Tree Exploration with Logarithmic Memory

Leszek Gąsieniec* Andrzej Pelc† Tomasz Radzik‡ Xiaohui Zhang§

Abstract

We consider the task of network exploration by a mobile agent (robot) with small memory. The agent has to traverse all nodes and edges of a network (represented as an undirected connected graph), and return to the starting node. Nodes of the network are unlabeled and edge ports are locally labeled at each node. The agent has no *a priori* knowledge of the topology of the network or of its size, and cannot mark nodes in any way. Under such weak assumptions, cycles in the network may prevent feasibility of exploration, hence we restrict attention to trees. We present an algorithm to accomplish tree exploration (with return) using $O(\log n)$-bit memory for all n-node trees. This strengthens the result from [15], where $O(\log^2 n)$-bit memory was used for tree exploration, and matches the lower bound on memory size proved there.

1 Introduction

We consider the task of network exploration with return, by a mobile entity with small memory. The mobile entity has to traverse all nodes and edges of a network (represented as an undirected connected graph), and return to the starting node. The mobile entity may be a piece of software traveling throughout the network, in the case when the graph represents a computer network, or a mobile robot, if the graph represents an unknown terrain, e.g., a labyrinth. For brevity we will call this mobile entity an *agent*. (automaton).

The task of visiting all nodes of a network is fundamental in searching for data stored at unknown nodes, 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 may be hidden due to security reasons. In other situations, limited sensory capabilities of the agent may prevent it from perceiving such labels. Hence it is important to be able to program the agent 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 ring: without any labels of nodes and without the possibility of marking them, it is clearly impossible to explore a ring 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, attention has to be restricted to a class of graphs that can be explored without marking. An important class of such graphs are connected graphs without cycles, i.e., trees, cf. [15].

In this paper we study the problem of graph exploration under very weak assumptions: we do not assume any labels of nodes, do not assume any *a priori* knowledge about the topology of the graph, its diameter, size, etc., and do not allow marking of nodes in any way. Hence we restrict attention to exploration of trees. Clearly the agent 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 agent 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. No coherence between those local labels is assumed.

In many applications, mobile agents are meant to be simple, often small, and inexpensive devices. This limits the amount of memory with which they can be equipped. Thus it is important to investigate the minimum size of the memory of an agent capable of exploring networks of some class. The size of memory required for network exploration under various scenarios has been studied in [12, 15, 17, 18, 19].

Our results. The main result of this paper is an algorithm to accomplish anonymous tree exploration

*Department of Computer Science, The University of Liverpool, Chadwick Building, Liverpool L69 7ZF, United Kingdom. *Leszek@scs.liv.ac.uk*.

†Département d'informatique, Université du Québec en Outaouais, Gatineau, Québec J8X 3X7, Canada. *Pelc@uqo.ca*. Research partly supported by NSERC discovery grant, and by the Research Chair in Distributed Computing at the Université du Québec en Outaouais.

‡Department of Computer Science, King's College, London WC2R 2LS, United Kingdom. *Tomasz.Radzik@kcl.ac.uk*.

§Department of Computer Science, The University of Liverpool, Chadwick Building, Liverpool L69 7ZF, United Kingdom. *Cloud@csc.liv.ac.uk*.

(with return), using $O(\log n)$-bit memory for all trees of size at most n. An agent executing this algorithm and starting in any node of an arbitrary tree T, decides if T is of size at most n, and if so, traverses all edges of T and returns to the starting node. This strengthens the result from [15], where $O(\log^2 n)$-bit memory was used for anonymous tree exploration, and matches the lower bound $\Omega(\log n)$-bit on memory size proved there.

Related work. Graph exploration by mobile agents (robots) has recently attracted growing attention. The unknown environment in which the agent operates is often modeled as a graph, assuming that the agent may only move along its edges. The graph setting is specified in two different ways. In [1, 7, 8, 14, 17], the agent explores strongly connected directed graphs and it can move only in the direction from tail to head of an edge, not vice-versa. In [4, 9, 12, 16, 18, 19, 27], the explored graph is undirected and the agent can traverse edges in both directions. It is sometimes required that, apart from completing exploration, the agent draw a map of the graph, i.e., output an isomorphic copy of it.

Two alternative efficiency measures are adopted in most papers dealing with exploration of graphs: the time of completing this task, measured by the number of edge traversals by the agent [1, 4, 7, 8, 9, 14, 16], or the number of bits of memory available to the agent [12, 15, 17, 18, 19].

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 agent can recognize, or it is assumed that nodes are anonymous. Exploration of directed graphs assuming the existence of labels was investigated in [1, 14]. In this case no restrictions on the agent 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 [4, 9, 16, 27]. 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 agent, or on studying efficient exploration when moves of the agent are restricted in some way. The first approach was adopted in [27], where an exploration algorithm working in time $e + O(n)$, with n being the number of nodes, was proposed. Restricted agents were investigated in [4, 9, 16]. It was assumed that the agent is a robot with either a restricted tank [4, 9], forcing it to periodically return to the base for refueling, or that it is a tethered robot, 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 agent 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 agent to that of two cooperating agents with a constant number of pebbles. In [7], it was shown that one pebble is enough, if the agent knows an upper bound on the size of the graph, and $\Theta(\log \log n)$ pebbles are necessary and sufficient otherwise.

In [12, 15, 17, 18, 19], the adopted measure of efficiency was the memory size of the agent exploring anonymous graphs. In [17, 19], the agent was allowed to mark nodes by pebbles, or even by writing messages on whiteboards with which nodes were equipped. In [12], the authors studied special schemes of labeling nodes, which facilitate exploration with small memory. In [18], attention was focused on lower bounds on memory size of an agent capable of exploring all graphs of given diameter and given maximum degree.

