this is the case, the agent executes in the reverse order the last D_k edge traversals, at the end of which it is at node s and Sk is empty. Then, the agent comes back to u_k using the known path of length at most f_i from node s to node u_k (as when $\alpha f_i \geq 2$, if there are several such paths, the agent chooses the lexicographically smallest shortest among them). Once this is done, the interruption is over: the agent resumes S_i and makes the kth edge traversal to reach node u_{k+1}. We can easily show that at the end of this edge traversal, and thus at the beginning of the next interruption if any, property $H(k + 1)$ is satisfied.

So, assume that at the beginning of the kth interruption, the agent does not know a path of length at most f_i from node s to node u_k. In view of the fact that $D_k \leq f_i + 1$, the shortest path from node s to node u_k that is known by the agent has actually length exactly $f_i + 1$. Before explaining what the agent does, let us give some properties that necessarily hold in this situation. We have the following claim, whose proof is omitted.

▷ Claim 9. e_k belongs to G_{i-1} or to the sequence $(e_1, e_2, \dots, e_{k-1})$.

From the above claim, it follows that at the beginning of the kth interruption the agent has already traversed edge e_k before, and already knows which edge of G_{i-1} or of $(e_1, e_2, \dots, e_{k-1})$ corresponds to it. Thus, at the beginning of the kth interruption, the agent can already determine a path of length at most $f_i + 1$ from node s to node u_{k+1} because in view of Lemma 4 it must know such a path when reaching node u_{k+1} and because the traversal of e_k does not bring extra topological information on $\mathcal{G}$.

Now we are are able to formulate what the agent does, when it has noticed that it does not know a path of length at most f_i from node s to node u_k. It executes in the reverse order the last D_k edge traversals, at the end of which it is at node s and Sk is empty. Then, instead of coming back to u_k, it goes directly to node u_{k+1} using the known path (highlighted in the previous paragraph) of length at most $f_i + 1$ from node s to node u_{k+1}. Once this is done, the interruption is over: the agent resumes S_i and acts as if it had just traversed edge e_k (as previously mentioned, it just performs some computations before interrupting again S_i for the next edge traversal, if any). It follows that at the end of the interruption, and thus at the beginning of the following one if any, $H(k + 1)$ is satisfied. We have shown by induction on k that, at the beginning of the kth interruption, for any $k \geq 1$, the property $H(k)$ is satisfied. This closes the description of the interruptions.

It remains to deal with the close period. At the beginning of it, property $H(c_i + 1)$ is satisfied, which implies that the agent has performed $D_{c_i+1} \leq f_i + 1$ edge traversals since the last time when Sk was empty. Hence, during the close period, the agent simply executes in the reverse order the last D_{c_i+1} edge traversals, at the end of which Sk is empty. In view of this, of the fact that Sk is empty at the beginning of S'_i, and of property $H(c_i + 1)$, the frequent return property is satisfied during S'_i.

It follows from the above explanation that at the end of S'_i, G_i is entirely explored and the agent is at the node containing the treasure if $i = |S|$. If $i < |S|$, it also follows that the size of Sk is 0 at the beginning of the next emulation S'_{i+1}, and $M_1(S'_{i+1}) = M_1(S_{i+1}) = B_{f_{i+1}}(\mathcal{G}, s)$ because $M_2(S'_i) = M_2(S_i) = M_1(S_{i+1})$. Finally, concerning the cost of S'_i, observe that the number of interruptions is c_i and during the close period as well as during each interruption the agent makes at most $2(f_i + 1)$ edge traversals. The cost of S'_i is then upper bounded by $2(f_i + 1)c_i + 2f_i + 2$ which is at most $2(\frac{2}{\alpha} + 1)c_i + \frac{4}{\alpha} + 2$, as $f_i < \frac{2}{\alpha}$ in the currently analysed case. Hence, the cost of S'_i is $\mathcal{O}(c_i)$. This concludes the second case and thus concludes the proof of the theorem. ◀

References

1 Steve Alpern and Shmuel Gal. *The Theory of Search Games and Rendezvous*. Kluwer Academic Publisher, 2003. doi:10.1007/b100809.

2 Spyros Angelopoulos, Diogo Arsénio, and Christoph Dürr. Infinite linear programming and online searching with turn cost. *Theor. Comput. Sci.*, 670:11–22, 2017. doi:10.1016/j.tcs.2017.01.013.

3 Baruch Awerbuch, Margrit Betke, Ronald L. Rivest, and Mona Singh. Piecemeal graph exploration by a mobile robot. *Inf. Comput.*, 152(2):155–172, 1999. doi:10.1006/inco.1999.2795.

4 Ricardo A. Baeza-Yates, Joseph C. Culberson, and Gregory J. E. Rawlins. Searching in the plane. *Inf. Comput.*, 106(2):234–252, 1993. doi:10.1006/inco.1993.1054.

5 Anatole Beck. On the linear search problem. *Israel Journal of Mathematics*, 2:221–228, 1964. doi:10.1007/BF02759737.

6 Anatole Beck and Donald J. Newman. Yet more on the linear search problem. *Israel Journal of Mathematics*, 8:419–429, 1970. doi:10.1007/BF02798690.

