Tree Exploration in Dual-Memory Model

Dominik Bojko
Department of Fundamentals of Computer Science,
Wroclaw University of Science and Technology, Poland

Karol Gotfryd
Department of Fundamentals of Computer Science,
Wroclaw University of Science and Technology, Poland

Dariusz R. Kowalski
School of Computer and Cyber Sciences, Augusta University, GA, USA

Dominik Pająk
Department of Pure Mathematics, Wroclaw University of Science and Technology,
Infermedica, Poland

Abstract

We study the problem of online tree exploration by a deterministic mobile agent. Our main objective is to establish what features of the model of the mobile agent and the environment allow linear exploration time. We study agents that, upon entering a node, do not receive as input the edge via which they entered. In such model, deterministic memoryless exploration is infeasible, hence the agent needs to be allowed to use some memory. The memory can be located at the agent or at each node. The existing lower bounds show that if the memory is either only at the agent or only at the nodes, then the exploration needs superlinear time. We show that tree exploration in dual-memory model, with constant memory at the agent and logarithmic in the degree at each node is possible in linear time when one of the two additional features is present: fixed initial state of the memory at each node (so called clean memory) or a single movable token. We present two algorithms working in linear time for arbitrary trees in these two models. On the other hand, in our lower bound we show that if the agent has a single bit of memory and one bit is present at each node, then the exploration may require quadratic time even on paths, if the initial memory at nodes could be set arbitrarily (so called dirty memory). This shows that having clean node memory or a token allows linear time exploration of trees in the dual-memory model, but having neither of those features may lead to quadratic exploration time even on a simple path.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis; Theory of computation → Online algorithms

Keywords and phrases Graph exploration, agent, memory, tree, deterministic algorithms, lower bound

Digital Object Identifier 10.4230/LIPIcs.MFCS.2022.22

Related Version *Full Version*: `https://arxiv.org/abs/2112.13449` [6]

Funding *Dominik Pająk*: Supported by Polish National Science Centre grant UMO-2019/33/B/ST6/02988.

1 Introduction

Consider a mobile entity deployed inside an undirected graph with the objective to visit all its nodes without any a priori knowledge of the topology or size of the graph. This problem, called a graph exploration, is among the basic problems investigated in the context of a mobile agent in a graph [1, 10, 24, 26, 33] and has applications to robot navigation [35] and searching the World Wide Web [7]. In this paper we focus on tree exploration by a deterministic agent. Clearly, to explore the whole tree of n nodes, any agent needs time $\Omega(n)$. A question arises:

"What are the minimum agent capabilities required to complete the tree exploration in linear time?" Many practical applications may not allow the agent to easily backtrack its moves due to technical, security or privacy reasons. Thus, in this work we assume that the agent, upon entering a node, does ***not*** receive any information about the edge via which it entered.

We consider three components of the input to the agent: memory at the agent, memory at each node, and movable tokens (cf. e.g. [4, 3, 10, 11]). A deterministic agent given some input always chooses the same outgoing edge. Hence at a node of degree d needs at least d different inputs in order to be able to choose all of d outgoing edges. Thus in order to ensure the sufficient number of possible inputs, the number of possible values of the memory and possible present or absent states of the tokens, must exceed d. Hence, the sum of the memory at the agent, memory at the node and the number of tokens must be at least $\lceil \log d_v \rceil$ bits, for any node of degree d_v, where log denotes base-2 logarithm (cf. [34, Observation 1.1]).

In many applications, mobile agents are meant to be small, simple and inexpensive devices or software, cf. [11, 24]. These restrictions limit the size of the memory with which they can be equipped. Thus it is crucial to analyse the performance of the agent equipped with the minimum size of the memory. In this paper we assume the asymptotically minimum necessary memory size of $O(\log d_v)$, where only a constant number of bits is stored at the agent and logarithmic number (in the degree) at each node.

We consider two additional features of the model, namely – clean memory or a single token, and we show that each of these assumptions alone allows linear-time tree exploration. We also prove that when both features are absent, then the linear time exploration with small memory may be impossible even on a simple path.

2 Model

The agent is located in an initially unknown tree $T = (V, E)$ with $n = |V|$ nodes, where n is not known to the agent. The agent can traverse one of the edges incident to its current location within a single step, in order to visit a neighbouring node at its other end. The nodes of the tree are unlabelled, however in order for the agent to locally distinguish the edges outgoing from its current position, we assume that the tree is port-labelled. This means that at each node v with some degree d_v, its outgoing edges are uniquely labelled with numbers from $\{1, 2, \ldots, d_v\}$. Throughout the paper we assume that the port labels are assigned by an adversary that knows the algorithm used by the agent in advance and wants to maximize the exploration time.

Memory. The agent is endowed with some number of memory bits (called *agent memory* or *internal memory*), which it can access and modify. In our results, we assume that the agent has $O(1)$ bits of memory.

Each node $v \in V$ contains some number of memory bits as well, which can be modified by the agent when visiting that node. We will call these bits *node memory* or *local memory*. We assume that node v contains $\Theta(\log d_v)$ bits of memory. Hence, each node is capable of storing a constant number of pointers to its neighbours in the tree.