An even weaker scenario was considered in [15]: nodes do not have labels, no *a priori* knowledge of the graph is assumed, and no node marking is allowed. Since under such weak assumptions the presence of even one cycle may preclude exploration with stop (in fact, such exploration is impossible even in a ring of unknown size), the authors restricted attention to the class of (undirected) graphs in which this task is possible: the class of trees. They considered two related exploration tasks: exploration with stop, in which the robot has to stop in some (unspecified) node after traversing all edges, and exploration with return, in which the robot has to return to the starting node after completing exploration. They proved that $\Omega(\log \log \log n)$ bits of memory are needed for exploration with stop of all trees of size at most n, and that $\Omega(\log n)$ bits are necessary for exploration with return. Moreover, they constructed an algorithm of exploration with return for all trees of size at most n, using $O(\log^2 n)$ bits of memory. Our present scenario is identical as in [15] and our algorithm lowers required memory to the asymptotically optimal size of $O(\log n)$ bits.

We finally mention a vast body of literature more loosely connected to our topic. 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 of a given size. 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 [2]. Explicit UTS are known for 2-regular graphs (cf. [6, 10, 11, 22, 24]), for 3-regular graphs (cf. [5, 21, 26]), for cliques (cf. [3, 23]), and for expanders (cf. [20]). Some of these sequences can be constructed in log-space, and hence can produce perpetual exploration (i.e., exploration without the stop requirement) using compact memory. However, without the *a priori* knowledge of n, none of these constructions allows the agent to return to its original position, or even to stop. Koucký [25] introduced the notion of a universal *exploration* sequence (UXS), i.e., a sequence $q_1, \ldots, q_k$ such that the agent 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 agent 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. Reingold [28] has recently showed that a UXS for general graphs is log-space constructible. However, the knowledge of n is required to allow the agent to stop in the cycle (in the clique, $n = d-1$). Our objective differs in that we want the agent to return to the starting node after exploration, in the absence of any *a priori* knowledge of the size of the network.

2 Terminology and preliminaries

A tree T with locally labeled ports is an undirected tree whose nodes are unlabeled and edges incident to a node v have distinct labels assigned to them. Thus every edge $\{u, v\}$ has two labels, the label of the port at u and the label of the port at v. The labeling is local: there is no relation between labels given to an edge $\{u, v\}$ at u and at v. The labels are drawn from the set $\{0, 1, \ldots, D-1\}$, and the set of the labels of the edges at a node v is denoted by *labels*(v). We assume that $D = O(n)$, where n is the number of nodes in the tree. The i-th label at v is the i-th label in the increasing order of *labels*(v). The next label after the i-th label is the $(i + 1)$-st label, or the first label, if the i-th label is the last one. The i-th edge adjacent to v is the edge with the i-th label at v.

An agent begins at a *starting node* r, and moves along the edges of the tree according to a provided algorithm. When the agent traverses an edge e and enters a node w, it reads into its memory the port label l of edge e at w, the index i of this label at w, and the degree d of w. The algorithm determines then the index j of an edge adjacent to w which should be followed next, or tells the agent to terminate. An algorithm performs exploration with return in a tree, if the agent executing this algorithm from *any* starting node r, traverses all edges of the tree and terminates at node r.

From now on we assume that the port labeling is *symmetric*: for any edge $\{u, v\}$, its labels at u and at v are the same. Symmetric port labelings are equivalent to labeling all edges of the tree in such a way that labels of edges incident to the same node are different. The case of arbitrary port labeling can be reduced to that of symmetric labeling as follows. Every edge $\{u, v\}$ with port labels l_u and l_v is subdivided by a (virtual) node w and replaced by two edges $\{u, w\}$ and $\{w, v\}$ with (symmetric) labels l_u and l_v. An exploration of the subdivided tree yields an exploration of the original tree.

A *walk* in T of length q is a sequence of nodes $(v_0, v_1, \ldots, v_q)$ in T such that for $i = 1, \ldots, q$, nodes v_{i-1} and v_i are adjacent. A walk may contain (or "may pass through") the same node and the same edge more than once. The term *walk* will also mean the sequence of the directed edges $((v_0, v_1), (v_1, v_2), \ldots, (v_{q-1}, v_q))$, and the sequence $(l_1, l_2, \ldots, l_q)$ of the labels of these edges, depending on the context. If a walk contains two occurrences of the same directed edge (v, w), then it must contain edge (w, v) somewhere in between.

An Euler tour of T is a walk in T which traverses each (undirected) edge of T exactly twice. We say that such a walk visits both sides of each edge. The *length* of an n node tree T is the length of its Euler tour, that is, $2(n - 1)$. Any fragment f of an Euler tour in T defines uniquely some shape of the subtree T' of T; see Figure 1. The fragment f may form a complete Euler tour of some subtree T' in T, if each edge in T' is visited in each direction exactly once and no other edge is visited. Otherwise, the fragment f defines a subtree T'' with an *open path* p, i.e., the (simple) non-empty path consisting of the edges visited only once (that is, the edges with only one side visited); see Figure 1. The term *the open path of* f will mean both the open path of T'' and the sequence of labels of the edges on this path, depending on the context. The open path always begins at the node where f starts.

