Tree Exploration with Little Memory*

Krzysztof Diks[†] Pierre Fraigniaud[‡] Evangelos Kranakis[§] Andrzej Pelc[¶]

Abstract

A robot with k-bit memory has to explore a tree whose nodes are unlabeled and edge ports are locally labeled at each node. The robot has no *a priori* knowledge of the topology of the tree or of its size, and its aim is to traverse all the edges. While $O(\log \Delta)$ bits of memory suffice to explore any tree of maximum degree Δ if stopping is not required, we show that bounded memory is not sufficient to explore with stop all trees of bounded degree (indeed $\Omega(\log \log \log n)$ bits of memory are needed for some such trees of size n). For the more demanding task requiring to stop at the starting node after completing exploration, we show a sharper lower bound $\Omega(\log n)$ on required memory size, and present an algorithm to accomplish this task with $O(\log^2 n)$-bit memory, for all n-node trees.

Keywords: Graph exploration, universal traversal sequences.

*Extended abstract appeared in the Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '02).

[†]Instytut Informatyki, Uniwersytet Warszawski, Banacha 2, 02-097 Warszawa, Poland. *diks@mimuw.edu.pl*.

[‡]CNRS-LRI, Université Paris-Sud, 91405 Orsay, France. *http://www.lri.fr/~pierre*.

[§]School of Computer Science, Carleton University, Ottawa, Ontario, K1S 5B6, Canada. *kranakis@scs.carleton.ca*.

[¶]Département d'Informatique, Université du Québec à Hull, Québec J8X 3X7, Canada. *pelc@uqah.uquebec.ca*.

1 Introduction

A robot (mobile agent) has to *explore* an undirected graph by visiting all its nodes and traversing all edges, without any *a priori* knowledge of the topology of the graph nor of its size. The task of visiting all nodes of a network is fundamental in searching for data stored at unknown nodes of a network, and traversing all edges is often required in network maintenance and when looking for defective components. If nodes and edges have unique labels, this can be easily done by depth-first search. However, in some navigation problems in unknown environments such unique labeling may not be available, or limited sensory capabilities of the robot may prevent it from perceiving such labels. Hence it is important to be able to program the robot to explore *anonymous* graphs, i.e., graphs without unique labeling of nodes or edges. Unfortunately, arbitrary graphs cannot be explored under such weak assumptions, as witnessed by the case of a cycle: without any labels of nodes and without the possibility of putting marks on them, it is clearly impossible to explore a cycle of unknown size and stop. If marking of nodes (e.g., by dropping and removing *pebbles*) is available then the problem can be solved even in directed graphs (cf. [7]). Otherwise, the class of graphs that can potentially be explored has to be restricted to connected graphs without cycles, i.e., to trees.

In this paper we study the problem of graph exploration under very weak assumptions: we do not assume any labels on nodes and do not allow marking, hence we restrict attention to exploration of trees. Clearly the robot has to be able to *locally* distinguish ports at a node: otherwise it is impossible to explore even the star with 3 leaves (after visiting the second leaf the robot cannot distinguish the port leading to the first visited leaf from that leading to the unvisited one). Hence we make a natural assumption that all ports at a node are locally labeled 1,...,d, where d is the degree of the node. No coherence between those local labelings is assumed.

In many applications, robots (mobile agents) are meant to be simple, often small, and inexpensive devices which limits the amount of memory with which they can be equipped. As opposed to numerous papers that imposed no restrictions on the memory of the robot and sought exploration algorithms minimizing time, i.e., the number of traversals, we investigate the minimum size of the memory of the robot that allows exploration of trees of given (unknown) size, regardless of the time of exploration. More precisely, we consider the following main tasks:

- exploration with stop: starting at any node of the tree, the robot has to traverse all edges and stop at some node;
- exploration with return: starting at any node of the tree, the robot has to traverse all edges and stop at the starting node.

In both cases we want to find an algorithm for a robot performing the given task using as little memory as possible.

1.1 Our results.

We first consider the auxiliary easier task of *perpetual exploration* in which the robot has to traverse all edges of the tree but is not required to stop. Perpetual exploration may be of independent interest, e.g., if regular control of a network for the presence of faults is required, and all edges must be periodically traversed over long periods of time. In our context, the study of perpetual exploration shows that from the point of view of memory use, the most demanding part of exploration with stop is indeed stopping. Here is why. We show an algorithm performing perpetual exploration of any tree of maximum degree Δ using $O(\log \Delta)$-bit memory. In particular, bounded memory is sufficient for perpetual exploration of all bounded degree trees. In contrast we show that this is not the case for exploration with stop. Indeed, $\Omega(\log \log \log n)$ bits of memory are needed for exploration with stop of some bounded degree trees of size n. The additional memory is essentially needed to decide when to stop in our anonymous environment. Moreover, our memory-efficient perpetual exploration algorithm yields algorithms for exploration with stop in special classes of trees, or for all trees under additional knowledge of a bound on size. Indeed, if we know a bound N on the size of the tree, we can explore it with stop using $O(\log N)$-bit memory, while if we know a bound m on the number of nodes of a given degree then we can explore a tree with stop using $O(\log \Delta + \log m)$-bit memory.

For exploration with return, we show a lower bound $\Omega(\log n)$ on the number of memory bits needed for trees of size n. As for upper bounds, a simple DFS-based algorithm performs exploration with return using $O(D \log \Delta)$ memory bits for trees of diameter D and maximum degree Δ. However, the memory used by this algorithm can be linear in the size n for some trees. Our main algorithm for exploration with return is much more memory-efficient in general: it uses only $O(\log^2 n)$ memory bits for all trees of size n.

1.2 Related work.

Exploration and navigation problems for robots in an unknown environment have been extensively studied in the literature (cf. the survey [26]). There are two groups of models for these problems. In one of them a particular geometric setting is assumed, e.g., unknown terrain with convex obstacles [10], or room with polygonal [14] or rectangular [5] obstacles. Another approach is to model the environment as a graph, assuming that the robot may only move along its edges. The graph setting can be further specified in two different ways. In [1, 7, 8, 15] the robot explores strongly connected directed graphs and it can move only in the direction from head to tail of an edge, not vice-versa. In [3, 11, 16, 24] the explored graph is undirected and the robot can traverse edges in both directions. In the graph setting it is often required that apart from completing exploration the robot has to draw a map of the graph, i.e., output an isomorphic copy of it.

The efficiency measure adopted in most papers dealing with exploration of graphs is the time of completing this task, measured by the number of edge traversals by the robot. On the other hand, there are no restrictions imposed on the memory of the robot. (This is precisely, where our setting differs: we are interested in minimizing the memory of the robot but do not restrict exploration time.)

Graph exploration scenarios considered in the literature differ in an important way: it is either assumed that nodes of the graph have unique labels which the robot can recognize, or it is assumed that nodes are anonymous. Exploration of directed graphs assuming the existence of labels was investigated in [1, 15]. In this case no restrictions on the robot moves were imposed, other than by directions of edges, and fast exploration and mapping algorithms were sought. Exploration of undirected labeled graphs was considered in [3, 11, 16, 24]. Since in this case a simple exploration based on depth-first search can be completed in time $2e$, where e is the number of edges, investigations concentrated either on further reducing time for an unrestricted robot, or on studying efficient exploration when moves of the robot are restricted in some way. The first approach was adopted in [24], where an exploration algorithm working in time $e + O(n)$, with n being the number of nodes, was proposed. Restricted robots were investigated in [3, 11, 16]. It was assumed that the robot has either a restricted tank [3, 11], forcing it to periodically return to the base for refueling, or that it is tethered, i.e., attached to the base by a rope or cable of restricted length [16]. It was proved in [16] that exploration and mapping can be done in time $O(e)$ under both scenarios.

Exploration of anonymous graphs presents a different type of challenges. In this case it is impossible to explore arbitrary graphs if no marking of nodes is allowed. Hence the scenario adopted in [7, 8] was to allow *pebbles* which the robot can drop on nodes to recognize already visited ones, and then remove them and drop in other places. The authors concentrated attention on the minimum number of pebbles allowing efficient exploration and mapping of arbitrary directed n-node graphs. (In the case of undirected graphs, one pebble suffices for efficient exploration.) In [8] the authors compared exploration power of one robot to that of two cooperating robots with a constant number of pebbles. In [7] it was shown that one pebble is enough if the robot knows an upper bound on the size of the graph, and $\Theta(\log\log n)$ pebbles are necessary and sufficient otherwise.

Our scenario is even weaker than that in [7, 8]: nodes do not have labels and no marking is allowed. Since the presence of even one cycle precludes exploration with stop, we restrict attention to the class of (undirected) graphs in which this task is possible under these very weak assumptions: the class of trees. Moreover, as previously mentioned, our scenario differs in that we optimize memory, not exploration time.

In the framework of derandomized random walks, the objective is to produce an explicit universal traversal sequence (UTS), i.e., a sequence $p_1, \ldots, p_k$ of port labels, such that the path guided by this sequence visits all edges of any graph. It is known that, with high probability, a sequence of length $O(n^3 d^2 \log n)$, chosen uniformly at random, guides a walk in any d-regular (connected) graph of n nodes. Explicit UTS are known for 2-regular graphs (cf. [6, 12, 13, 19, 21]), for 3-regular graphs (cf. [4, 18, 23]), for cliques (cf. [2, 20]), and for expanders (cf. [17]). Some of these sequences can be constructed in log-space, and hence can produce perpetual exploration with compact memory. However, without the *a priori* knowledge of n, non of these constructions allows the robot to return to its original position, or even to stop. Kouckỳ [22] introduced the notion of a universal *exploration* sequence (UXS), i.e., a sequence $q_1, \ldots, q_k$ such that the robot leaves the current node x via port $p + q_i$ at the ith step, where p is the label of the port through which the robot entered node x. This notion allows to construct shorter sequences. For instance, 1^n is a UXS for cycles of order n, and $(10)^n$ is a UXS for cliques of order n. However, the knowledge of n is required to allow the robot to stop in the cycle (in the clique, $n = d - 1$). Our objective differs in that we want the robot to stop (possibly at its original position) in the absence of any *a priori* knowledge of the size of the network.