General Exploration Algorithm. Upon entering a node v, the agent a receives, as input, its current state s_a, the state s_v of the current node v, the number tok_v of tokens at v, the number tok_a of tokens at the agent and the degree d_v of the current node. It outputs its new state s'_a, new state s'_v of node v, new state $\mathrm{tok}'_a, \mathrm{tok}'_v$ of the tokens at the agent/node, respectively, and port p_{out} via which it exits node v; hence, the algorithm defines a transition:

$$(s_a, s_v, \mathrm{tok}_a, \mathrm{tok}_v, d_v) \to (s'_a, s'_v, \mathrm{tok}'_a, \mathrm{tok}'_v, p_{\mathrm{out}}),$$

that must satisfy $\mathrm{tok}_a + \mathrm{tok}_v = \mathrm{tok}'_a + \mathrm{tok}'_v$. In models without the tokens, the state transition of the algorithm can be simplified to:

$$(s_a, s_v, d_v) \to (s'_a, s'_v, p_{\mathrm{out}})$$

Starting state. We assume that for a given deterministic algorithm, there is one starting state s of the agent (obtained from the agent memory), in which the agent is at the beginning of (any execution of) the algorithm. The agent can also use this state later (it can transition to s in subsequent steps of the algorithm). The starting location of the agent is chosen by an adversary accordingly to agent's algorithm (to maximize the agent's exploration time).

Our models. In this paper we study the following three models:

In CleanMem, there is a fixed state $\hat{s}$ and each node v is initially in the state $s_v = \hat{s}$, regardless of the degree of the vertex. In this model the agent does not have access to any tokens.

In DirtyMem, the memory at the nodes is in arbitrary initial states, which are chosen by an adversary. In this model the agent does not have access to any tokens.

In Token, the agent is initially equipped with a single token. If the agent holds a token, it can drop it at a node upon visiting that node. When visiting a node, the agent receives as input whether the current node already contains a token. In this case the agent additionally outputs whether it decides to pick up the token, i.e., deduct from tok_v and add to tok_a. Clearly, since the agent has only one token, then after dropping it at some node, it needs to pick it up before dropping it again at some other node. In this model we assume that the initial state of the memory at each node is chosen by an adversary (like in DirtyMem model).

3 Our results

Upper bounds. While a lot of the existing literature focuses on feasibility of exploration, we show that it is possible to complete the tree exploration in the minimum possible linear time using (asymptotically) minimal memory. We show two algorithms in models CleanMem and Token, exploring arbitrary unknown trees in the optimal time $O(n)$ if constant memory is located at the agent and logarithmic memory is located at each node. Our results show that in the context of tree exploration in dual-memory model, the assumption about clean memory (fixed initial state of node memory) can be "traded" for a token.

It is worth noting that in both our algorithms, the agent returns to the starting position and terminates after completing the exploration, which is a harder task than a perpetual exploration. If the memory is only at the agent, exploration of trees with stop at the starting node requires $\Omega(\log n)$ bits of memory [14]; if the agent is only required to terminate at any node then still a superconstant $\Omega(\log \log \log n)$ memory is required [14]; while exploration without stop is feasible using $O(\log \Delta)$ agent memory [14] (where Δ is the maximum degree of a node). All these results assume that the agent could receive the port number via which it entered the current node, while our algorithms operate without this information.

Lower bound. To explore a path with a single bit of memory and with arbitrary initial state at each node (note that $\log d_v = 1$ for internal nodes on the path), one can employ the Rotor-Router algorithm and achieve exploration time of $O(n^2)$ [41] (there is no need for agent memory), which is time and memory optimal in the model with only node memory [34].

We want to verify the hypothesis that dual-memory allows to achieve linear time of exploration of trees. In our lower bound we analyse path exploration with one bit at each node and additional one bit of memory at the agent. We prove that in this setting, in DirtyMem model an exploration of the path requires $\Omega(n^2)$ steps in the worst case. Interestingly, since time $O(n^2)$ in already achievable with only a single bit at each node (and no memory at the agent), our lower bound shows that adding a single bit at the agent does not reduce the exploration time significantly.

3.1 Previous and related work

Only node memory. One approach for exploration using only node memory is Rotor-Router [41]. It is a simple strategy, where upon successive visits to each node, the agent is traversing the outgoing edges in a round-robin fashion. Its exploration time is $\Theta(mD)$ for any graph with m edges and diameter D [2, 41]. It is easy to see that this algorithm can be implemented in port-labelled graphs with zero bits of memory at the agent and only $\lceil \log d_v \rceil$ bits of memory at each node v with degree d_v (note that this is the minimum possible amount of memory for any correct exploration algorithm using only memory at the nodes). Allowing even unbounded memory at each node still leads to $\Omega(n^3)$ time for some graphs, and $\Omega(n^2)$ time for paths [34]. These lower bounds hold even if the initial state of the memory at each node is clean (i.e., each node starts in some fixed state $\hat{s}$ like in CleanMem model). Hence, only node memory is insufficient for subquadratic time exploration of trees.

Only agent memory. When the agent is not allowed to interact with the environment, then such agent, when exploring regular graphs, does not acquire any new information during exploration. Hence such an algorithm is practically a sequence of port numbers that can be defined prior to the exploration process. In the model with agent memory, the agent is endowed with some number of bits of memory that the agent can access and modify at any step. Thus for the case of the path, the lower bound (for unlimited agent memory) can be infered from a lower bound $\Omega(n^{1.51})$ for Universal Traversal Sequences (UTS) [9]. If this number of bits is logarithmic in n (otherwise, the exploration is infeasible [14]), then in this model it is possible to implement UTS (the agent only remembers the position in the sequence). The best known upper bound is $O(n^3)$ [1] and the best known constructive upper bound is $O(n^{4.03})$ [31]. Memory $\Theta(D \log \Delta)$ is sufficient and sometimes required to explore any graph with maximum degree Δ [24]. For directed graphs, memory $\Omega(n \log \Delta)$ at the agent is sometimes required to explore any graph with maximum outdegree Δ, while memory $O(n\Delta \log \Delta)$ is always sufficient [23].