The following simple algorithm performs perpetual exploration of a tree (without stopping requirement), cf. [15, 25], by repeatedly walking along the following Euler tour: the agent leaves the starting node r by the first port. After entering any node by the i-th port, the agent leaves it by the $(i+1)$-st port, or by the 1-st port, if the i-th port is the last one. This way of traversing the tree will be called the *basic (perpetual) walk*. A similar way of traversing the tree is the *inverse basic walk*: after entering any node by the i-th port, the agent leaves it by the $(i-1)$-st port, or by the last port, if the i-th port is the first one.

By performing the basic walk, the agent traverses the whole n-node tree in $2(n-1)$ steps. Hence exploration with return can be easily accomplished with small memory, 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 agent can perform the basic walk for $2(N-1)$ steps, return using the inverse basic walk again for $2(N-1)$ steps, and stop. Counting steps requires $O(\log N)$-bit memory. Thus we have a simple algorithm which uses $O(\log N)$-bit memory and enables the agent to explore with return an arbitrary tree T of size at most N starting from an arbitrary node. This should be contrasted with our result: we construct an algorithm which also uses only $O(\log N)$-bit memory, but enables the agent to *decide* whether T is of size at most N, and if so, to explore T with return. The whole difficulty is not in exploring all sufficiently small trees, but in recognizing when the tree is too large for its capacity of exploration.

The basic walk $t_{v,l}[1..s]$ is the initial segment of length s of the basic perpetual walk starting from node v by the port with label l. It contains s *edge occurrences* and $s+1$ *node occurrences*, and is called a *complete basic walk* of T, if it is an Euler tour of T. We use both terms "edge (node)" and "edge (node) occurrence" when we refer to a walk, choosing the latter, if we want to stress that we refer to a walk rather than a tree. An *incomplete basic walk* is a proper initial segment of a complete basic walk. For the simplicity of presentation we may omit the subscripts v and l if v is the starting node r and the subscript l is the first (lowest) port label at v.

Let $\alpha = (a_0, \dots a_{k-1})$ be a string of integers. The string $rotate(\alpha, x) = (b_0, \dots b_{k-1})$ is said to be the xth *cyclic rotation* of string α, for $x = 0, \dots, k-1$, if $b_i = a_{(i+x) \bmod k}$, for all $i = 0, \dots, k-1$. The concatenation of strings α and β is denoted by $\alpha \circ \beta$, or simply $\alpha\beta$. We also use notation α^k for concatenation of k copies of α, α^R for the string reversed to α, and $length(\alpha)$ for the length of α. A string α is *periodic*, if $\alpha = w^k$ for some w and $k \geq 2$, or equivalently, if $\alpha = rotate(\alpha, x)$ for some $0 < x < length(\alpha)$.

3 Properties of basic walks

Our algorithm keeps increasing an index q and considers those values of q for which the basic walk $t[1..q]$ keeps repeating itself. More concretely, the algorithm checks whether $t[1..3q] = (t[1..q])^3$, and if so, then it considers the possibility that the basic walk $t[1..3q]$ has already traversed the whole tree. If $t[1..3q] = (t[1..q])^3$, then one obvious case is that $t[1..q]$ is the complete walk of the tree. Unfortunately there are other cases as well, and $t[1..q]$ may be only a small initial part of the complete walk. Therefore we need some additional tests to establish if the walk $t[1..3q]$ has already traversed the whole tree and the algorithm may terminate, or it has not and the algorithm has to consider further, larger values of q. The following theorem characterizes the cases which may occur when $t[1..kq] = (t[1..q])^k$, for some $k \geq 3$.

THEOREM 3.1. *Let T be a tree of length at least $q \geq 1$ with the starting node r. Assume that $t[1..q]$ is not periodic and $t[1..kq] = (t[1..q])^k$ for some $k \geq 3$. Then one of the following three cases must hold.*

1. *The length of T is q (that is, $t[1..q]$ is the complete basic walk of T).*
2. *The length of T is $2q$, and the open path (as a sequence of edge labels) defined by $t[1..q]$ is an odd-length palindrome.*
3. *The length of T is greater than kq, and there are two possibly empty strings w and α and two distinct symbols $x \neq y$ such that for $i = 1, 2, \dots k$, the open path defined by $t[1..iq]$ is equal to $w(x\alpha y)^i w^R$.*

Proof. Assume that Case 1 does not hold. Thus the length of T is greater than q and the basic walk $t[1..q]$ is incomplete. Moreover, $t[1..q]$ is not a complete walk of a subtree of T, because if it were, then $t[q+1] \neq t[1]$ and $t[1..kq] \neq (t[1..q])^k$. This means that $t[1..q]$ defines a (non-empty) open path p. Path p cannot be an even length palindrome, because if it were, then there would be two adjacent edges with the same label (the two middle edges on p). Hence p is either an odd length palindrome or it is not a palindrome.

Consider first the case when path p is an odd length palindrome, that is, $p = wfw^R$ for some possibly empty string $w = (l_1, l_2, \dots, l_j)$, $j \geq 0$, and a symbol f. In this case, illustrated in Figure 2, we verify that the basic walk $t[1..2q] = (t[1..q])^2$ returns back to the root. The walk $t[1..q]$ starts at node r, may traverse some subtrees $T_1, \dots, T_a$ rooted at r. Then it leaves r along the first edge of the open path, labeled l_1, and enters a node v. Then it may traverse some subtrees $T_{a+1}, \dots, T_b$ rooted at v before making another step along the open path, and so on. The walk ends at a node r' after possibly traversing some subtrees rooted at r'. See Figure 2.b.