2 Terminology and preliminaries

A tree with locally labeled ports is an undirected tree whose nodes are unlabeled and edges incident to a node v have distinct labels 1,...,d, where d is the degree of v. Thus every edge uv has two labels which are called its port numbers at u and at v. Port numbering is local: there is no relation between labels given to an edge uv at u and at v.

A robot with k-bit memory is a deterministic state machine with $K = 2^k$ states among which a specified state S_0 is called *initial* and some specified states are called *final*. Such a robot, when placed in the initial state in any node of a tree with locally labeled ports, operates as follows. If the robot is in a node in a non-final state S, the state determines a local port number i_S. The robot leaves the node by this port. Upon traversing the corresponding edge, the robot reads the port number at the node it enters and the degree of this node. This pair of integers is an input symbol that causes the transition from state S to S'. The robot continues moving in this way until it enters a final state for the first time. Then it stops.

As said in Section 1, we consider three tasks of increasing difficulty: *perpetual exploration* in which the robot has to traverse all edges of the tree but is not required to stop, *exploration with stop* in which starting at any node of the tree, the robot has to traverse all edges and stop at some node, and *exploration with return* in which starting at any node of the tree, the robot has to traverse all edges and stop at the starting node. A robot is said to perform one of the above tasks in a tree, if starting at *any* node of this tree in state S_0 it completes this task in finitely many steps. (Notice that in the case of perpetual exploration, completing this task after finitely many steps means only traversing all edges, not necessarily stopping after it.)

The way in which operation of the robot in a tree is defined implies that a robot exploring a star with Δ leaves needs at least Δ states: if it has fewer states then, starting at the center of the star it could never enter some ports. Hence we get the following trivial lower bound.

Proposition 2.1 *A robot exploring all trees of maximum degree Δ must have $\Omega(\log \Delta)$-bit memory.*

The following simple algorithm performs perpetual exploration using only $O(\log \Delta)$ memory bits. It is actually just a reformulation of a proposition in [22] stating that 1^{2n} is a universal exploration sequence for trees of size n.

Algorithm `CHOOSE-NEXT`**.** The robot leaves the starting node by port 1. After entering any node of degree d by port i, the robot leaves it by port $(i \bmod d)+1$.

It is easy to see that, using algorithm `CHOOSE-NEXT` the robot traverses all edges of an n-node tree after at most $2(n-1)$ steps. Hence we have

Proposition 2.2 *Algorithm* `CHOOSE-NEXT` *accomplishes perpetual exploration of any tree of maximum degree Δ using $O(\log \Delta)$ bits of memory.*

This observation implies algorithms for exploration with stop if some additional information about the tree is available. One such situation arises when an upper bound N on the number of nodes in the tree is known. Then the robot can perform algorithm `CHOOSE-NEXT`, additionally counting steps, and stop after $2(N-1)$ steps. This can be done with $O(\log N)$-bit memory. Another type of additional information could be an upper bound m on the number of nodes of given degree d. Using algorithm `CHOOSE-NEXT` every such node is visited at most $2d$ times before all edges are traversed. Hence it is enough to perform this algorithm, additionally counting visits at nodes of degree d, and stop after $2md$ visits. This can be done with $O(\log \Delta + \log m)$-bit memory. For example, this enables exploration with stop of any rooted binary tree (whose root is the only node of degree 2) using bounded memory.

3 Exploration with stop

As we have seen, perpetual exploration can be performed in all bounded degree trees using bounded memory, i.e., the same robot can perform perpetual exploration in all trees of maximum degree bounded by a constant. This ability can be transfered to the more demanding task of exploration with stop for restricted classes of bounded degree trees, such as trees with bounded number of nodes of given degree. However, the main result of this section shows that exploration with stop of all trees of bounded degree, even of all trees of maximum degree 3, cannot be performed by the same robot. (Of course, all trees of maximum degree 2, i.e., paths, can be explored with stop by the same robot with only 3 memory bits).

Theorem 3.1 *For every robot there exists a tree of maximum degree 3 which this robot cannot explore with stop.*

Proof. Fix a robot with the set $\mathcal{S}$ of states, such that $|\mathcal{S}| = K$. The idea of the proof is to construct two trees, T and T', such that if the robot explores T and stops then, when run on T', it stops before exploring it. In fact, we will construct parts of the (larger) tree T' by simulating actions of the robot in T. These parts of T' will be "blown-up" images of parts of T. We will ensure that the state of the robot in specific nodes visited in T' is the same as in corresponding nodes in T. Hence the robot must stop in T' because it stops in T. However, due to the "blow-up" feature of T', we will be able to argue that some nodes of this tree remain unexplored by the robot.

We restrict attention to the class of trees all of whose internal nodes have degree 3. Moreover we consider only *symmetric* port labelings, i.e., such that both labels on an edge joining internal nodes are equal. Hence we may speak of colors 1,2,3, of such edges, where the color number of an edge is equal to both port numbers at this edge. The color of an edge joining a leaf with an internal node is the port number at the internal node (the port number at a leaf is always 1). Notice that incident edges have different colors. Call such colored trees of degree 3 *proper trees*.

For proper trees the behavior of the robot is particularly simple. If the robot is in a non-final state and is situated at a leaf, it takes the unique port, reads the port number at the neighbor (equal to the edge color) and transits to some state. If the robot is in a non-final state and is situated at an internal node, the state determines the color of the edge to take, the robot moves, it only verifies if the neighbor is a leaf or an internal node (in both cases the entry port number is already known) and it transits to a new state. In particular, on a path consisting only of internal nodes, the state of the robot at the end of the path depends only on the state at the beginning of it.

A *trajectory* of the robot starting in a node of a proper tree in the initial state is the sequence of nodes it visits until stop. A *run* is defined as any segment of a trajectory. To any run corresponds a sequence of states of the robot when it leaves consecutive nodes of the run.

Consider a proper infinite tree with exactly one leaf. Let the robot start at the leaf. Without loss of generality we may assume that the robot visits nodes of this tree arbitrarily distant from the leaf (otherwise it is easy to construct a finite tree which the robot would not explore). For any positive integer x, define the following function $f_x : \mathcal{S} \times \{1,2,3\} \longrightarrow \mathcal{S} \times \{1,2,3\}$. Fix $S \in \mathcal{S}$ and $i \in \{1,2,3\}$. Suppose that the robot starts at the leaf in state S and that the color of the edge incident to the leaf is i. Let v be the first node at distance x from the leaf, visited by the robot. Cut the two edges incident to v different from the edge by which the robot entered v, thus making v a leaf. Let $f_x(S,i) = (S', j)$, where S' is defined as the state of the robot after entering the leaf v and j is defined as the color of the edge by which the robot enters v.

A (non-necessarily simple) path in a proper tree is called *normal* if both of its ends are leaves of the tree and all internal nodes of the path are internal nodes of the tree.

Let $g = 3 \cdot 3K^2(3K)^{3K}$. Even integers a and b are called *siblings* if $2K < a < a + 3K^2 < b < 2K + g$ and $f_a = f_b$.

Claim 3.1 *(1) There exists a pair of siblings. (2) If a and b are siblings and the robot starts traversing a normal path in a given state then the state of the robot at the other end of the path and the color of the entry edge to it are the same, regardless of whether the length of the path is a or b.*

Part 1 of the Claim follows from the fact that there are only $(3K)^{3K}$ possible functions f_x. Part 2 follows from the definition of functions f_x.

Fix a pair of siblings a and b, with $a < b$. We now construct the two proper trees with the property mentioned in the beginning of the proof.

Construction of the tree T.

Take a node v of degree 3 and attach to each of its 3 neighbors a complete binary tree of height $a/2 - 1$. Each of these complete binary trees is called a *principal subtree* of T and v is called the center of T. The tree T is proper and has diameter a. The coloring of its edges is unique up to automorphism. This completes the construction.

Suppose that the robot explores T and eventually stops, when starting at a leaf w. We define a *long trip* in T to be a run of the robot in T which starts and ends at leaves at distance a and traverses only internal nodes on the way. (Hence a long trip is a normal path.) The robot must make at least 2 and at most K long trips before stopping. The first is clear and the second follows from the fact that if the robot is twice in the same non-final state in the same node then it never stops. More than K long trips would cause the robot to be in the same non-final state in the central node v at least twice.

The rest of the proof is devoted to the construction of tree T' and to showing that the robot must stop before exploring it entirely. The construction proceeds incrementally by adding consecutive pieces to the tree T' under construction. Steps of the construction will be interleaved with arguments concerning the behavior of the robot. In order to facilitate reading, each step of the construction (consisting in adding a new piece of the tree) will be preceded by a bullet.

Construction of the tree T'.

- Initialize T' to the empty tree.

Let T'_1 be an isomorphic copy of the principal subtree T_1 of T containing the leaf w_1 beginning the first long trip, with the same coloring as in the original. Let w'_1 be the leaf in T'_1 corresponding to w_1.

- Add T'_1 to T'.

Let σ_1 be the run of the robot before the first long trip in T. σ_1 may have nodes outside of T_1, in principal subtrees of T, other than T_1. However, the only leaves on this run are those of T_1. Since there are no leaves at distance a on it, call this run a *short trip*. Let σ'_1 be an isomorphic copy of σ_1.

- Add σ'_1 to T'.

Consider the first long trip r_1 in T starting at node w_1 and ending at node u_1 at distance a from w_1. Denote by p_1 the simple path between w_1 and u_1. Consider the state of the robot at the first visit of each node of this path. Since $a/2 > K$, for some internal nodes on the part of path p_1 inside T_1, this state is the same. Call it $\hat{S}$. Let x be the first and y the second such node on p_1. Let α denote the part of the run r_1 between first visits of x and y. During α the robot does not visit any leaf of T_1. This proves the following claim.