Finite state automata. An agent equipped with only constant number of bits of persistent memory can be regarded as a finite state automaton (see e.g. [8, 24, 25]). Movements of such agent, typically modelled as a finite Moore or Mealy automaton, are completely determined by a state transition function $f(s, p, d_v) = (s', p')$, where s, s' are the agent's states and p, p' are the ports through which the agent enters and leaves the node v. A finite state automaton cannot explore an arbitrary graph in the setting, where the nodes have no unique labels [39]. Fraigniaud et al. [24] showed that for any $\Delta \geqslant 3$ and any finite state agent with k states, one can construct a planar graph with maximum degree Δ and at most $k + 1$ nodes, that cannot be explored by the agent. It is, however, possible in this model to explore (without stop) the trees, assuming that the finite state agent has access to the incoming port number (cf. [14]).

Two types of memory. Sudo et al. [40] consider exploration of general graphs with two types of memory, where both memories may have arbitrary initial states. They show that $O(\log n)$ bits at the agent and at each node allows exploration in time $O(m + nD)$. Cohen et al. [8] studied the problem of exploring an arbitrary graph by a finite state automaton, which is capable of assigning $O(1)$-bit labels to the nodes. They proposed an algorithm, that – assuming the agent knows the incoming port numbers – dynamically labels the nodes with three different labels and explores (with stop) an arbitrary graph in time $O(mD)$ (if the graph can be labelled offline, both the preprocessing stage and the exploration take $O(m)$ steps). They also show that with 1-bit labels and $O(\log \Delta)$ bits of agent memory it is possible to explore (with stop) all bounded-degree graphs of maximum degree Δ in time $O(\Delta^{10} m)$.

Tokens. Deterministic, directed graph exploration in polynomial time using tokens has been considered in [3], where it was shown that a single token is sufficient if the agent has an upper bound on the number of vertices n and $\Theta(\log \log n)$ tokens are sufficient and necessary otherwise. In [4] the authors proposed a probabilistic polynomial time algorithm that allows two cooperating agents without the knowledge of n to explore any strongly connected directed graph. They also proved that for a single agent with $O(1)$ tokens this is not possible in polynomial time in n with high probability. Using one pebble, the exploration with stop requires an agent with $\Omega(\log n)$ bits of memory [25]. The same space bound remains true for perpetual exploration [24]. Disser et al. [15] show that for a single agent with $O(1)$ memory, to explore any undirected anonymous graph (i.e. with unlabelled vertices and port-labelled edges) with n vertices, $\Theta(\log \log n)$ distinguishable pebbles are necessary and sufficient. They proposed an algorithm based on universal exploration sequences (UXS, cf. [30, 38]), which runs in polynomial time and the agent terminates after returning to the starting vertex with all pebbles. For the lower bound they show that an agent with $O((\log n)^{1-\varepsilon})$ bits of memory (for any constant $\varepsilon > 0$) needs $\Omega(\log \log n)$ pebbles to explore all undirected graphs.

Randomized and collaborative exploration. Although we focus on deterministic graph exploration by a single agent, there is a vast body of literature on randomized exploration techniques, see e.g. [1, 12, 18, 19]. A classical and well-studied processes are random walks, where the agent in each step moves to a neighbour chosen uniformly at random (or does not move with some constant probability). Exploration using this method takes expected time $\Omega(n \log n)$ [20] and $O(n^3)$ [21], where for each of these bounds there exists a graph class for which it is tight. Randomness alone cannot ensure linear time of tree exploration, since expected time $\Omega(n^2)$ is required even for paths [32]. However approaches using memory at the agent [28], local information on explored neighbours [5], or local information on degrees [36] have shown that there are many methods to speedup random walks. Finally, to achieve fast exploration, usage of multiple agents is possible, cf. [12, 13, 15, 16, 17, 22, 27, 29, 37].

4 Upper bounds

4.1 Exploration in CleanMem model

To simplify descriptions, let us denote the starting position of the agent as Root. In this section we show an algorithm exploring any tree in $O(n)$ steps using $O(1)$ bits of memory at the agent and $O(\log d_v)$ bits of memory at each node v of degree d_v in CleanMem model. The memory at each node is organized as follows. It contains two port pointers (of $\lceil \log d_v \rceil$ bits each):

- v.parent – at some point of the execution, contains the port number leading to Root,
- v.last – points to the last port taken by the agent during the exploration

and two flags (of 1 bit):

- v.root – indicates whether the node is Root of the exploration,
- v.visited – indicates whether the node has already been visited.

At each node, the initial state of last pointers is 1, initial state of parent is NULL and initial state of each flag is False. The agent's memory contains one variable State that can take one of five possible values: Initial, Roam, Down, Up, Terminated.

For simplicity of the pseudocode we use a flag *parentSet*, which controls if parent is correctly set. This flag does not need to be stored because it is set and accessed in the same round. It is possible to write a more complicated pseudocode without this variable.

In our pseudocode, we use a procedure MOVE(p), in which the agent traverses an edge labelled with port p from its current location. When the agent changes its state to Terminated, it does not make any further moves.