The second part of the walk, $t[q+1..2q]$ follows exactly the same edge labels as the first part since $t[q+1..2q] = t[1..q]$, but starting at node r'. That is, the second part of the walk first traverses subtrees rooted at r' which are isomorphic to subtrees $T_1, \dots, T_a$. Then it leaves node r' along the edge with label l_1, which is the last edge of the open path, and enters a node v'. Then it traverses subtrees rooted at v' which are isomorphic to

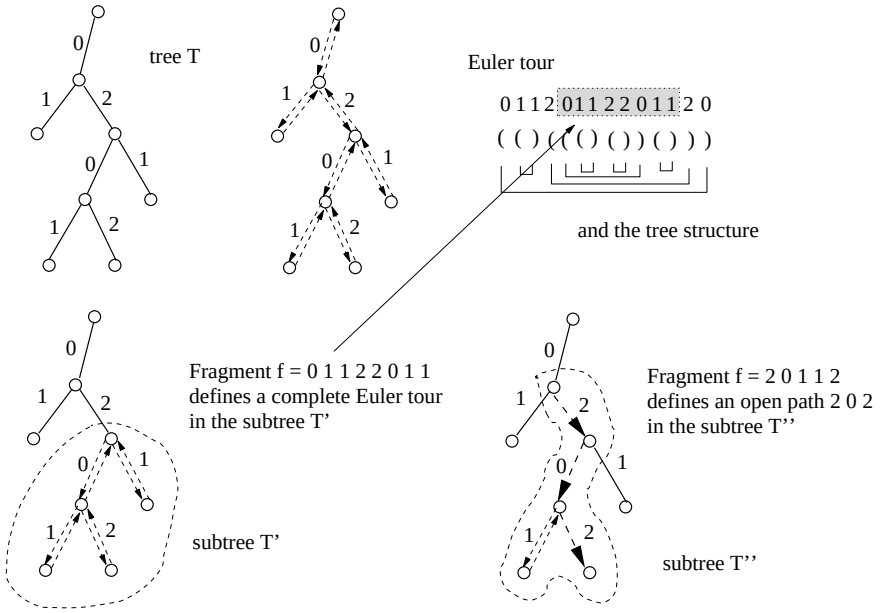

Figure 1: Euler tours and an open path

subtrees $T_{a+1}, \ldots, T_b$, and makes another step back on the open path, and so on. The walk eventually returns to the starting node r. See Figure 2.a.

When the walk $t[1..2q]$ returns back to r, all edges adjacent to node r have been already visited, because $t[2q+1] = t[1]$. This means that $t[1..2q]$ is the complete basic walk of T, and Case 2 of the theorem holds.

Consider now the case when the open path p is not a palindrome, that is, $p = wx\alpha yw^R$ for some possibly empty strings w and α and two symbols $x \neq y$. In this case $(t[1..q])^2$ is still an incomplete basic walk of T, and it defines a subtree of T with the open path $w(x\alpha y)^2w^R$; see Figure 3.a. Generally, considering the subsequent repetitions of $t[1..q]$ on the basic walk $(t[1..q])^k$, one can show that for $i = 1, 2, \ldots, k$, $(t[1..q])^i$ is an incomplete basic walk of T, and it defines a subtree of T with the open path $w(x\alpha y)^iw^R$. See Figure 3.b. Thus the length of T is greater than kq and Case 3 of the theorem holds.

If Case 2 of Theorem 3.1 holds, that is, if $(t[1..q])^2$ is the complete basic walk of T, then T is called a *symmetric tree*. If Case 3 holds, then the sequence of edges corresponding to the $(x\alpha y)^i$ part of the open path defined by $t[1..iq]$ is called the *backbone of the open path*. In the remaining part of this section we consider an index $q \geq 1$, and we assume that the length of tree T is at least q, $t[1..3q] = (t[1..q])^3$ and $t[1..q]$ is not periodic. The correctness of our algorithm is based on its ability to distinguish, under these assumptions, which of the three cases of Theorem 3.1 occurs. We do not consider the case when the basic walk $t[1..q]$ is periodic, that is, the case when $t[1..q] = t[1..q']^j$ for some non-periodic $t[1..q']$ and $j \geq 2$, because our algorithm handles such a case when it checks the index $q' < q$. In what follows, "Case i" means "Case i of Theorem 3.1."

For each node occurrence v on the basic walk and each label $l \in \mathit{labels}(v)$ of an edge adjacent to the tree node v, let $0 \leq s_l^v \leq q-1$ be such that $t_{v,l}[1..q] = \mathit{rotate}(t[1..q], s_l^v)$, which is equal to $t[s_l^v + 1..s_l^v + q]$. If $t_{v,l}[1..q]$ is not a cyclic rotation of $t[1..q]$, then s_l^v is not defined. If $t_{v,l}[1..q]$ is a cyclic rotation of $t[1..q]$, then s_l^v is uniquely defined, or otherwise $t[1..q]$ would have to be periodic (if $t_{v,l}[1..q] = \mathit{rotate}(t[1..q], x') = \mathit{rotate}(t[1..q], x'')$ for some $0 \leq x' < x'' < q$, then $t[1..q] = \mathit{rotate}(t[1..q], x'' - x')$, so $t[1..q]$ is periodic). If s_l^v is defined for each $l \in \mathit{labels}(v)$, then the sequence $\langle s_l^v \rangle_{l \in \mathit{labels}(v)}$ is called the *signature* of node v.

It is not difficult to see that if $t[1..q]$ is the complete basic walk, then each node of T, and each node occurrence on $t[1..q]$, has a well defined signature. One can also show that if $t[1..2q]$ is the complete basic walk (tree T is symmetric), then each node of T, and each node occurrence on $t[1..q]$, has a well defined signature.