Claim 3.2 *During α the robot is never at distance larger than $a/2$ from the path p_1.*

Let S_1 be the state of the robot at w_1 at the beginning of the first long trip r_1 in T. Consider the infinite proper tree containing T'_1 whose set of leaves is exactly the set of leaves of T'_1. Consider the run r'_1 of the robot in this infinite tree starting at w'_1 in state S_1, until hitting a node u'_1 at distance b from w'_1. Let p'_1 be the simple path joining w'_1 and u'_1. p'_1 has an initial segment which is an isomorphic copy of p_1 with the same coloring as in p_1. Let x' and y' be nodes on p'_1 corresponding to x and y, respectively. Let $S(\alpha)$ be the sequence of states associated with run α. Since the robot is in state $\hat{S}$ at the first visits of x and y in run r_1, it is also in state $\hat{S}$ at the first visits of x' and y' in run r'_1.

Claim 3.3 *There exists a sequence of nodes $s_0 = x'$, $s_1 = y'$, $s_2, \ldots$, in path p'_1, such that the sequence of states associated with the run between s_i and s_{i+1} is exactly $S(\alpha)$.*

Nodes s_i, for $i \geq 0$, are defined as nodes visited in run r'_1 at the end of consecutive segments of $\beta = |S(\alpha)|$ states, starting at x' in state $\hat{S}$. Since the robot leaves each node s_i in state $\hat{S}$, the construction of nodes s_i can be carried on inductively all along path p'_1, provided that no leaf is visited between visits of s_i and s_{i+1}. For s_i inside T'_1, this follows from the definition of a long trip. For s_i outside of T'_1 (and hence further than $a/2$ from any leaf), this follows from Claim 3.2.

Let I be the sequence of edge colors on path p'_1 between nodes x' and y'. Claim 3.3 implies:

Claim 3.4 *The sequence I of edge colors is repeated periodically on path p'_1 until u'_1 is reached.*

- Add run r'_1 to T', with appropriate coloring of edges.

Let T_2 be the principal subtree of T containing the leaf u_1 at the end of the first long trip in T. Let T'_2 be an isomorphic copy of T_2 such that the node u'_1 ending run r'_1 becomes a leaf of T'_2 corresponding to u_1. Reproduce edge coloring from T_2 in T'_2.

- Add tree T'_2 to T'.

We observed that, during run r'_1 no leaves are visited. Hence the run r'_1 is a normal path (provided that all nodes of run r'_1 have degree 3 in T', which we will guarantee at the end of the construction). Hence the state of the robot after hitting the leaf u'_1 in T' is the same as the state at the end of the first long trip in T . Intuitively speaking, the robot does not know at this point if it is at u'_1 in T', or at u_1 in T.

Since I is of length at most K, the definition of b and Claim 3.4 imply that there are at least $3K$ periods I of edge colors on p'_1 between the root of T'_1 and the root of T'_2.

Let now w_2 be the leaf in T_2 from which the second long trip in T is started and let w_2' be the corresponding leaf in T_2'. Consider the run of the robot in T between the end of the first long trip and the beginning of the second. Call it a short trip, as before: it has the same properties as those observed for the first short trip. Let σ_2' be an isomorphic copy of this short trip between leaves u_1' and w_2' (again a part of it may be outside of T_2').

- Add σ_2' to T'.

Consider the state S_2 of the robot at the beginning of the second long trip in T. In the tree T' under construction, the robot is in the same state, at w_2', after faithfully reproducing the short trip between u_1 and w_2 by an isomorphic run between u_1' and w_2' (provided that all those nodes on this run which correspond to internal nodes of T have degree 3 in T', which we will guarantee at the end of the construction).

Consider the infinite proper tree containing the part of T' already constructed, with no leaves outside of T_1' and T_2'. Consider the run r_2' of the robot in this infinite tree starting at w_2' in state S_2, until hitting a node u_2' at distance b from w_2'. As before, the robot does not visit any leaf of T_2' during this run. Two cases are possible: either u_2' is a leaf in T_1' or u_2' is some node of degree 3. In the first case we repeat the previous part of the construction, simulating the short trip of the robot following the second long trip in T by its run beginning at u_2' and ending at a leaf w_3' in T_1'. In the second case we continue the construction of the tree T' as follows:

- Add run r_2' to T' (with appropriate coloring of edges).

Let T_3 be the principal subtree of T containing the leaf u_2 at the end of the second long trip in T. (Notice that T_3 could be equal to T_1.) Let T_3' be an isomorphic copy of T_3 such that the node u_2' ending run r_2' becomes a leaf of T_3' corresponding to u_2. Reproduce edge coloring from T_3 in T_3'.

- Add tree T_3' to T'.

(Note that, if $T_3' = T_1'$, the above step can be omitted.) We need to observe a property that will be crucial for continuing the construction inductively. Let p_2' be the simple path between w_2' and u_2'. Let p_1'' be the reverse of path p_1'. Path p_2' is called *equivalent* to p_1'' if $T_3' = T_1'$.

Claim 3.5 *If p_2' is not equivalent to p_1'' then the last common node on these paths is before the middle of p_1'' (and hence also of p_2').*

Suppose that p_2' is not equivalent to p_1'' and let z be the last common node on these paths. There are at least $3K$ periods I of edge colors on p_1' (and hence also on p_1'') between the root of T_1' and the root of T_2'. Suppose that z is in the second half of p_1''. Hence there are more than K periods before z. Consider nodes $z_1,...,z_j$ beginning these periods. Since $j > K$, it follows that, for at least two of these nodes, the state of the robot is the same when it executes run r_2'. Since these nodes are separated by at least one period I, it follows that the robot will reproduce period I, at least until x', on the path p_2'. This implies that z is after x' on path p_2'. Hence z is a node of T_1', and consequently $T_3' = T_1'$, contradicting the assumption that p_2' is not equivalent to p_1''. This completes the proof of Claim 3.5.

We continue the construction of T', adding new runs of the robot in the infinite tree (corresponding to long trips and to short trips in T) and adding trees T'_i isomorphic to principal subtrees of T at the end of each run corresponding to a long trip. Suppose that $T'_1, ..., T'_i$ are already constructed and that we start the ith run r'_i, corresponding to the ith long trip in T, at a leaf w'_i of T'_i (as before, until hitting a node at distance b from w'_i). We must show that this run cannot hit any of the leaves of $T'_1, ..., T'_{i-2}$ unless they are at distance b from w'_i. Otherwise we could not argue that the state after the ith run is identical to that after the ith long trip in T.

Claim 3.6 *During the run r'_i the robot does not visit any of the leaves of $T'_1, \ldots, T'_{i-2}$, unless they are at distance b from w'_i.*

Let p'_j, for $1 \leq j \leq i$, be the simple path between a leaf of T'_j and a leaf of T'_{j+1}, corresponding to the jth long trip, and let p''_j be the reverse of this path. Suppose that during run r'_i the robot hits a leaf in some T'_m, for $m \leq i-2$, and that leaves of T'_m are at distance other than b from w'_i. Hence pairs of paths p'_i, p''_{i-1} and p'_{i-1}, p''_m are not equivalent. Similarly as argued in Claim 3.5, p'_i deviates from p''_{i-1} before the middle of it. On the other hand, p'_{i-1} deviates from p''_m before the middle of it and hence after the middle of p''_{i-1}. Consequently, p'_i cannot enter T'_m, which gives a contradiction. (Notice that r'_i may hit a leaf of T'_{i-1} but this is not a problem, because such a leaf is at distance b from w'_i.) This completes the proof of Claim 3.6.

- Add as many runs and trees T'_j in the above described manner, as there are long and short trips in the operation of the robot in tree T.

Suppose that after the run corresponding to the last long trip, the robot is in leaf u'_i of tree T'_{i+1} in state S^*. Consider the behavior of the robot in tree T when it is situated in leaf u_i finishing the last long trip, and is in state S^*. The robot executes some run σ without ever hitting leaves in principal subtrees other than that to which u_i belongs (i.e., it makes another short trip), and then stops at some node. Let σ' be an isomorphic copy of this final short trip.

- Add σ' to T'.

Now the construction is almost finished. Let $\hat{T}$ be the tree constructed so far. Its maximum degree is 3. Let L be the set of all its leaves which are not leaves of any tree T'_j (these are nodes in which the robot returned during runs corresponding to long or short trips). Let M be the set of nodes of degree 2 in $\hat{T}$.

- To each node in L add two new neighbors of degree 1, and to each node in M add a new neighbor of degree 1.

This completes the construction of tree T'.

The tree T' is a proper tree with the following property: all internal nodes of a run corresponding to a long trip in T are of degree 3. Hence, in view of the properties of siblings a and b, the state of the robot in the leaf of T' ending a given run is the same as the state of the robot in the leaf of T ending the corresponding long trip. The definition of the final run σ' (after leaving u'_i) implies that the robot must stop at the end of it. However, it did not explore the entire tree T': indeed, it only explored nodes of the subtree $\hat{T}$, and $T' \setminus \hat{T}$ is nonempty because either the set L or the set M are nonempty. This concludes the proof. $\square$

The following result can be obtained by estimating the size of the tree T' (constructed in the above proof), which the robot fails to explore with stop.

Theorem 3.2 *A robot which can explore with stop every n-node tree of maximum degree 3 must have $\Omega(\log\log\log n)$-bit memory.*

Proof. We use the notation and terminology from the proof of Theorem 3.1. Consider a simple path p_i' of length b joining two ends of a run r' of the robot in T', corresponding to a long trip in T. By Claim 3.2 from the proof of Theorem 3.1, during the run r' the robot is never at distance larger than a from the path p_i'.

On the other hand, during runs corresponding to short trips, the robot is never farther than at distance a from the closest leaf of the tree T_i' where this run starts and ends. Otherwise, this short trip would contain a long trip because in T the robot would hit a leaf at distance a from the start of the short trip.

Let T^* be the union of all trees T_i' and all simple paths p_i' defined during the construction of T'. Attach to each node u of degree 2 of T^* a new neighbor u' and attach to it a complete binary tree of height a rooted at u'. Denote the resulting proper tree by T^{**}. Hence T^{**} includes T'.