7 Richard E. Bellman. An optimal search. *SIAM Review*, 5(3):274, 1963. doi:10.1137/1005070.

8 Pallab Dasgupta, P. P. Chakrabarti, and S. C. De Sarkar. Agent searching in a tree and the optimality of iterative deepening. *Artif. Intell.*, 71(1):195–208, 1994. doi:10.1016/0004-3702(94)90066-3.

9 Pallab Dasgupta, P. P. Chakrabarti, and S. C. De Sarkar. A correction to "agent searching in a tree and the optimality of iterative deepening". *Artif. Intell.*, 77(1):173–176, 1995. doi:10.1016/0004-3702(95)00089-W.

10 Erik D. Demaine, Sándor P. Fekete, and Shmuel Gal. Online searching with turn cost. *Theor. Comput. Sci.*, 361(2-3):342–355, 2006. doi:10.1016/j.tcs.2006.05.018.

11 Christian A. Duncan, Stephen G. Kobourov, and V. S. Anil Kumar. Optimal constrained graph exploration. *ACM Trans. Algorithms*, 2(3):380–402, 2006. doi:10.1145/1159892.1159897.

12 Rudolf Fleischer, Thomas Kamphans, Rolf Klein, Elmar Langetepe, and Gerhard Trippen. Competitive online approximation of the optimal search ratio. *SIAM J. Comput.*, 38(3):881–898, 2008. doi:10.1137/060662204.

13 G. Matthew Fricke, Joshua P. Hecker, Antonio D. Griego, Linh T. Tran, and Melanie E. Moses. A distributed deterministic spiral search algorithm for swarms. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2016, Daejeon, South Korea, October 9-14, 2016*, pages 4430–4436. IEEE, 2016. doi:10.1109/IROS.2016.7759652.

14 Shmuel Gal. Search games: A review. In *Search Theory: A Game Theoretic Perspective*, pages 3–15. Springer, 2013. doi:10.1007/978-1-4614-6825-7_1.

15 Subir Kumar Ghosh and Rolf Klein. Online algorithms for searching and exploration in the plane. *Comput. Sci. Rev.*, 4(4):189–201, 2010. doi:10.1016/j.cosrev.2010.05.001.

16 Artur Jez and Jakub Lopuszanski. On the two-dimensional cow search problem. *Inf. Process. Lett.*, 109(11):543–547, 2009. doi:10.1016/j.ipl.2009.01.020.

17 Ming-Yang Kao, John H. Reif, and Stephen R. Tate. Searching in an unknown environment: An optimal randomized algorithm for the cow-path problem. *Inf. Comput.*, 131(1):63–79, 1996. doi:10.1006/inco.1996.0092.

18 David G. Kirkpatrick and Sandra Zilles. Competitive search in symmetric trees. In Frank Dehne, John Iacono, and Jörg-Rüdiger Sack, editors, *Algorithms and Data Structures - 12th International Symposium, WADS 2011, New York, NY, USA, August 15-17, 2011. Proceedings*, volume 6844 of *Lecture Notes in Computer Science*, pages 560–570. Springer, 2011. doi:10.1007/978-3-642-22300-6_47.

19 Dennis Komm, Rastislav Královic, Richard Královic, and Jasmin Smula. Treasure hunt with advice. In Christian Scheideler, editor, *Structural Information and Communication Complexity - 22nd International Colloquium, SIROCCO 2015, Montserrat, Spain, July 14-16, 2015, Post-Proceedings*, volume 9439 of *Lecture Notes in Computer Science*, pages 328–341. Springer, 2015. doi:10.1007/978-3-319-25258-2_23.

20 Elmar Langetepe. On the optimality of spiral search. In Moses Charikar, editor, *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pages 1–12. SIAM, 2010. doi:10.1137/1.9781611973075.1.

21 Elmar Langetepe. Searching for an axis-parallel shoreline. *Theor. Comput. Sci.*, 447:85–99, 2012. doi:10.1016/j.tcs.2011.12.069.

22 Alejandro López-Ortiz and Sven Schuierer. The ultimate strategy to search on m rays? *Theor. Comput. Sci.*, 261(2):267–295, 2001. doi:10.1016/S0304-3975(00)00144-4.

23 Avery Miller and Andrzej Pelc. Tradeoffs between cost and information for rendezvous and treasure hunt. *J. Parallel Distributed Comput.*, 83:159–167, 2015. doi:10.1016/j.jpdc.2015.06.004.

24 Andrzej Pelc. Reaching a target in the plane with no information. *Inf. Process. Lett.*, 140:13–17, 2018. doi:10.1016/j.ipl.2018.04.006.

25 Sven Schuierer. Lower bounds in on-line geometric searching. *Comput. Geom.*, 18(1):37–53, 2001. doi:10.1016/S0925-7721(00)00030-4.

26 Amnon Ta-Shma and Uri Zwick. Deterministic rendezvous, treasure hunts, and strongly universal exploration sequences. *ACM Transactions on Algorithms*, 10(3):12, 2014. doi:10.1145/2601068.