LEMMA 3.1. *In Cases 1 and 2, the signature of every node on the basic walk $t[1..q]$ is well defined.*

Our algorithm checks if every node encountered on the basic walk $t[1, ..q]$ has a well defined signature. However, even if this test passes, we may still have Case 3, so further tests are needed.

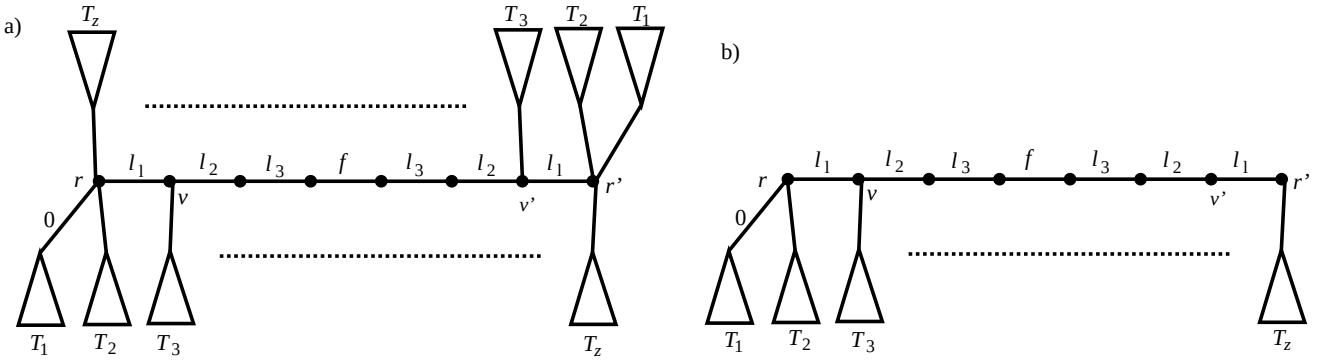

Figure 2: Example for Theorem 3.1 Case 2 – symmetric tree. a) A tree T with the complete basic walk $t[1..2q] = (t[1..q])^2$. b) The subtree of T defined by the incomplete basic walk $t[1..q]$. The sequence of labels on the open path is the odd-length palindrome $(l_1, l_2, l_3, f, l_3, l_2, l_1)$. The edges adjacent to the same node are drawn counter-clockwise according to increasing labels.

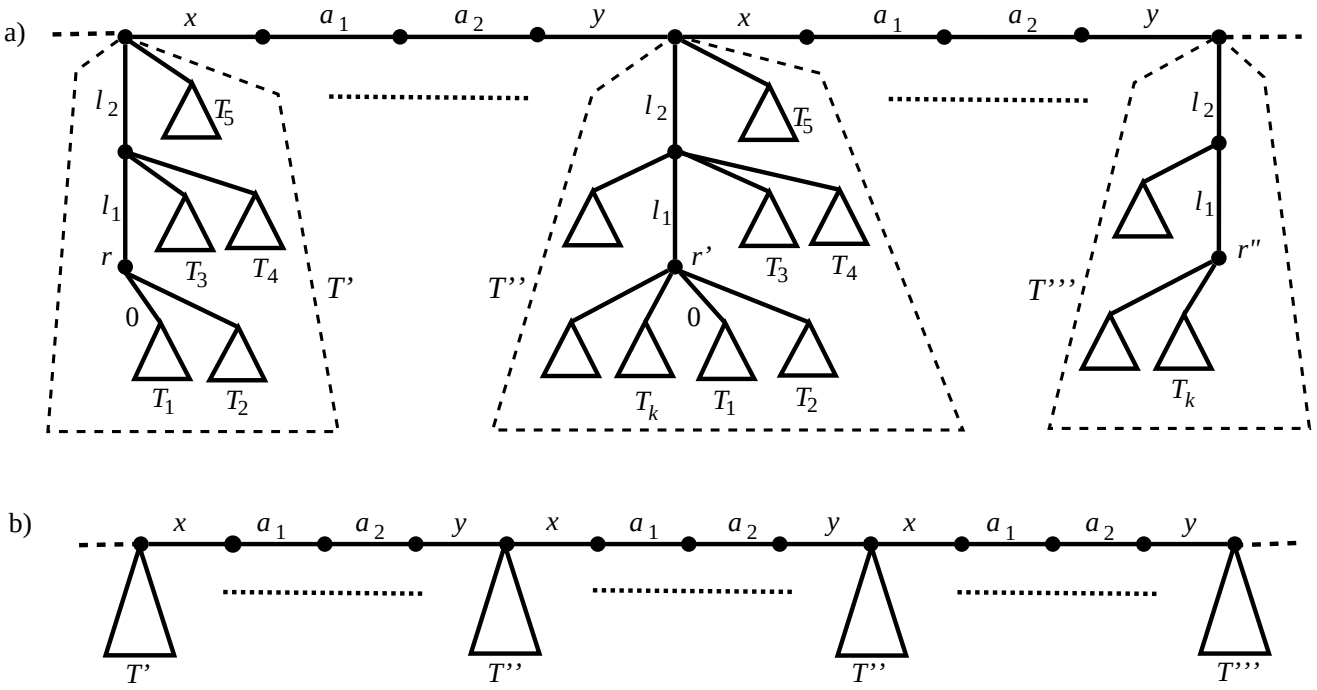

Figure 3: Example for Theorem 3.1, Case 3. The open path of the incomplete basic walk $t[1..q]$ is $(l_1, l_2, x, a_1, a_2, y, l_2, l_1)$, $x \neq y$. a) The subtree defined by $(t[1..q])^2$; the open path is $(l_1, l_2) \circ (x, a_1, a_2, y)^2 \circ (l_2, l_1)$. b) The subtree defined by $(t[1..q])^3$; the backbone of the open path is $(x, a_1, a_2, y)^3$. Tree T'' can be viewed as obtained from trees T' and T''' by joining them along the paths from (l_1, l_2).