Algorithm's description. For the purpose of the analysis, assume that tree T is rooted at the initial position of the agent (the Root). The main challenge in designing an exploration algorithm in this model is that the agent located at some node v may not know which port leads to the parent of v. Indeed, if the agent knew which port leads to its parent, it could perform a DFS traversal. In Algorithm 1 the agent would traverse the edges corresponding to outgoing ports in order $1, 2, 3, \ldots, d_v$ (skipping the port leading to its parent) and take the edge to its parent after completing the exploration of the subtrees rooted at its current position. Note that it is possible to mark, which outgoing ports have already been traversed by the agent using only a single pointer at each node hence 0 bits at the agent and $\lceil \log d_v \rceil$ bits at each node allow for linear time exploration in this case.

Since in our model, the agent does not know, which port leads to the parent, we need a second pointer parent at each node. To set it correctly, we first observe that in the model CleanMem, using flag visited, it is possible to mark the nodes that have already been visited. Notice that when the agent traverses some edge outgoing from v in the tree for the first time and enters to a node that has already been visited, then this node is certainly the parent of v. We can utilize this observation to establish correctly v.parent pointer using state Down. When entering to a node in this state, we know that the previously taken edge (port number of this edge is stored in v.last) leads to the parent of v. Our algorithm also ensures, that after entering a subtree rooted at v, the agent leaves it once, and the next time it enters this subtree it correctly sets v.parent and in subsequent steps it visits all the nodes of the subtree. Finally, having correctly set pointers parent at all the nodes, allows the agent to efficiently return to the starting node after completing the exploration and flag root allows the agent to terminate the algorithm at the starting node.

▶ **Theorem 1.** *Algorithm 1 explores any tree and terminates at the starting node in $O(n)$ steps in CleanMem model.*

Proof. First note that the algorithm marks the starting node with flag root (line 2). The algorithm never returns to state Initial, hence only this node will be marked with the root flag. The only line, where the agent terminates the algorithm is line 14 hence the agent can only terminate in the starting node. We need to show that the agent will terminate in every tree and before the termination it will visit all the vertices and the time of the exploration will be $O(n)$. Let us denote the starting node as Root and for any node v different from Root, will call the single neighbour of v that is closer to Root as the parent of v.

Algorithm 1 Tree exploration in CleanMem model.

```
// Agent is at some node $v$ and the outgoing ports are $\{1, 2, \ldots, d_v\}$
1  if $State = Initial$ then
2      $v$.root ← True, State ← Roam, $v$.visited ← True;
3      MOVE($v$.last);
4  else
5      if $State = Down$ then
6          $v$.parent ← $v$.last, State ← Roam;      // mark edge as leading to parent
7          $v.parentSet$ ← True;
8      else
9          $v.parentSet$ ← False
10     if (($State = Roam$ and $d_v = 1$) or $State = Up$ or $v.parentSet = True$) and
        $v.root = False$ and $v.last = d_v$ and $v.parent \neq NULL$ then
11         State ← Up;                    // exploration of this subtree completed
12         MOVE($v$.parent) ;                              // return to the parent
13     else if $State = Up$ and $v.root = True$ and $v.last = d_v$ then
14         State ← Terminated;       // exploration of the whole tree completed
15     else
16         if $State = Roam$ and $v.visited = True$ and $v.parentSet = False$ then
17             State ← Down;
18         else if $State = Up$ then
19             State ← Roam, $v$.last ← $v$.last + 1;
20         else
21             if $v.visited = True$ then $v$.last ← $v$.last + 1;
22             else $v$.visited ← True;
23         MOVE($v$.last);
```

We will show the following claim using induction over the structure of the tree.

▷ Claim 2. Assume that the agent enters to some previously unvisited subtree rooted at v with n_v vertices for the first time in state Roam in step t_s. Then the agent:

C.1 returns to the parent of v for the first time in state Roam (denote by $t_r > t_s$ the step number of the first return from v to its parent),
C.2 goes back to v in the state Down at step $t_r + 1$,
C.3 returns to the parent of v for the second time in state Up (denote by $t_f > t_r + 1$ the step number of the second return from v to its parent),
C.4 visits all the vertices of this subtree and spends $O(n_v)$ steps within time interval $[t_s, t_f]$.

Proof. We will first show it for all the leaves. Then, assuming that the claim holds for all the subtrees rooted at the children of some node v, we will show it for v. To show this claim for any leaf l, consider the agent entering in state Roam to l. Upon the first visit to l, the agent sets the flag l.visited to True and leaves (without changing the state of the agent) with port 1. This proves C.1. In the next step, at the parent of l, a state changes to Down and the agent uses the same port as during the last time in parent of l. Therefore the agent moves back to l (C.2) and sets l.parent ← 1 (line 6). Then the agent changes its state to Up and moves to the parent (lines 11–12). This shows C.3. Node l was visited twice within the considered time steps, which shows C.4. This completes the proof of the claim for all the leaves.

Now, consider any internal node v of the tree (with $d_v > 1$) and assume that the claim holds for all its children. When the agent enters to v for the first time, it sets the flag v.visited to `True` and moves to its neighbour w (while being in state Roam) via port 1.

Case 1: w is the parent of v. This immediately shows C.1. If the parent of v is not Root, then it has the degree at least 2, hence the agent will evaluate the if-statement in line 10 to `False`. Thus the agent will execute line 16 and change its state to Down. The agent uses the same port as during the previous visit of w, therefore the agent goes back to v (C.2), hence it will correctly set the pointer v.parent (it will point to the parent of v), which shows C.3.