We now estimate the size of T^{**}. Every path p_i' has length $b \leq 2K + 3 \cdot 3K^2(3K)^{3K}$ and every tree T_i' has size smaller than $2^a < 2^b$. There are at most K paths p_i' and at most $K+1$ trees T_i'. The total length of all paths is at most Kb and a tree of size at most 2^{a+1} is attached to each of their nodes. Consequently the number $n = 2^{(4K)^{4K}}$ is an overestimate of the size of T^{**} and hence also of T'. Hence a robot with k-bit memory, where $2^k = K$, fails to explore with stop some tree of maximum degree 3 and of size $n = 2^{(4K)^{4K}}$. This implies that in order to explore with stop all n-node trees of maximum degree 3, the robot must have $\Omega(\log\log\log n)$-bit memory. $\square$

4 Exploration with return

We now turn attention to the most demanding of our exploration tasks, when we require the robot to stop at the starting node after performing exploration. For this task we have a much sharper lower bound on memory than for exploration with stop. In fact, even a robot specifically constructed for exploration with return of an *a priori given* tree requires logarithmic memory.

Theorem 4.1 *Let T be any tree of size n. A robot which can explore with return the tree T must have $\Omega(\log n)$-bit memory.*

Proof. Suppose that a robot with $k < \lfloor \log n \rfloor$ memory bits can perform exploration with return in T. Fix any node v of T. For any node u, let S_u be the state of the robot at the first visit of v when u is the node at which the robot starts. Since the robot has fewer than n states, for two different starting nodes u_1 and u_2 we have $S_{u_1} = S_{u_2}$. Hence the part of the trajectory of the robot after the first visit of v is the same, regardless of whether it started at u_1 or at u_2. Consequently, the node at which the robot stops is the same in both cases. This cannot be both u_1 and u_2, which contradicts the definition of exploration with return. $\square$

As for upper bounds, we present two algorithms. The first, Algorithm `STACK-OF-PORTS`, is a natural exploration with return, based on depth-first search. The label of the port by which a robot enters a new node is pushed on a stack and popped after the last visit of the node.

Algorithm `STACK-OF-PORTS`. The robot can be in two modes: F (forward) or B (backward) and uses a stack of integers.

1. The robot starts in mode F in some node v of degree d. It pushes a special symbol $\star$ on the empty stack and leaves v by port 1.
2. When the robot enters an internal node w of degree d_w by port i in mode F, it pushes the integer i on the stack and leaves by port $(i \bmod d_w)+1$.
3. When the robot enters a leaf it switches to mode B and returns to the neighbor of this leaf.
4. When the robot enters an internal node w of degree d_w by port i in mode B it compares i to the top of the stack.
   (a) If the top of the stack is an integer j and $i \not\equiv j-1 \bmod d_w$ then the robot switches to mode F and leaves by port $(i \bmod d_w)+1$.
   (b) If the top of the stack is an integer j and $i \equiv j-1 \bmod d_w$ then the robot pops the stack and leaves by port j.
   (c) If the top of the stack is $\star$ and $i < d_w$ then the robot switches to mode F and leaves by port $(i \bmod d_w)+1$.
   (d) If the top of the stack is $\star$ and $i = d_w$ then the robot stops.

The height of the stack never exceeds the diameter D of the tree and each label uses $O(\log \Delta)$ memory bits, where Δ is the maximum degree. Hence we have:

Theorem 4.2 *Algorithm* `STACK-OF-PORTS` *accomplishes exploration with return of any tree of diameter D and maximum degree Δ using $O(D \log \Delta)$ bits of memory.*

In fact the bound on the memory required by Algorithm `STACK-OF-PORTS` can be slightly improved by observing that the total number of bits in all elements simultaneously on the stack is at most $O(\log c_1 + \cdots + \log c_k)$, where $c_1, ..., c_k$ is a sequence of degrees of nodes on a branch of the tree. However, this number of bits can still be linear in the size of the tree. This should be compared to our main algorithm for exploration with return which we present below. It accomplishes exploration with return in any n-node tree using only $O(\log^2 n)$ memory bits.

We first describe a family of exploration protocols. Each protocol is designed for a specific value of the number of nodes. A protocol running for any value of n while preserving the nice features of this family will be presented later. We need the following concepts:

Definition 4.1 *Given a tree T, a node s of T, and $\ell \in \{0, 1, \ldots, \deg(s)\}$, the* open subtree $T_{s,\ell}$ *of T is T if $\ell = 0$, and it is the subtree of T consisting of the connected component of T containing s after the removal of its incident edge corresponding to port ℓ, otherwise.*

Exploring an open subtree $T_{s,\ell}$ means exploring T from s if $\ell = 0$, and otherwise it consists of the following two tasks: (1) explore $T_{s,\ell}$ from s and return to s, and (2) leave s through port ℓ. Note that $T_{s,\ell}$ might be a subtree of an unknown tree. Note also that the case $\ell = 0$ would not need a special treatment if we assumed that the expression "leave through port 0" means "stay here" (we recall that ports are labeled from 1 to the degree of the node).

Lemma 4.1 *There exists a family of exploration protocols $\{\mathcal{E}_k, k \geq 0\}$, satisfying the following properties:*

1. *$\mathcal{E}_k$ allows exploration of any n-node open subtree $T_{s,\ell}$ of any unknown tree, for $n \leq 2^k$.*
2. *When the exploration terminates, the robot reports* SUCCESS *if $n \leq 2^k$, and* FAILURE *otherwise.*
3. *If the robot reports* SUCCESS *then it knows n.*

Proof. The construction of the family $\{\mathcal{E}_k, k \geq 0\}$ is by induction on k. Let us start by the description of the trivial protocol $\mathcal{E}_0$. We consider exploration of an open subtree $T_{s,\ell}$ by a robot $\mathcal{R}$. The case $\ell = 0$ is straightforward: if $\deg(s) = 0$, then $\mathcal{R}$ reports SUCCESS, otherwise it reports FAILURE. The case $\ell > 0$ is not much more involved: $\mathcal{R}$ leaves through port ℓ and reports FAILURE if $\deg(s) > 1$ and SUCCESS otherwise. It is easy to check that $\mathcal{E}_0$ satisfies all the properties of the lemma.

Let us now assume that we have constructed a sequence $\mathcal{E}_0, \mathcal{E}_1, \ldots, \mathcal{E}_k$ of protocols satisfying the properties of the lemma, and let us construct the protocol $\mathcal{E}_{k+1}$. We will later prove that $\mathcal{E}_{k+1}$ is correct, i.e., satisfies the properties of the lemma.

Construction. We consider exploration of an open subtree $T_{s,\ell}$. The robot $\mathcal{R}$ can be in three states: EXPLORE, SUCCESS, or FAILURE. It starts in the state EXPLORE.

Variables used by the robot:

$\mathcal{R}$ uses the following set of variables:

- `parent` is a stack used to store "ranks" of port-labels, encoded by integers of variable size;
- `counter` is a stack that stores counters of non-negative values;
- `threshold` is a stack that stores integers ≥ 1 that are ceilings of ratios of subtree-sizes;
- `number` is a counter which counts the number of nodes that have been visited so far;
- `hop-count` is a counter which stores the distance between s and the current position of $\mathcal{R}$; and
- a constant number of integer variables used to store port-labels.

Initially `hop-count` $= 0$, `number` $= 1$, `parent` and `threshold` are empty, and `counter` contains a single 0.

Informal description.

Stack `parent` stores the port-labels that will be used to go back to s. Actually, not exactly port-labels are stored in `parent` but rather the "ranks" of these port-labels. For the sake of clarity however, just consider that port-labels are stored, as in the case of Algorithm `STACK-OF-PORTS`. However, as opposed to the case of `STACK-OF-PORTS`, not all port-labels are stored in `parent`. The basic idea is the following. Assume $\mathcal{R}$ is currently at node u of tree T, and entered u via port p. Consider T as rooted at u. $\mathcal{R}$ must explore all the subtrees of T rooted at the neighbors of u, distinct from the subtree $T^{(p)}$ attached to port p. For that purpose, $\mathcal{R}$ uses the protocols $\mathcal{E}_i$ for $i \leq k$. If $\mathcal{R}$ succeeds to explore all subtrees using these protocols, then it just backtracks through port p, reporting SUCCESS of $\mathcal{E}_{k+1}$. Similarly, if $\mathcal{R}$ fails to explore at least two subtrees using $\mathcal{E}_k$, then at least two subtrees have size $> 2^k$, thus T has more than 2^{k+1} nodes, and therefore $\mathcal{R}$ backtracks through port p, reporting FAILURE of $\mathcal{E}_{k+1}$.

The delicate case is when there is a unique subtree $T^{(q)}$, $q \neq p$, that $\mathcal{R}$ fails to explore using $\mathcal{E}_k$. Then $\mathcal{R}$ tries to explore $T^{(q)}$ using $\mathcal{E}_{k+1}$. However, before leaving u, $\mathcal{R}$ must have a way to "remember" p so that it can return to s after having explored $T^{(q)}$ and returning to u. Recall that we are in the following situation: all subtrees can be explored using some $\mathcal{E}_i$ for $i \leq k$, $T^{(q)}$ cannot be explored using $\mathcal{E}_k$, and $T^{(p)}$ is the subtree attached to the port p through which $\mathcal{R}$ entered u. Then $\mathcal{R}$ tries to explore $T^{(p)}$ as well, using protocols $\mathcal{E}_i$, for $i \leq k$.

First assume that $T^{(p)}$ cannot be explored using $\mathcal{E}_k$. Then port p will be easy to compute when $\mathcal{R}$ is back at u after having explored $T^{(q)}$: it will be the only subtree distinct from $T^{(q)}$ that even $\mathcal{E}_k$ fails to explore.

Similarly, assume that $T^{(p)}$ can be explored using some $\mathcal{E}_i$, $i \leq k$, and all other subtrees distinct from $T^{(p)}$ and $T^{(q)}$ can be explored using some $\mathcal{E}_j$, for $j < i$. Then again port p will be easy to compute when $\mathcal{R}$ is back at u after having explored $T^{(q)}$: it will be the unique subtree distinct from $T^{(q)}$ requiring the protocol $\mathcal{E}_i$ with largest i for its exploration.