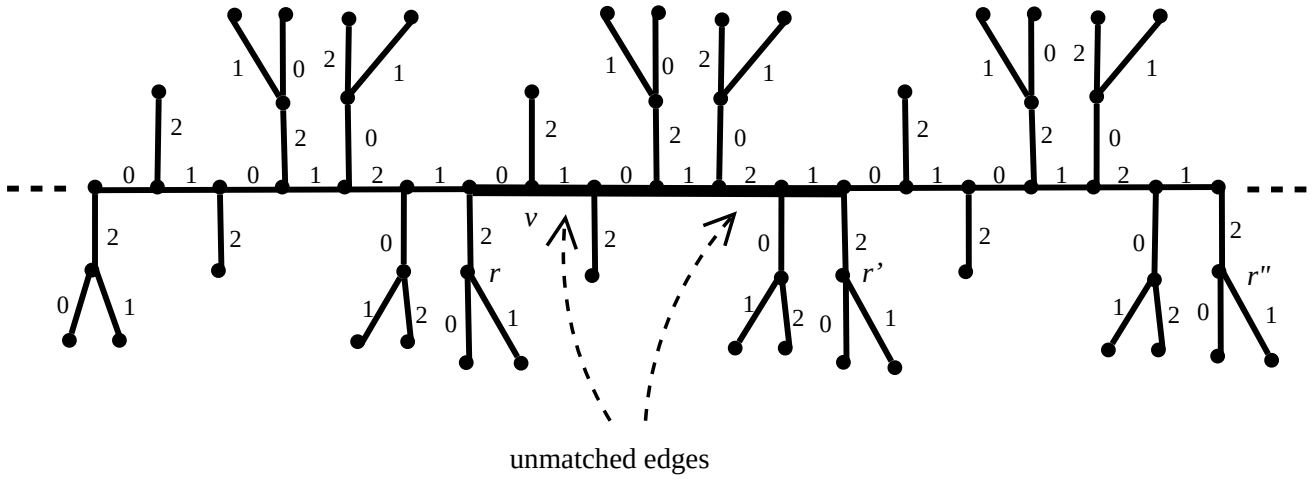

Figure 4: The signatures of the nodes defined by an incomplete basic walk. The basic walk starting from r repeats periodically $t[1..20] = 00112012201201122012$. The sequence of the labels on the backbone has period 010121. The signature of node v is $\{(0,9),(1,6),(2,7)\}$. Indicating the pairs of matching edges with parentheses and underlying the unmatched edges, the basic walk $t[1..20]$ has the following structure: $(00)(11)(2(0\underline{1}(22)0)(1\underline{2}(0(11)(22)0)1)2)$.

We say that two distinct edge occurrences (v,w) and (w',v') on the basic walk $t[1..q]$ *match*, if they have the same labels, node occurrences v and v' have the same defined signatures, and node occurrences w and w' have the same defined signatures.

LEMMA 3.2. *Each edge on the basic walk $t[1..q]$ may have at most one matching edge.*

Proof. If the i-th edge and the j-th edge on $t[1..q]$, for some $1 \leq i < j \leq q$, were both matching edges for the same edge (v,w), then $t[1..q] = \mathit{rotate}(t[1..q], j-i)$, so $t[1..q]$ would be periodic.

LEMMA 3.3. *In Case 1, each edge in the basic walk $t[1..q]$ has a matching edge. In Case 2, the middle edge of the open path defined by $t[1..q]$ is the only edge on $t[1..q]$ without a matching edge.*

Proof. In Case 1, for each edge $\{v,w\}$ in T, the two occurrences (v,w) and (w,v) of this edge in $t[1..q]$ match.

In Case 2, if an edge (v,w) on the basic walk $t[1..q]$ is not on the open path, then there must be an edge (w,v) on $t[1..q]$, and these two edges match. We consider now the edges on the open path. The open path is a palindrome in terms of edge labels. One can check that the open path is also a palindrome in terms of the node signatures. The first node on the open path has the same signature as the last node, the second node has the same signature as the last but one node, and so on. This implies that an edge (v',w') on the open path on one side of the middle edge matches the symmetrical edge (w'',v'') on the other side, while the middle edge does not have the matching edge.

In the further analysis of Case 3, we will use the following Lemma.

LEMMA 3.4. *Consider Case 3 of Theorem 3.1 and assume that each node occurrence on the basic walk $t[1..q]$ has a well defined signature. If (a,b) and (b',a') are matching edges on $t[1..q]$, then either both or none of them belong to the backbone of the open path.*

Proof. The following claim implies this lemma. If v and u are two node occurrences on $t[1..q]$ such that v belongs to the open path but u does not, then these nodes must have different signatures. To show that this claim is true, consider the labels g and h of the two edges on the backbone of the open path adjacent to v. No prefix of the basic walk $t_{v,g}[1..q]$ is a complete walk of a subtree of T, and similarly no prefix of the basic walk $t_{v,h}[1..q]$ is a complete walk of a subtree of T. On the other hand, since each but one subtree of u has length less than q, for each but one label l adjacent to u, $t_{u,l}[1..q]$ has a prefix which is a complete walk of a subtree of T. Thus, if there are two edges with labels g and h adjacent to u, then $t_{v,g}[1..q] \neq t_{u,g}[1..q]$ or $t_{v,g}[1..q] \neq t_{u,g}[1..q]$. Hence v and u must have different signatures.