Case 2: w is a child of v. In this case we enter to a child of v. We have by the inductive assumption (C.1), that the agent will return from w to v in state Roam. Since the agent enters to v in state Roam, then the value of *parentSet* is `False` and the agent executes line 16, transitions to state Down and then line 23 it takes the same edge as during the last visit to v, hence it moves back to node w (C.2). By C.3 we get that the next time the agent will traverses the edge from w to v, it will be in state Up. Then the agent increments pointer v.last (line 19) and moves to the next neighbour of v (line 23).

Thus the agent either finds the parent or explores the whole subtree rooted at one of its children w in $O(n_w)$ steps (by the inductive assumption C.4). An analogous analysis holds for ports $2, 3, \ldots, d_v$. When the agent returns from the neighbour connected to v via the last port d_v, it is either in state Up or Down (the second case happens if the edge leading from v to its parent has port number d_v). In both cases it executes lines 11 and 12 and leaves to its parent (the pointer to parent is established since the agent had traversed each outgoing edge, hence it moved to its parent and correctly set the parent pointer).

By this way, the agent visits all the subtrees rooted at v's children $w_1, w_2, \ldots, w_{d_v-1}$ (or up to w_{d_v} if $v = \text{Root}$). Hence, the total number of steps for a node $v \neq \text{Root}$ is $O\left(\sum_{i=1}^{d_v-1} n_{w_i}\right) = O(n_v)$ and $v = \text{Root}$ it is $O\left(\sum_{i=1}^{d_v} n_{w_i}\right) = O(n)$. ◁

To complete the proof of Theorem 1 we need to analyse the actions of the agent at Root. Consider the actions of the agent at Root when the Root.last pointer takes values $i = 1, 2, \ldots, d_{\text{Root}}$. Let v_i be the neighbour of Root pointed by port number i at Root. Observe that the agent at Root behaves similarly as in all the other internal nodes (only exception is that the agent will never enter Root in state Down, because by Claim 2 the agent can enter to it only in states Roam or Up, since the Root has no parent). By Claim 2, the agent visits the whole subtrees rooted at these nodes in time proportional to the number of nodes in these subtrees. When the agent returns from node $v_{d_{\text{Root}}}$ in state Up then the algorithm terminates (line 14). The total runtime is proportional to the total number of nodes in the tree. ◀

4.2 Exploration in Token model

Algorithm's description. In Algorithm 2, the agent has a single token, which can be DROP*ped*, TAKE*n* and MOVE*d* (i.e., carried by the agent across an edge of the graph). Moreover, the agent is always in one state from set {Initial, Roam, RR, Down, Up, Terminated}. Our algorithm ensures, that in states Roam and RR the agent does not hold the token and in the remaining states, it does. In these four states the agent can eventually DROP it. Each node v has degree denoted by d_v, which is part of the input to the agent, when entering to a node. Moreover, the memory at each node v is organized into three variables: v.last and v.parent of size $\lceil \log d_v \rceil$ and a flag v.root $\in$ {`True`, `False`} to mark the Root. We assume that in all vertices, variables last, parent and root have initially any admissible value (if not, then the agent would easily notice it and change such a value). Moreover, when the agent is entering to a node, it can see whether the node contains the token or not.

We would like to perform a similar exploration as in CleanMem model – we will use pointers last and parent as in Algorithm 1. However, the difficulty in this model is that these pointers may have arbitrary initial values. Especially, if the initial value of parent is incorrect, Algorithm 1 may fall into an infinite loop. Hence in this section we propose a new Algorithm 2 that handles dirty memory using a single token. In this algorithm the agent maintains an invariant that the node with the token, and all the nodes on the path from the token's location to the Root are guaranteed to have correctly set parent pointers. To explore new nodes, the agent performs a Rotor-Router (shortly RR) traversal, starting from node v with the token to its neighbour w (by the invariant, the agent chooses w as one of its children, not its parent). During this traversal, the agent is resetting the parent pointers at each node (and cleaning the root flags). The agent is using last as the pointer for the purpose of RR algorithm. The agent does **not** have to reset last as the RR requires no special initialization. Moreover, by the properties of RR the agent does not traverse the same edge twice in the same direction before returning to the starting node. Since the agent starts in the node with the token, it can notice that it completed a traversal. During this traversal each edge is used at most twice (once in each direction). Moreover, each node visited during this traversal has cleaned memory (pointer parent points to NULL and root is set to False). After returning to the node with the token, pointer v.last points at w and w.last points at v. Hence it is possible to traverse this edge and correctly set pointer w.parent maintaining the invariant. After moving the token down, the agent starts the RR procedure again. Since the agent is not cleaning the pointer last, the RR will use different edges than during the previous traversal. Using this we show in the proof of Theorem 2 that our algorithm traverses every edge at most 6 times.

Preliminaries. Let us introduce some notation: a variable Token associated with a vertex currently occupied by the agent, which is 1, when the agent meets the token and 0 otherwise. If Token $= 1$, then the agent can TAKE the token and the agent with the token can DROP it. Let Path(v) denote a set of all vertices, which are on the shortest path from Root to v, excluding Root. Let T_v denote the subtree of T rooted at v, i.e., $(w \in T_v) \equiv (w = v \vee v \in \mathsf{Path}(w))$. Let $T_{v,p}$ denote a subtree of T_v, rooted at a node connected by edge labelled by outport p at vertex v, i.e. if p leads from v to w (where $v \in \mathsf{Path}(w)$), then $T_{v,p} = T_w$ (we do not define such a tree when p directs towards Root). We say that the action is performed away from Root (downwards), if it starts in v and ends in T_v. Otherwise the action is towards Root (upwards). Let Tok(t) and Ag(t) be the positions of the token and the agent, respectively, at the end of the moment t. Let Act(t) be the action performed during the t-th step. Let the variable v.visited$(t) \in \{0, 1\}$ indicate whether v was visited by Algorithm 2 until the moment t. Note that this variable is not stored at the nodes and this notation is only for the analysis.