The two above cases are the ones for which port p is <u>not</u> stored in stack `parent`. The former case is easy: when $\mathcal{R}$ is back at u after having explored $T^{(q)}$, it will re-explore all subtrees, and $T^{(p)}$ is the only one that it cannot explore using $\mathcal{E}_k$. However, a question remains in the latter case: $\mathcal{R}$ will re-explore all subtrees and find a unique subtree $T^{(\varrho)}$, $\varrho \neq q$, requiring the protocol $\mathcal{E}_i$ with largest i for its exploration. $\mathcal{R}$ needs to know whether $p = \varrho$, or $p \neq \varrho$ but p is stored in `parent`. This is the purpose of stack `counter`.

Stack `counter` stores counters that count the number of times p is indeed equal to ϱ. If at u, $p \neq \varrho$, then a new counter is added at the top of `counter`, with value 1. If at u, $p = \varrho$, then the counter currently at the top of `counter` is incremented by 1. So, when $\mathcal{R}$ is back at u, there are two cases. If the counter at the top of `counter` has value > 1, then $p = \varrho$ and $\mathcal{R}$ decrements the counter by 1. If the counter at the top of `counter` has value 1, then p is in stack `parent`, and $\mathcal{R}$ pops the counter currently at the top of `counter`. Using this approach, we save space in the stack parent, storing the port label p only if there is no unique subtree $T^{(\varrho)}$, $\varrho \neq q$, requiring the protocol $\mathcal{E}_i$ with largest i for its exploration. It will then be easy to prove that at most $O(\log n)$ pushes are performed in `parent`.

There still remains a problem, though. Since $\mathcal{R}$ increments the counter currently at the top of `counter` every time when $p = \varrho$, it follows that the value of a counter may become very large. For example, a path of $\Theta(\sqrt{n})$ nodes in the tree yields a counter on $\Theta(\log n)$ bits, and there might be lots of such paths. To overcome this problem, we use thresholds, and increase the counters only if there are other subtrees, distinct from $T^{(p)}$ and $T^{(q)}$, with size larger than a given threshold.

More precisely, when a new counter with value 1 is placed in the stack `counter`, say at node x, $\mathcal{R}$

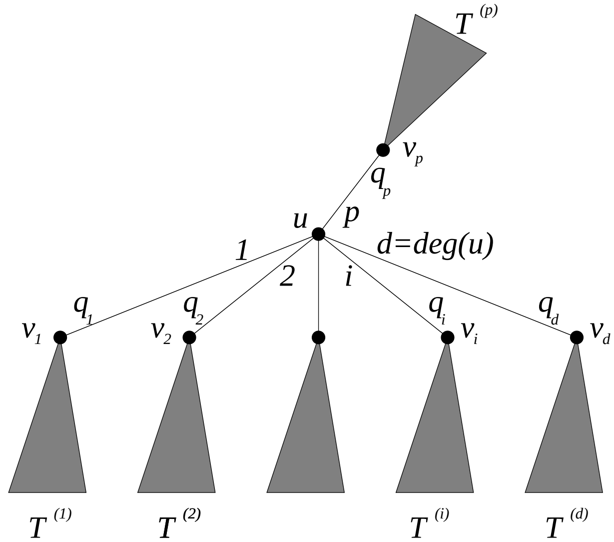

Figure 1: The robot is currently at node u

computes the ratio $r = |T^{(\varrho)}|/|T^{(p_x)}|$ which is pushed in the stack `threshold`. Here p_x denotes the port number leading to node s at x. Later, say at node y, the counter at the top of `counter` is increased only if there is a subtree $T^{(j)}$, $j \notin \{p_y, q\}$, such that $|T^{(j)}| \geq |T^{(p_y)}|/r$. Here p_y denotes the port number leading to node s at y. It follows that if z denotes the next node at which a new counter with value 1 is placed in the stack `counter`, then, between x and z, the current counter has not been increased more than $|T^{(p_z)}|/|T^{(p_x)}|$ times.

Hence an amortized analysis will allow us to prove that $O(\log n)$ bits are sufficient to encode all counters in the stack `counter`. A similar type of analysis will allow us to prove that $O(\log n)$ bits are also sufficient to encode all thresholds in `threshold`.

As a last remark, the stack `counter` also contains counters that are set to 0. They are pushed in `counter` at nodes for which there are at least two subtrees $T^{(i')}$ and $T^{(i'')}$ of maximum size. These counters are never increased, but only used as markers to separate the "real" counters.

Tables 1 and 2 summarize the operations performed by the robot on the three stacks. Now, we give the details of the protocol $\mathcal{E}_{k+1}$.

Actions of the robot:

We describe the actions of $\mathcal{R}$ in state EXPLORE, and currently at node u (possibly $u = s$), see Figure 1. Let p be the port through which $\mathcal{R}$ entered u ($p = \ell$ if $u = s$). For $i = 1, \ldots, \deg(u)$, let v_i be the neighbor of u connected to u by the link corresponding to port i at u. Viewing T as rooted at u, let $T^{(i)}$ be the subtree of T rooted at v_i. $\mathcal{R}$ successively tries to explore every $T^{(i)}$ with the $\mathcal{E}_j$'s, $j \leq k$, by increasing order of the index j. More precisely, for every $i = 1, ..., \deg(u)$ let q_i be the port label at v_i of the link connecting v_i to u. For $i = 1$ to $\deg(u)$, including $i = p$, $\mathcal{R}$ does the following. Starting from $j = 0$, and increasing j by one in case of failure, $\mathcal{R}$ repeats (1) move to v_i, and (2) explore the open subtree $T^{(i)}_{v_i,q_i}$ with $\mathcal{E}_j$. This process is carried on until either there is $k_i \leq k$ for which $\mathcal{E}_{k_i}$ succeeds (i.e., for which $\mathcal{R}$ comes back to u in the state SUCCESS), or all $\mathcal{E}_j$'s, $0 \leq j \leq k$, fail to explore $T^{(i)}$. When $\mathcal{R}$ has completed the previous operations for each $T^{(i)}$, it continues as follows:

I1 If, for every subtree $T^{(i)}$, $i \neq p$, there is $k_i \leq k$ such that $\mathcal{E}_{k_i}$ succeeded to explore the open subtree $T^{(i)}_{v_i,q_i}$, then $\mathcal{R}$ switches to the state SUCCESS. Indeed, the whole tree has been explored, and it just remains to return to s.

I2 If there is exactly one subtree $T^{(q)}$, $q \neq p$, for which all $\mathcal{E}_j$'s fail, then $\mathcal{R}$ stores q and stays in state EXPLORE. Indeed, one subtree remains to be fully explored. ($\mathcal{R}$ failed to explore that tree using $\mathcal{E}_k$, but exploration with $\mathcal{E}_{k+1}$ is potentially doable.)

I3 If there are at least two distinct subtrees $T^{(q)}$ and $T^{(q')}$, $q \neq p$ and $q' \neq p$, for which all $\mathcal{E}_j$'s fail, then $\mathcal{R}$ switches to the state FAILURE. Indeed, two subtrees have more than 2^k nodes, and hence the tree has more than 2^{k+1} nodes, yielding failure of $\mathcal{E}_{k+1}$.

In the three cases, $\mathcal{R}$ sums the number of vertices of the subtrees $T^{(i)}$, $i \neq p$, whose exploration succeeded and adds it to `number`. If `number` $> 2^{k+1}$, then $\mathcal{R}$ switches to the state FAILURE (if not yet in that state). At this point of the exploration, two cases have to be considered, depending on whether $\mathcal{R}$ is in state EXPLORE or not.

Case 1: $\mathcal{R}$ is in state EXPLORE. In this case, only one subtree $T^{(q)}$ of u remains to be explored. Before leaving u, $\mathcal{R}$ updates the three stacks `parent`, `counter` and `threshold` as follows.

– Stack `parent`:

(1) If there is $j \notin \{p, q\}$ such that $k_j \geq k_p$ then let $\nu_q(p)$ be the number of indices $j \neq q$ such that either (i) $k_j > k_p$, or (ii) $k_j = k_p$ and $j \leq p$. The value of $\nu_q(p)$ is placed at the top of `parent`.

(2) Otherwise (i.e., all $\mathcal{E}_j$'s, $0 \leq j \leq k$, fail to explore $T^{(p)}$ or $k_p > k_i$ for all $i \notin \{p, q\}$) `parent` is left unchanged.

The port-label p that leads to s is stored in `parent` only if k_p is not the largest index, i.e., there is at least a tree $T^{(j)}$, $j \neq p$, as large as $T^{(p)}$. This insures that the maximum size of `parent` never exceeds $O(\log n)$ elements. Actually, not the port-label p is stored in `parent`, but its rank $\nu_q(p)$.

– Stacks `counter` and `threshold`:

These two stacks are introduced to store additional information used to recover p in case (2), i.e., when no information about p is stored in `parent`. They are updated according to the following rules:

(a) If all $\mathcal{E}_j$'s, $0 \leq j \leq k$, fail to explore $T^{(p)}$ then `counter` and `threshold` are left unchanged.

(b) If there are two distinct indices i' and i'', both different from q, such that $k_{i'} = k_{i''} = \max_{j \neq q} k_j$, then push(0) into `counter`. The stack `threshold` remains unchanged.

(c) If there is a unique index $\varrho \notin \{p, q\}$ such that $k_\varrho = \max_{j \neq q} k_j$, then push(1) into `counter`. Moreover, let $r = \left\lceil \frac{|T^{(\varrho)}|}{|T^{(p)}|} \right\rceil$. Push($r$) into `threshold`.

(d) If p satisfies $k_p > k_i$ for all $i \notin \{p, q\}$ then let c be the counter at the top of `counter`. If $c = 0$ then `counter` is left unchanged. If $c \geq 1$, then let r be the ratio at the top of `threshold`. Let **C** be the following proposition:

$$\mathbf{C}\text{: there exists } i \notin \{p,q\} \text{ such that } \left\lceil \frac{|T^{(p)}|}{|T^{(i)}|} \right\rceil \leq r.$$

If **C** is satisfied, then c is increased by 1. In any case, `threshold` is left unchanged.