We say that the edge matching relationship satisfies the *nesting property*, if the following condition holds.

For every two pairs of indices $1 \leq i < i' \leq q$ and $1 \leq j < j' \leq q$, if the edges at positions i and i' in $t[1..q]$ match, the edges at positions j and j' match, and $i < j$, then either $i < j < j' < i'$ or $i < i' < j < j'$. In Cases 1 and 2 of Theorem 3.1, the edge matching relationship satisfies the nesting property. The final point in our analysis of Case 3 is that if all node occurrences on the basic walk $t[1..q]$ have well defined signatures and the edge matching relationship satisfies the nesting property, then $t[1..q]$ must contain at least two unmatched edges. Figure 4 shows an example of a Case-3 tree with indicated unmatched edges.

LEMMA 3.5. *In Cases 1 and 2 of Theorem 3.1, the matching relationship has the nesting property.*

Proof. The proof in trees of length q is straightforward. In trees of length $2q$ the only problem may appear on edges belonging to the open path defined by $t[1...q]$. However since we know that the open path is a palindrome with symmetrically matched edges, the nesting property is also satisfied for those edges.

LEMMA 3.6. *In Case 3 of Theorem 3.1, if each node on the basic walk $t[1..q]$ has a well defined signature, and the matching relationship has the nesting property, then $t[1..q]$ has at least two unmatched edges.*

Proof. Let $wx\alpha yw^R$ be the sequence of labels on the open path defined by the basic walk $t[1..q]$, as given in Theorem 3.1. The sequence of labels on the open path defined by $t[1..3q]$ is $w(x\alpha y)^3w^R$. Let (a, b) and (c, d) be the first edge and the last edge of the backbone of the open path in $t[1..q]$. The labels of these edges are x and y, respectively.

If edge (a, b) has a matching edge (b', a'), then the definition of matching edges and Lemma 3.4 imply that (b', a') must belong to the open path and must be different than edge (c, d) (edge (b', a') has label x while edge (c, d) has label y). Moreover, Lemma 3.4 and the nesting property imply that for any edge (v, u) on the open path between edges (a, b) and (b', a'), if this edge has the matching edge (u', v'), then (u', v') must be on the open path between edges (a, b) and (b', a'). Thus there must be an unmatched edge (a'', b'') on the open path between edges (a, b) and (b', a'), since if all edges between (a, b) and (b', a') were matched, then a pair of inner-most matched edges would be a pair of adjacent edges with the same label (while adjacent edges must have different labels).

If (c, d) has a matching edge (d', c'), then (d', c') must be on the open path and, because of the nesting property, it must be between edges (b', a') and (c, d). Similarly as above, we can conclude that there must be an unmatched edge (c'', d'') between edges (d', c') and (c, d). Thus there must be at least two unmatched edges on $t[1..q]$: (a, b) or (a'', b''), and (c, d) or (c'', d'').

4 The algorithm

Our algorithm LOGEXPLORATION(N) follows from the analysis of the periodic basic walks presented in the previous section. The algorithm keeps increasing index q from 1 to N, where N is a given bound on the length of trees which are to be explored successfully, and checks for each q if Case 1 or 2 of Theorem 3.1 occurs. To check this, the algorithm tests (i) if all node occurrences on the basic walk $t[1..q]$ have well defined signatures (procedure NODESIGNATUREOK), (ii) if the matching relationship on the edges on $t[1..q]$ has the nesting property (procedure NESTINGOK), and (iii) if the number of the unmatched edges on the basic walk $t[1..q]$ (counted by procedure UNMATCHEDEDGES) is at most 1.

Algorithm LOGEXPLORATION(N):
  $q := 1$;
  while $q \leq N$ **do**
    { invariant: the length of the tree is at least q }
    if $t[1..q]$ is not periodic
      and $t[1..3q] = (t[1..q])^3$
      and NODESIGNATUREOK
      and NESTINGOK
      and UNMATCHEDEDGES ≤ 1
    then terminate; { the tree is explored }
    else $q := q + 1$;
  end_while

All tests performed by algorithm LOGEXPLORATION can be implemented on the basis of the procedure GETSYMB(i, k, z) with the following specification. It returns the label of the z-th edge on the basic walk which begins from node i on $t[1..q]$ leaving this node via the k-th port. The agent is at the starting node r at the beginning of this procedure, and returns to r at the end of the procedure. Observe that GETSYMB$(0, 1, z)$ returns the label of the z-th edge of the basic walk $t[1..q]$.

Procedure GETSYMB(i, k, z):
  { the agent is at the starting node r }
  i steps of basic walk starting via the first port;
  z steps of basic walk starting via the k-th port;
  $l :=$ the label of the last edge;
  reverse all steps;
  { the agent is back at the starting node r }
  return l;

Procedure NodeSignatureOK uses procedure SignOK(i) which checks if node i on $t[1..q]$ has a well defined signature, $0 \le i \le q$. Procedure Deg(i) returns the degree of node i by following the basic walk from the start node r for i steps, recording the degree of the current node, and reversing the steps to return to r. Procedure IsSignature(i, k, x) checks if the k-th coordinate of the signature of node i is equal to x. That is, denoting by l_k the k-th label at node i, it checks if $t_{i,l_k}[1..q] = t[x+1..x+q]$. Procedure FindSignature(i, k) returns the k-th coordinate of the signature of node i, or -1, if not defined.