Additionally, we consider two substates of RR, depending on the value of the Token variable. Substate RR_0 is state RR, if variable Token $= 0$ at the node to which the agent entered in the considered step. Similarly RR_1 indicates that the agent enters in state RR to a node with Token $= 1$. Note that this distinction is only for the purpose of the analysis, and it does not influence the definition of the algorithm. In our analysis, we will call Initial, Roam, RR_0, RR_1, Down, Up, Terminated *actions* as these states (and substates) correspond to different commands executed by the agent (see the pseudocode of Algorithm 2).

Algorithm 2 Tree exploration in Token model.

```
   // Agent is at some node $v$ and the
      outgoing ports are $\{1, 2, \ldots, d_v\}$
1  if State = Initial then
2      Clean(), $v$.last ← 1, DROP;
       // mark the root for termination
3      $v$.root ← True;
4      State ← Roam;
5  else if State = RR and Token = 1 then
       // substate RR$_1$
6      TAKE, State ← Down;
7  else if State = RR and Token = 0 then
       // substate RR$_0$
8      Clean();
9      Progress() ;        // increment last
10 else if State = Down then
11     DROP, $v$.parent ← $v$.last;
12     Progress();
13     State ← Roam;
14     IfUp() ;     // check if in a leaf
15 else if State = Up then
16     DROP, Progress();
17     State ← Roam;
18     if $v$.last = 1 and $v$.root = True then
19         State ← Terminated;
20     IfUp();
21 else    // State = Roam
22     Clean(), State ← RR;
23 if State ≠ Terminated then
24     MOVE($v$.last);
```

```
1  Procedure Clean()
2      $v$.root ← False, $v$.parent ← NULL;
```

```
1  Procedure Progress()
2      $v$.last ← ($v$.last mod $d_v$) + 1;
```

```
1  Procedure IfUp()
2      if $v$.parent = $v$.last then
3          TAKE, State ← Up;
```

We assume that Initial action is performed at time step 0. Let Root denote the initial position of the agent. Let v.last(t) and v.parent(t) denote, respectively, the values of v.last and v.parent pointers at the end of the moment t. If v is the starting point of the t-th step, then we say that v.last(t) is the outport related to t-th moment. Moreover, v.last$(\cdot)$ cannot be changed until the node v will be visited for the next time. Each MOVE(v.last) involves traversing an edge outgoing from the current position of the agent via the port indicated by the current value of variable last at the current position.

Properties of the algorithm. Let us define the following set of properties P(t) (the k-th property at moment t is denoted as $P.k(t)$; in this definition we denote Ag$(t) = w$) that describe the structure of the walk and the interactions with the memory at the nodes by an agent performing Algorithm 2.

P.1 During t-th step:
  a. If Down or Roam is performed, the move of the agent is away from Root (downwards).
  b. If Up or RR$_1$ is performed, the move of the agent is towards Root (upwards).
P.2 $w \in T_{\mathsf{Tok}(t)}$.
P.3 If w is visited for the first time at step t (Act$(t) \in \{$Initial, Roam, RR$_0\}$), then it is also cleaned, which means that w.parent is set to NULL and w.root is set to False.
P.4 For every $v \in$ Path(Tok(t)), v.parent$(t) \neq$ NULL.
P.5 If w.visited$(t-1) = 1$ and w.parent$(t-1) \neq$ NULL, then Act$(t) \notin \{$Roam, RR$_0\}$.
P.6 If w.visited$(t) = 1$ and w.parent$(t) \neq$ NULL, then w.parent(t) points towards Root.
P.7 If the state at the end of t-th moment is Up, then T_w is explored.
P.8 If $t > 0$, then for every $x \leq$ Root.last$(t-1)$, $T_{\mathsf{Root},x}$ is explored before t-th moment.

P.9 If $w.\mathsf{parent}(t) = \mathsf{NULL}$, then there do not exist $s_1 < s_2 < t$ such that $w.\mathsf{last}(s_1) = w.\mathsf{last}(t) \neq w.\mathsf{last}(s_2)$, where $w.\mathsf{parent}(s_1) = \mathsf{NULL}$ ($w.\mathsf{last}(\cdot)$ cannot be changed to the same value two times before $w.\mathsf{parent}(\cdot)$ was established).

P.10 There do not exist $s_1 < s_2 < t$ such that $w.\mathsf{last}(s_1) = w.\mathsf{last}(t) \neq w.\mathsf{last}(s_2)$ and $w.\mathsf{parent}(s_1) \neq \mathsf{NULL}$ ($w.\mathsf{last}(\cdot)$ cannot be changed to the same value two times after $w.\mathsf{parent}(\cdot)$ was established).

▶ **Lemma 3.** *At time* 0*, Initial phase of Algorithm 2 guarantees* $\mathsf{P}(0)$*.*

▶ **Lemma 4.** *If Algorithm 2 satisfies* $\mathsf{P}(s)$ *for all* $s \leq t$*, then it also fulfills* $\mathsf{P}(t+1)$*.*

All the omitted proofs from this section are deferred to the full version of the paper [6].