Informally, `counter` aims to solve ambiguous situations in which, when the robot is on its way back to the source, it does not know whether the port-label p is stored into `parent` or it is equal to ϱ. This is done by counting the number of times $p = \varrho$. Actually, one does not count all cases where $p = \varrho$ because this would yield too large counters. To decrease the size of the counters, we count only cases in which there is a unique ϱ and there is another tree of large size. This latter condition is checked by the use of the stack `threshold`. On the way back, the counter is decremented. While the counter is larger than 1, $p = \varrho$. When the counter reaches 1, the robot finds the correct value of p at the top of `parent`.

Table 1 summarizes the operations performed by the robot on its three stacks. **C** is the condition on the ratio of the tree sizes given in (d), and c is the counter at the top of `counter`.

	$c = 0$	$c = 1$	$c > 1$
i' and i''	push(0) in `counter` push(ν) in `parent`	push(0) in `counter` push(ν) in `parent`	push(0) in `counter` push(ν) in `parent`
$p = \varrho$	no action	$\mathbf{C} \Rightarrow c := c + 1$	$\mathbf{C} \Rightarrow c := c + 1$
$p \neq \varrho$	push(1) in `counter` push(r) in `threshold` push(ν) in `parent`	push(1) in `counter` push(r) in `threshold` push(ν) in `parent`	push(1) in `counter` push(r) in `threshold` push(ν) in `parent`

Table 1: Operations performed by $\mathcal{R}$ in state EXPLORE

Once these operations have been performed, $\mathcal{R}$ increases `hop-count` by 1, and leaves through port q, to reach a new node u' from where the exploration carries on in the same way as it was performed at u.

Case 2: $\mathcal{R}$ is in state SUCCESS or FAILURE. Its objective is then to return back to s and to report the success or the failure of the exploration. For that purpose, $\mathcal{R}$ moves back along the edge labeled p. $\mathcal{R}$ will eventually return to s by using the information previously stored in `parent`. We describe bellow how this process is carried out.

Assume that $\mathcal{R}$ enters a node u via port q, in state SUCCESS or FAILURE, with `hop-count` $\neq 0$. $\mathcal{R}$ tries to compute the port p leading toward s. For that purpose, $\mathcal{R}$ repeats the same process as when $\mathcal{R}$ entered u in the state EXPLORE. In particular, $\mathcal{R}$ re-explores the subtrees rooted at u's neighbors. Again, viewing T as rooted at u, let $T^{(i)}$ be the subtree of T rooted at u's neighbor v_i, $i = 1, \ldots, \deg(u)$, and let q_i be the port label at v_i of the link connecting v_i to u. For $i = 1, \ldots, \deg(u)$, $i \neq q$, $\mathcal{R}$ does the following. Starting from $j = 0$, and increasing j by one in case of failure, $\mathcal{R}$ repeats (1) move to v_i, and (2) explore the open subtree $T^{(i)}_{v_i,q_i}$ with $\mathcal{E}_j$. This process is carried on until either there is $k_i \leq k$ for which $\mathcal{E}_{k_i}$ succeeds, or all $\mathcal{E}_j$'s, $0 \leq j \leq k$, fail to explore $T^{(i)}$.

The index p is then obtained as follows:

(α) If there exists $i \neq q$ such that all $\mathcal{E}_j$'s, $0 \leq j \leq k$, fail to explore $T^{(i)}$, then $p = i$.

(β) If there are two distinct indices i' and i'', both different from q, such that $k_{i'} = k_{i''} = \max_{j \neq q} k_j$, then the integer ν at the top of `parent` is popped out, and p is such that $\nu_q(p) = \nu$. Moreover the counter at the top of `counter` is popped out.

(γ) If there is a unique index $\varrho \neq q$ such that $k_\varrho = \max_{j \neq q} k_j$, then the actions depend on the value of the counter c at the top of `counter`.

- $c = 0$. Then set $p = \varrho$.
- $c = 1$. If there exists $i \neq q$ such that $\left\lceil \frac{|T^{(\varrho)}|}{|T^{(i)}|} \right\rceil \leq r$ then pop(1) out of `counter`, pop(ν) out of `parent`, and pop(r) out of `threshold`; p is such that $\nu_q(p) = \nu$. Otherwise, $p = \varrho$.
- $c > 1$. Then set $p = \varrho$. Moreover, if there exists $i \neq q$ such that $\left\lceil \frac{|T^{(\varrho)}|}{|T^{(i)}|} \right\rceil \leq r$ then c is decreased by 1.

Table 2 summarizes the operations performed by the robot on its three stacks. Again, **C** is the condition on the ratio of the tree sizes given in (γ), and c is the counter at the top of `counter`.

	$c = 0$	$c = 1$	$c > 1$
i' and i''	pop(`counter`) p :=pop(`parent`)	not specified	not specified
ϱ and **C**	$p := \varrho$	pop(`counter`) p :=pop(`parent`) pop(`threshold`)	$c := c - 1$ $p := \varrho$
ϱ and $\not\mathbf{C}$	$p := \varrho$	$p := \varrho$	$p := \varrho$

Table 2: Operations performed by $\mathcal{R}$ in state FAILURE or SUCCESS

Once p has been identified, $\mathcal{R}$ decreases `hop-count` by 1, and leaves through port p.

When `hop-count` $= 0$, $\mathcal{R}$ leaves through port ℓ to report the success of the exploration together with the total number of nodes of the open subtree $T_{s,\ell}$ if it is in the state SUCCESS, or the failure of the exploration, if it is in the state FAILURE. This completes the description of $\mathcal{E}_{k+1}$. In the remaining, we show that this protocol satisfies the properties of the lemma.

Proof of correctness. If $T_{s,\ell}$ has at most 2^k nodes, then all explorations of open subtrees rooted at the neighbors v_i's of s, $i \neq \ell$, will succeed for some $\mathcal{E}_{k_i}$, $k_i \leq k$. Therefore, $\mathcal{E}_{k+1}$ will also succeed in this case by application of **I1**.

If $2^k < n \leq 2^{k+1}$, then, since $\mathcal{E}_k$ explores all trees of size at most 2^k, $\mathcal{R}$ can only apply **I1** or **I2** at s. If **I2** is applied, $\mathcal{R}$ moves to the root of the yet unexplored tree. $\mathcal{R}$ proceeds in this way until it reaches a node u^* in which every open subtree $T^{(i)}_{v_i,q_i}$, $i \neq p$, will be successfully explored by some $\mathcal{E}_{k_i}$, $k_i \leq k$. This will eventually occur because every time the robot goes "down" through the yet unexplored tree, the number of nodes of the tree currently explored decreases by at least one. Since all open subtrees $T^{(i)}_{v_i,q_i}$, $i \neq p$, of u^* have been explored, all nodes of the open subtree $T_{s,\ell}$ have been visited. At u^* the robot applies **I1** and enters state SUCCESS.

If $n > 2^{k+1}$, then $\mathcal{R}$ will eventually reach a node u^* where either two subtrees cannot be explored with $\mathcal{E}_k$ or the total number of nodes visited so far exceeds 2^{k+1}. In the former case, the robot applies **I3** and enters state FAILURE. In the latter case, since $\texttt{number} > 2^{k+1}$, the robot enters state FAILURE as well.

It remains to check that the robot returns to s from u^*, and then from s moves through link ℓ.

When **I1** or **I3** is executed, $\mathcal{R}$ still knows the label p of the port leading to s, and the protocol specifies that $\mathcal{R}$ leaves through that port. Now, let u be the current position of $\mathcal{R}$, and assume that $\mathcal{R}$ entered u through port q. $\mathcal{R}$ tries to find the port p leading to s by re-exploring the open subtrees $T^{(i)}$ rooted at u's neighbors. Since **I2** was applied in u when $\mathcal{R}$ was in state EXPLORE, there is at most one open subtree $T^{(i)}$ distinct from $T^{(q)}$ whose exploration may fail with $\mathcal{E}_k$. Indeed, $T_{s,\ell}$ is an open subtree of an unknown tree that can be arbitrarily large, and hence exploring $T^{(p)}$ may fail. However, when $\mathcal{R}$ was in state EXPLORE, it was able to explore all the other open subtrees $T^{(i)}$, $i \notin \{p,q\}$, each with some $\mathcal{E}_{k_i}$, $k_i \leq k$. Hence, if there exists $i \neq q$ such that all $\mathcal{E}_j$'s, $0 \leq j \leq k$, fail to explore $T^{(i)}$, then setting $p = i$ as in (α) is correct. Let us show that the two other settings (β) and (γ) of p are correct as well.

Let us define the *exploration path* of $\mathcal{R}$ in $\mathcal{E}_{k+1}$ as the path $u_0, u_1, \ldots, u_h$ where $u_0 = s$ (i.e., the source), $u_h = u^*$ (i.e., the node where $\mathcal{R}$ switches from the state EXPLORE to the state SUCCESS or FAILURE), and u_{i+1} is the unique node adjacent to u_i, distinct from u_{i-1}, for which all $\mathcal{E}_j$'s, $0 \leq j \leq k$, fail.

$\mathcal{E}_{k+1}$ is designed so that $\mathcal{R}$ proceeds along the exploration path, first forward from s to u^*, and then backward from u^* to s. We prove that $\mathcal{R}$ is able to follow the exploration path backward. First observe that when the robot reaches u^*, it backtracks correctly to u_{h-1}. Moreover, since the states of `parent`, `counter`, and `threshold` were non modified at u^*, $\mathcal{R}$ reaches u_{h-1} with stacks in the same state as when it left u_{h-1}. We prove that this property holds at every node of the exploration path.

Assume $\mathcal{R}$ backtracks along the exploration path up to u_ℓ, $\ell < h$, and that, for every edge $e_i = (u_i, u_{i-1})$, $h \geq i \geq \ell + 1$, of the exploration path, the states of `parent`, `counter`, and `threshold` when $\mathcal{R}$ traverses e_i backward are identical to their states when $\mathcal{R}$ traversed e_i forward.