Procedure NodeSignatureOK:
  for $i := 0$ **to** q **do**
    if not SignOK(i) **then return** $false$;
  return $true$;

Procedure SignOK(i):
  $d :=$ Deg(i);
  for $k := 1$ **to** d **do**
    if FindSignature(i, k) $= -1$ **then return** $false$;
  return $true$;

Procedure FindSignature(i, k):
  for $x := 0$ **to** $q - 1$ **do**
    if IsSignature(i, k, x) **then return** x;
  return -1;

Procedure IsSignature(i, k, x):
  for $z = 1$ **to** q **do**
    if GetSymb$(0, 1, x + z) \neq$ GetSymb(i, k, z)
    then return $false$;
  return $true$;

Procedures NestingOK and UnmatchedEdges use procedure MatchingEdge(i), which for edge i on $t[1..q]$, finds its matching edge. It returns -1, if edge i does not have the matching edge.

Procedure NestingOK:
  for $left := 1$ **to** q **do**
    $right :=$ MatchingEdge($left$);
    if $right > left$
      for $i = left + 1$ **to** $right - 1$ **do**
        if MatchingEdge(i)
          $\notin \{left + 1, \ldots, right - 1) \cup \{-1\}$
        then return $false$;
  return $true$;

Procedure UnmatchedEdges:
  $unmatched := 0$;
  for $i := 1$ **to** q **do**
    if MatchingEdge(i) $= -1$ **then** $unmatched{+}{+}$;
  return($unmatched$);

Procedure MatchingEdge uses procedure SignatureEq(i, j), which checks if nodes i and j on the basic walk $t[1..q]$ have the same signatures and is based on procedure FindSignature.

Procedure MatchingEdge(i):
  $l :=$ GetSymb$(0, 1, i)$; { i-th edge label on $t[1..q]$ }
  for $j := 1$ **to** q **do**
    if $j \neq i$ **and** GetSymb$(0, 1, j) = l$
      and SignatureEq$(i - 1, j)$
      and SignatureEq$(i, j - 1)$
    then return j;
  return(-1);

Procedure SignatureEq(i, j):
  $deg :=$ Deg(i);
  if $deg \neq$ Deg(j) **then return** $false$;
  for $k := 1$ **to** deg **do**
    if FindSignature(i, k) $\neq$ FindSignature(j, k)
    then return $false$;
  return($true$);

Checking if $t[1..q]$ is periodic and if $t[1..3q] = (t[1..q])^3$ can be easily implemented using procedure GetSymbol.

The correctness of algorithm LogExploration (properties 1 and 2 in the theorem below) follows from the analysis given in Section 3. The whole algorithm can be viewed as a fixed number of nested loops. The index q of the outermost loop changes from 1 to N, while the ranges of the inner loops are from 1 to $O(q)$. Thus the algorithm can be implemented using a fixed number of $O(\log N)$-bit counters plus a fixed number of additional $O(\log D)$-bit variables for storing the labels and indices of edges.

Theorem 4.1. *Algorithm* LogExploration(N), *executed in an arbitrary tree starting at an arbitrary node, has the following properties.*

1. *When the algorithm terminates with $q \le N$, then the whole tree has been explored.*
2. *If the tree has length $L \le N$, then the algorithm terminates in the iteration when $q = L/2$, if the tree is symmetric, or when $q = L$, if the tree is not symmetric.*
3. *The algorithm uses $O(\log N)$ memory bits.*

Acknowledgement. We would like to thank Darek Kowalski, Gadi Landau and David Peleg for helpful discussion during the early stages of this work.

References

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

[2] R. Aleliunas, R. M. Karp, R. J. Lipton, L. Lovász and C. Rackoff, Random walks, universal traversal sequences, and the complexity of maze problems, Proc. 20th Ann. Symp. on Foundations of Computer Science (FOCS 1979), 218-223.

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

[4] 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.

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

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

[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 (STOC 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 (FOCS 1994), 75-85.

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

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

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

[12] R. Cohen, P. Fraigniaud, D. Ilcinkas, A. Korman, D. Peleg, Label-guided graph exploration by a finite automaton, Proc. 32nd International Colloquium on Automata, Languages and Programming (ICALP 2005), LNCS 3580, 335-346.

[13] 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.

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

[15] K. Diks, P. Fraigniaud, E. Kranakis, A. Pelc, Tree exploration with little memory, Journal of Algorithms 51 (2004), 38-63.

[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] P. Fraigniaud, D. Ilcinkas, Digraphs exploration with little memory, Proc. 21st Annual Symposium on Theoretical Aspects of Computer Science (STACS 2004), LNCS 2996, 246-257.

[18] P. Fraigniaud, D. Ilcinkas, G. Peer, A. Pelc, D. Peleg, Graph exploration by a finite automaton, Theoretical Computer Science 345 (2005), 331-344.

[19] P. Fraigniaud, D. Ilcinkas, S. Rajsbaum, S. Tixeuil, Space lower bounds for graph exploration via reduced automata, Proc. 12th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2005), LNCS 3499, 140-154.

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

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

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

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

[24] M. Koucký, 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.

[25] M. Koucký, Universal Traversal Sequences with Backtracking, Proc. 16th IEEE Conference on Computational Complexity (2001), 21-26.

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

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

[28] O. Reingold, Undirected ST-connectivity in log-space, Proc 37th Annual ACM Symposium on Theory of Computing (STOC 1998), 376-385.