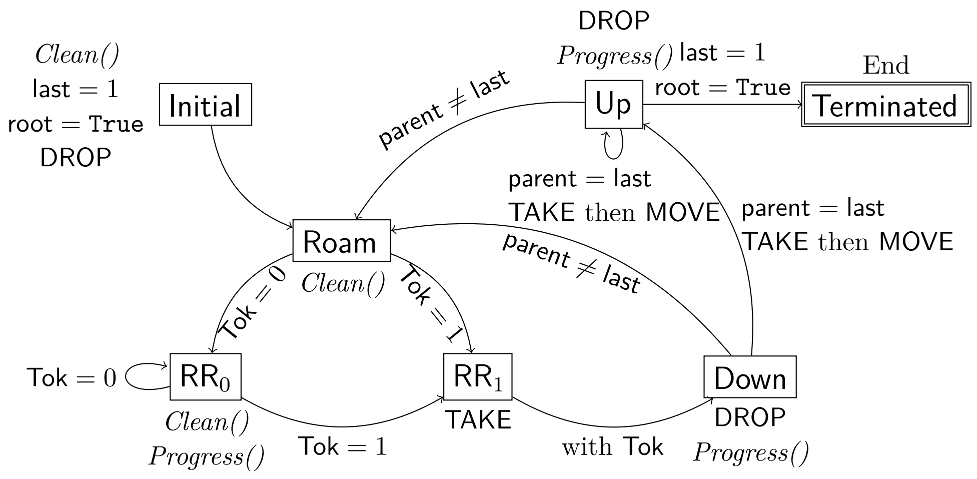

Figure 1 Illustration of actions and state transitions in Algorithm 2.

We will use properties $\mathsf{P}(t)$ to show correctness and time complexity of our Algorithm 2.

▶ **Theorem 5.** *Algorithm 2 explores any tree and terminates at the starting node in at most* $6(n-1)$ *steps in the Token model.*

Proof. Using Lemmas 3 and 4, we get that Algorithm 2 satisfies $\mathsf{P}(t)$ for each step t. First of all, from P.3, all visited vertices are cleaned upon the first visits.

Consider some vertex v and its arbitrary outport p. If $v \neq \mathsf{Root}$, then from P.9 and P.10, p may be used two times instantly after the incrementation of $v.\mathsf{last}(\cdot)$ (once when $v.\mathsf{parent}(\cdot) = \mathsf{NULL}$ and once when $v.\mathsf{parent}(\cdot) \neq \mathsf{NULL}$). Realize that, when $v = \mathsf{Root}$, then from P.2 and the fact, that any vertex with the token cannot be cleaned by RR_0 and Roam moves, we claim that $\mathsf{Root.parent}(\cdot)$ will always be NULL. By P.9, p may be used once instantly after the incrementation of $\mathsf{Root.last}(\cdot)$. Moreover, regardless of the choice of p, it may be used also after Roam or RR_1 action without a change of $v.\mathsf{last}(\cdot)$.

Case 1. Assume that $\mathsf{Act}(t) = \mathsf{RR}_1$ is taken via port p. From P.1(t) and P.2(t), p directs upwards towards the token. Then $\mathsf{Act}(t+1) = \mathsf{Down}$ is performed downwards (from P.1$(t+1)$) via the same edge as p (but with the opposite direction) and changes the position of the token to $\mathsf{Tok}(t+1) = v$ and establishes $v.\mathsf{parent}(t+1) \leftarrow p$. By P.1, p cannot be used during Roam action. Realize that if $p = v.\mathsf{last}(s-1)$ for some moment $s > t$ and $\mathsf{Act}(s) = \mathsf{RR}_1$, then $\mathsf{Tok}(s-1)$ is one step towards Root from v (from P.1(s)). Since the token can be moved only during Up and Down actions, then there exists a moment s' such that $t < s' < s$ and $\mathsf{Act}(s') = \mathsf{Up}$ changes the position of the token from v to $\mathsf{Tok}(s)$. However then $\mathsf{Tok}(s).\mathsf{last}(s) \neq \mathsf{Tok}(s).\mathsf{last}(s-1)$ ($\mathsf{Tok}(s)$ is not a leaf) and by P.10, P.6 and P.4, it cannot be changed back since $\mathsf{parent}(\mathsf{Tok}(s)) \neq \mathsf{NULL}$.

Case 2. Assume that $\mathsf{Act}(t) = \mathsf{Roam}$ is taken from node $v = \mathsf{Tok}(t-1)$ via port p (downwards, by P.1(t)). The first action after time t, which leads to the token, is always RR_1. However, as showed before, after RR_1 action there is an instant Down action, which moves the token via p to some vertex w and sets $w.\mathsf{parent} \neq \mathsf{NULL}$. Hence by P.5, Roam action cannot be performed via p once again. By P.1, p cannot be used during RR_1 action.

Our considerations show that each outport can be used at most 3 times. Since each tree of size n has $2(n-1)$ outports, Algorithm 2 terminates after at most $6(n-1)$ moves.