Claim 4.1 *$\mathcal{R}$ is able to carry on backward to $u_{\ell-1}$. Moreover, the states of* `parent`*,* `counter`*, and* `threshold` *when $\mathcal{R}$ traverses $e = (u_\ell, u_{\ell-1})$ backward are identical to their states when $\mathcal{R}$ traversed e forward.*

To fit with the specification of the protocol $\mathcal{E}_{k+1}$, let p and q be the port-labels at $u = u_\ell$ of edges $(u_\ell, u_{\ell-1})$ and $(u_\ell, u_{\ell+1})$, respectively. We show that p can be identified, and that the stacks `parent`, `counter`, and `threshold` return in the same state as they came.

First assume that there exists $i \neq q$ such that all $\mathcal{E}_j$'s, $0 \leq j \leq k$, fail to explore $T^{(i)}$. Then we already know that $p = i$ is a correct setting. Moreover, since (2), (a) and (α) do not modify any stack, the claim holds in this case.

Next assume that there are two distinct indices i' and i'', both different from q, such that $k_{i'} = k_{i''} = \max_{j \neq q} k_j$. (1) specifies that $\nu_q(p)$ has been pushed into `parent` at u. Hence the setting of p specified by (β) is correct. Moreover (b) specifies that a 0 has been pushed in `counter`. Therefore, since (β) pops out an element from both `parent` and `counter`, $\mathcal{R}$ traverses e backward with stacks in the same states as when it came forward.

Finally, assume that there is a unique index $\varrho \neq q$ such that $k_\varrho = \max_{j \neq q} k_j$.

– If $\varrho \neq p$ then (1) and (c) specify that $\nu_q(p)$ is pushed into `parent`, a 1 is pushed into `counter`, and the ratio $r = \left\lceil \frac{|T^{(\varrho)}|}{|T^{(p)}|} \right\rceil$ is pushed into `threshold`. Hence the case (γ) is considered, with $c = 1$. Clearly, the inequality condition is satisfied, and hence (γ) sets correctly p. Moreover, as in the previous case, (γ) pops out an element in each stack, from which we deduce that $\mathcal{R}$ traverses e backward with stacks in the same states as when it came forward.

– If $\varrho = p$, i.e., $k_p > k_i$ for all $i \notin \{p, q\}$, then (2) specifies that `parent` is left unchanged. Similarly, (d) specifies that `threshold` is left unchanged. If the counter stored at the top of `counter` is null, then (d) lets `counter` unchanged, and the setting of p in (γ) is correct, while none of the stacks is modified. If $c \geq 1$ then the status of `counter` depends on whether the inequality condition **C** stated in (d) is satisfied. Assume first that it is satisfied. Then, by (d), c is incremented by 1, resulting in a counter $c \geq 2$. In this case, (γ) specifies that c must be decremented by 1, and (γ) sets correctly $p = \varrho$. Assume now that condition **C** is not satisfied. By (d), c remains unchanged. The two cases $c = 1$ and $c > 1$ in (γ) insure that p is set correctly to ϱ, and c remains unchanged.

This completes the proof of Claim 4.1.

By Claim 4.1, the robot moves backward along the exploration path. It remains to show that it stops when it reaches s.

Since $\mathcal{R}$ increases (resp. decreases) `hop-count` by one at every move away from (resp. towards) s, $\mathcal{R}$ is back at s when `hop-count` $= 0$. This completes the proof of the first property of the lemma.

The second property follows directly from the fact that if `number` $> 2^{k+1}$ then $\mathcal{R}$ enters state FAILURE.

If $\mathcal{R}$ returns to s in the state SUCCESS, then the whole tree has been explored, and the number of nodes has been accumulated in the variable `number`. Thus the third property is also satisfied, which completes the proof of the lemma. □

We are now ready to present our main algorithm of exploration with return. Given the family $\{\mathcal{E}_k, k \geq 0\}$ described in Lemma 4.1, our protocol can simply be described as follows.

Algorithm `EXPLORE`. The robot starting at node s of a tree T successively executes $\mathcal{E}_k$, $k = 0, 1, \ldots$, until there is a k for which it returns to s in the state SUCCESS. Then it stops.

Theorem 4.3 *Algorithm* `EXPLORE` *accomplishes exploration with return of any n-node tree using $O(\log^2 n)$ bits of memory.*

Proof. By Lemma 4.1, if $k = \lceil \log_2 n \rceil$, then $\mathcal{E}_k$ completes exploration with return of T. It remains to show that `EXPLORE` requires $O(\log^2 n)$ bits of memory for n-node trees.

Assume that $\mathcal{E}_k$ uses M_k bits of memory, and compute the memory requirement for $\mathcal{E}_{k+1}$. First, $\mathcal{E}_{k+1}$ reserves M_k bits of memory for the execution of the $\mathcal{E}_j$'s, $j \leq k$. In addition to that, $\mathcal{E}_{k+1}$ makes use of two counters (`number` and `hop-count`), each on $k+1$ bits since $\mathcal{E}_{k+1}$ does not count more than 2^{k+1} nodes, and does not explore further than 2^{k+1} hops away from the initial position s. $\mathcal{E}_{k+1}$ makes also use of a constant number of integers to store the labels of ports: e.g., the incoming port, the outgoing port, the label of the open subtree currently under consideration, etc. Since labels are between 1 and the degree of the nodes, all these integers are on $\lceil \log_2 n \rceil$ bits. The storage of k_p (if the exploration of $T^{(p)}$ succeeded) requires $O(\log k)$ bits, and the computation of $\nu_q(p)$ requires $O(\log k)$ bits as well. The remaining part of the memory is used to store the stacks.

Claim 4.2 *The stack* `parent` *can be stored using $O(k)$ bits.*

Assume that the stack `parent` stores indices $\nu_1, \dots, \nu_r$ as follows: we use two boolean strings, both of $\sum_{i=1}^{r} \lceil \log_2 \nu_i \rceil$ bits. One is just the concatenation of the binary expressions of the ν_i's, and the other indicates the separations between two successive ν_i's. From the specifications of $\mathcal{E}_{k+1}$, an index ν_i is placed in the stack `parent` every time there exists at least one index $j \notin \{p, q\}$ such that $k_j \geq k_p$. More precisely, if ν_i is stored, then there are at least $\mu_i = \max\{\nu_i, 2\}$ indices $j \neq q$ for which $k_j \geq k_p$. Each such index corresponds to a subtree of size at least $2^{k_p - 1}$, while $T^{(p)}$ is of size at most 2^{k_p}. Hence, if $T \setminus T^{(q)}$ is of size N, then $T^{(p)}$ is of size at most $\frac{2N}{\mu_i + 1}$. Therefore, $\Pi_{i=1}^{r} \frac{\mu_i + 1}{2} \leq 2^{k+1}$, and hence $\sum_{i=1}^{r} \lceil \log_2 \nu_i \rceil = O(k)$. Thus the stack `parent` requires $O(k)$ bits to be stored. This proves Claim 4.2.

Claim 4.3 *The stack* `counter` *can be stored using $O(k)$ bits.*

The stack `counter` consists of a sequence $S = s_1, c_1, s_2, c_2, \dots, s_t, c_t, s_{t+1}$ where the c_i's are counters ≥ 1, and the s_i's are non empty sequences of 0's (apart from s_{t+1} which can be empty). S is coded using two bit strings, both of $\sum_{i=1}^{t} \lceil \log_2 c_i \rceil + \sum_{i=1}^{t+1} |s_i|$ bits, where $|s_i|$ denotes the number of 0's in s_i. One bit string is the concatenation of the elements of the stack, and the other indicates the separation between the fields c_i's and s_i's. According to (b), a 0 is pushed into `counter` whenever there exist two subtrees $T^{(i')}$ and $T^{(i'')}$ of $T \setminus T^{(q)}$ with maximum index k_i. At least one of these subtrees contains as many nodes as 1/2 the size of $T^{(p)}$. Therefore, the total number of 0's in S is $O(k)$, i.e., $\sum_{i=1}^{r+1} |s_i| = O(k)$, and hence $t = O(k)$. Now we bound $\sum_{i=1}^{t} \lceil \log_2 c_i \rceil$.

Each counter c_i of the sequence S corresponds to a sub-path $x_1^{(i)}, \dots, x_{h_i}^{(i)}$ of the exploration path. For $1 \leq i \leq t$, let p_i and ϱ_i be the values at $x_1^{(i)}$ of the current port-labels p and ϱ, respectively. We have $p_i \neq \varrho_i$, a 1 is pushed into `counter`, and the ratio $r_i = \left\lceil \frac{|T^{(\varrho_i)}|}{|T^{(p_i)}|} \right\rceil$ is pushed into `threshold`. Let $\widehat{T}_i = T^{(\varrho_i)}$ at $x_1^{(i)}$, and $T_i = T^{(p_i)}$ at $x_1^{(i)}$. Hence $r_i = \left\lceil \frac{|\widehat{T}_i|}{|T_i|} \right\rceil \leq 2|\widehat{T}_i|/|T_i|$.

Let us consider the last counter c_t of the sequence S. It corresponds to a sub-path $x_1^{(t)}, \dots, x_{h_t}^{(t)}$ of the exploration path. At every node $x_j^{(t)}$, $1 < j \leq h_t$, we have $p = \varrho$. The counter c_t at the top of `counter` is incremented by 1 if the condition **C** is satisfied at $x_j^{(t)}$. Finally, the next node on the exploration path after $x_{h_t}^{(t)}$ does not satisfy $p = \varrho$. Now, if **C** is satisfied at $x_j^{(t)}$, $1 < j \leq h_t$, then there exists some subtree $T^{(j)}$, $j \notin \{p, q\}$, such that $\left\lceil \frac{|T^{(p)}|}{|T^{(j)}|} \right\rceil \leq r_t$ at $x_j^{(t)}$. Thus, $|T^{(j)}| \geq |T^{(p)}|/r_t \geq |\widehat{T}_t|/r$. Hence, there cannot be more than $|T|/(|\widehat{T}_t|/r_t)$ such nodes $x_j^{(t)}$. We get that c_t cannot be incremented more than $r_t|T|/|\widehat{T}_t| \leq 2|T|/|T_t|$ times. Therefore, c_t can be coded using at most $\log|T| - \log|T_t| + 2$ bits.

One can repeat the same arguments for c_{t-1}. It corresponds to a sub-path $x_1^{(t-1)}, \dots, x_{h_{t-1}}^{(t-1)}$ of the exploration path. At every node $x_j^{(t-1)}$, $1 < j \leq h_{t-1}$, we have $p = \varrho$. The counter c_{t-1} at the top of `counter` is incremented by 1 if the condition **C** is satisfied at $x_j^{(t-1)}$. Finally, the next node on the exploration path after $x_{h_{t-1}}^{(t-1)}$ does not satisfy $p = \varrho$. Now, if **C** is satisfied at $x_j^{(t-1)}$, $1 < j \leq h_{t-1}$, then there exists some subtree $T^{(j)}$, $j \notin \{p, q\}$, such that $\left\lceil \frac{|T^{(p)}|}{|T^{(j)}|} \right\rceil \leq r_{t-1}$ at $x_j^{(t-1)}$. Thus, $|T^{(j)}| \geq |T^{(p)}|/r_{t-1} \geq |\widehat{T_{t-1}}|/r$. Hence, there cannot be more than $|T_t|/(|\widehat{T_{t-1}}|/r_{t-1})$ such nodes $x_j^{(t-1)}$. We get that c_{t-1} cannot be incremented more than $r_{t-1}|T_t|/|\widehat{T_{t-1}}| \leq 2|T_t|/|T_{t-1}|$ times. Therefore, c_{t-1} can be coded using at most $\log|T_t| - \log|T_{t-1}| + 2$ bits.

Repeating the same process for all c_i's, we get that the number of bits $\sum_{i=1}^{t}\lceil\log_2 c_i\rceil$ required to encode all the counters c_i's is at most $\sum_{i=1}^{t}(\log|T_{i+1}| - \log|T_i| + 2)$ where $T_{t+1} = T$. Therefore, all counters in `counter` can be encoded using at most $\log|T| + 2t = O(k)$ bits, which completes the proof of Claim 4.3.

Claim 4.4 *The stack* `threshold` *can be stored using $O(k)$ bits.*

The stack `threshold` consists of a sequence $r_1, \ldots, r_t$ of integers. Assume r_i is pushed into `threshold` at node x_i. Let p_i, q_i, and m_i respectively denote the port-labels of the edge leading to the source s, the edge leading to the extremity u^* of the exploration path, and the edge leading to the subtree of maximum index ϱ at x_i. According to (c), we have $r_i = \left\lceil \frac{|T^{(m_i)}|}{|T^{(p_i)}|} \right\rceil$. Again `threshold` is coded using two bit strings, both of $\sum_{i=1}^{t}\lceil\log_2 r_i\rceil$ bits. One bit string is the concatenation of the elements of the stack, and the other indicates the separation between the fields r_i's. We have $\sum_{i=1}^{t}\lceil\log_2 r_i\rceil \leq \sum_{i=1}^{t}(\log|T^{(m_i)}| - \log|T^{(p_i)}| + 2)$. First observe that since $|T^{(m_i)}| \geq |T^{(p_i)}|$, we have $t = O(k)$. Next, observe that since $T^{(m_i)} \subset T^{(p_{i+1})}$, we have $|T^{(m_i)}| \leq |T^{(p_{i+1})}|$. Therefore, $\sum_{i=1}^{t}(\log|T^{(m_i)}| - \log|T^{(p_i)}|) \leq \log|T|$, which completes the proof of Claim 4.4.

In total, from the three previous claims, we have

$$M_{k+1} = M_k + O(k) + O(\log n) = O(k(k + \log n)).$$

Since Algorithm `EXPLORE` involves $\mathcal{E}_k$ for $k \leq \lceil\log_2 n\rceil$, we get that its memory requirement is at most $O(\log^2 n)$, which completes the proof. □

5 Conclusion

The following table is a summary of our results. It gives upper and lower bounds on memory size (in # of bits) of a robot for three types of exploration (perpetual, with stop, with return) of a tree on n nodes, with maximum degree Δ. The upper bound for exploration with stop assumes that the robot knows an upper bound m on the number of nodes of a given degree. Without this knowledge, the best we know is performing exploration with return, using $O(\log^2 n)$ memory bits.

Exploration	Upper Bound	Lower Bound
Perpetual	$O(\log\Delta)$	$\lceil\log\Delta\rceil$
(with) Stop	$O(\log\Delta + \log m)$	$\Omega(\log\log\log n)$
(with) Return	$O(\log^2 n)$	$\Omega(\log n)$

The table shows two interesting open problems. Firstly, what is the exact complexity in terms of memory bits of exploration with return? Secondly, can we perform exploration with stop using stricly less memory space than exploration with return?

Acknowledgements. This work was done during K. Diks' stay at the Université du Québec à Hull, and while P. Fraigniaud was visiting Carleton University in Ottawa. P. Fraigniaud is supported by NATO, and by the EU project DAVID: *Data and Voice Integration over TWDM*, see *http://david.com.dtu.dk*. E. Kranakis is supported in part by NSERC grant. Andrzej Pelc is supported in part by NSERC grant OGP 0008136. The three last authors were supported in part by MITACS (Mathematics of Information Technology and Complex Systems) project CANCOM grant.

References

[1] S. Albers and M. R. Henzinger, Exploring unknown environments, SIAM Journal on Computing 29 (2000), 1164-1188.

[2] N. Alon, Y. Azar, and Y. Ravid, Universal sequences for complete graphs, Discrete Applied Mathematics, 27:25-28, 1990.

[3] B. Awerbuch, M. Betke, R. Rivest and M. Singh, Piecemeal graph learning by a mobile robot, Proc. 8th Conf. on Comput. Learning Theory (1995), 321-328.

[4] L. Babi, N. Nisan, and M.Szegedy, Multiparty protocols and logspace-hard pseudorandom sequences, Proc 21st Ann. ACM Symposium on Theory of Computing (STOC), 1-11, 1989.

[5] E. Bar-Eli, P. Berman, A. Fiat and R. Yan, On-line navigation in a room, Journal of Algorithms 17 (1994), 319-341.

[6] A. Bar-Noy, A. Borodin, M. Karchmer, N. Linial, and M.Werman, Bounds on universal sequences, SIAM J. Computing, 18(2):268-277, 1989.

[7] M.A. Bender, A. Fernandez, D. Ron, A. Sahai and S. Vadhan, The power of a pebble: Exploring and mapping directed graphs, Proc. 30th Ann. Symp. on Theory of Computing (1998), 269-278.

[8] M.A. Bender and D. Slonim, The power of team exploration: Two robots can learn unlabeled directed graphs, Proc. 35th Ann. Symp. on Foundations of Computer Science (1994), 75-85.

[9] P. Berman, A. Blum, A. Fiat, H. Karloff, A. Rosen and M. Saks, Randomized robot navigation algorithms, Proc. 7th ACM-SIAM Symp. on Discrete Algorithms (1996), 74-84.

[10] A. Blum, P. Raghavan and B. Schieber, Navigating in unfamiliar geometric terrain, SIAM Journal on Computing 26 (1997), 110-137.

[11] M. Betke, R. Rivest and M. Singh, Piecemeal learning of an unknown environment, Machine Learning 18 (1995), 231-254.

[12] M. Bridgland, Universal traversal sequences for paths and cycles, Journal of Algorithms, 8(3):395-404, 1987.

[13] J. Buss, and M. Tompa, Lower bounds on universal traversal sequences based on chains of length five, Information and Computation, 120(2):326-329, 1995.

[14] X. Deng, T. Kameda and C. H. Papadimitriou, How to learn an unknown environment I: the rectilinear case, Journal of the ACM 45 (1998), 215-245.

[15] X. Deng and C. H. Papadimitriou, Exploring an unknown graph, Journal of Graph Theory 32 (1999), 265-297.

[16] C.A. Duncan, S.G. Kobourov and V.S.A. Kumar, Optimal constrained graph exploration, Proc. 12th Ann. ACM-SIAM Symp. on Discrete Algorithms (2001), 807-814.

[17] S. Hoory, and A. Widgerson, Universal traversal sequences for expander graphs, Information Processing Letters, 46(2):67-69, 1993.

[18] R. Impagliazza, N. Nisan, A. Widgerson, Pseudorandomness for network algorithms, Proc 26th Ann. ACM Symposium on Theory of Computing (STOC), 356-364, 1994.

[19] S. Istrail, Polynomial universal traversing sequences for cycles are constructible, Proc 20th Annual ACM Symposium on Theory of Computing (STOC), 491-503, 1988.

[20] H. Karloff, R. Paturi and J. Simon, Universal traversal sequences of length $n^{\log n}$ for cliques, Information Processing Letters, 28(5):241-243, 1988.

[21] M. Kouckỳ, Log-space constructible universal traversal sequences for cycles of length $O(n^{4.03})$, Proc. Electronic Colloquium on Computational Complexity, Report TR01-13, `http://www.eccc.uni-trier.de/eccc`, 2001.

[22] M. Kouckỳ, Universal Traversal Sequences with Backtracking, Proc. 16th IEEE Conference on Computational Complexity (2001), 21-26. (Also, to appear in Journal of Computer and System Sciences.)

[23] N. Nisan, Pseudorandom generators for space-bounded computation, Combinatorica, 12(4):449-461, 1992.

[24] P. Panaite and A. Pelc, Exploring unknown undirected graphs, Journal of Algorithms 33 (1999), 281-295.

[25] C. H. Papadimitriou and M. Yannakakis, Shortest paths without a map, Theoretical Computer Science 84 (1991), 127-150.

[26] N. S. V. Rao, S. Hareti, W. Shi and S.S. Iyengar, Robot navigation in unknown terrains: Introductory survey of non-heuristic algorithms, Tech. Report ORNL/TM-12410, Oak Ridge National Laboratory, July 1993.