It remains to show that all vertices are then explored. Realize that Algorithm 2 terminates in $(t+1)$-st moment only when the agent is in $\mathsf{Tok}(t)$, $\mathsf{Tok}(t).\mathsf{last}(t) = 1$, $\mathsf{Tok}(t).\mathsf{root} = \texttt{True}$ and the state is Up at the end of t-th moment. From P.3, we know that the first visit in some v sets root flag to `True` in Root and to `False` in all other nodes, and this flag does not change (Roam and RR_0 do not clean the vertices with the token, so by P.2, Root cannot be cleaned by these moves). Hence the termination entails $\mathsf{Tok}(t) = \mathsf{Root}$. Since $\mathsf{Root}.\mathsf{parent}(t) = \mathsf{NULL}$, then from P.9 we know that each port p in Root will be set by $\mathsf{Root}.\mathsf{last}(\cdot)$ only once (since the first moment) and P.8 means that each subtree $T_{\mathsf{Root},p}$ for $p \in \{1, \dots, d_{\mathsf{Root}}\}$ were explored before returning to Root, hence the exploration of T_{Root} was completed. ◀

5 Lower bound

In this section we analyse the exploration of paths with one bit of memory at each node and one bit at the agent. We show that in this setting, in the DirtyMem model, the exploration of the path sometimes requires $\Omega(n^2)$ steps.

Notation. In the following lower bound on the path, we will focus on the actions of the algorithm performed in vertices with degree 2. In such vertices, there are four possible inputs to the algorithm $S = \{(0,0),(0,1),(1,0),(1,1)\}$, where in a pair (a,v), a denotes a bit on the agent and v is a bit saved on the vertex. Let us denote the sets of agent and vertex states by As and Vs, respectively. We also use elements from Ps $= \{0,1\}$ to denote ports in vertices of degree 2. Moreover, a shorthand notation $\overline{b}$ indicates the inverted value of a bit. For example, if $a = 0$, then $\overline{a} = 1$ and vice versa. The goal of this section is to prove:

▶ **Theorem 6.** *Every deterministic algorithm that can explore any path in the DirtyMem model with one bit at the agent and one bit at each node requires time $\Omega(n^2)$ to explore some worst-case path with n nodes.*

Assume, for a contradiction, that there exists algorithm $\mathcal{A}$, which explores every path in $o(n^2)$ steps. Without loss of generality, we may assume that the adversary always sets the agent in the middle point of the path. For algorithm $\mathcal{A}$, for every $s \in S$, by $A(s)$, $V(s)$, $P(s)$ we denote, respectively, the returned agent state, vertex state and the chosen outport in each vertex with degree 2. Moreover, let $R_3(s) := (A(s), V(s), P(s))$ and $R_2(s) := (A(s), V(s))$. We will show that $\mathcal{A}$ either falls into an infinite loop or performs no faster than Rotor-Router.

The proof of Theorem 6 is divided into several parts. In the first one, we provide several properties of potential algorithms $\mathcal{A}$ that can eventually explore the path in $o(n^2)$ time. These properties are used in the second part, where we prove that the agent does not change its internal state every time, however it has to change the state of the vertex in each step. Further, we check two algorithms (called X and Y), which turn out to explore the path even slower than the Rotor-Router algorithm. The analysis of those two algorithms is quite challenging. Next we show that if three states of $\mathcal{A}$ return the same outport, then either it falls into an infinite loop or explores the path in $\Omega(n^2)$ steps. In the latter part we consider the rest of amenable algorithms, which can either be reduced to previous counterexamples or fall into an infinite loop. A big part of the proof is technical and due to space limitations it is moved to the full version of the paper [6]. Nevertheless, in here we present two parts of the proof to let the reader feel the flavour of high-level ideas and utilized techniques.

▶ **Fact 7.** $(\forall\, a \in \mathrm{As})(\exists\, v \in \mathrm{Vs})\ A(a, v) = \overline{a}$.

Proof. Assume that $(\exists\, a \in \mathrm{As})(\forall\, v \in \mathrm{Vs})\ A(a, v) = a$. If a is the initial state of the agent, then the agent will never change its state before reaching an endpoint of the path. Hence, the time to reach the endpoint cannot be faster than Rotor-Router algorithm, hence the adversary can initialize ports in such a way that the endpoint will be reached after $\Omega(n^2)$ steps. If $\overline{a}$ is the starting state, then either the agent state never changes which reduces to the previous case or $A(\overline{a}, w) = a$ for some $w \in \mathrm{Vs}$. Then, the adversary sets w as the initial state of the starting vertex and after the fist step the state of the agent changes to a and by the same argument as in the first case, the algorithm requires $\Omega(n^2)$ steps. ◀

▶ **Fact 8.** $(\forall\, a \in \mathrm{As})(\forall\, v \in \mathrm{Vs})\ R_2(a, v) \neq (a, v)$.

Proof. We will show that otherwise algorithm $\mathcal{A}$ falls into an infinite loop for some initial state of the path. Assume that $R_3(a, v) = (a, v, p)$ for some $a \in \mathrm{As}$, $v \in \mathrm{Vs}$, $p \in \mathrm{Ps}$. Then if a is the starting agent state, the agent falls into an infinite loop on the left gadget from Figure 2 (double circle denotes the starting position of the agent and a above the node indicates its initial state). If $\overline{a}$ is the starting state, then by Fact 7, $(\exists\, w \in \mathrm{Vs})\ A(\overline{a}, w) = a$. Let $q = P(\overline{a}, w)$ and note that the agent falls into an infinite loop in the right gadget at Figure 2. ◀

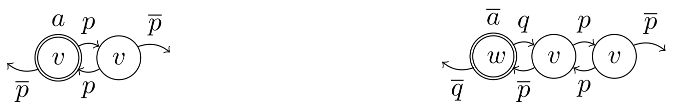

Figure 2 Two looping gadgets for an algorithm with $R_2(a, v) = (a, v)$.

Independently of the above facts, let us consider the foregoing **Algorithm Q** and more complicated gadgets, which force the agent to fall into an infinite loop (see Figure 3).

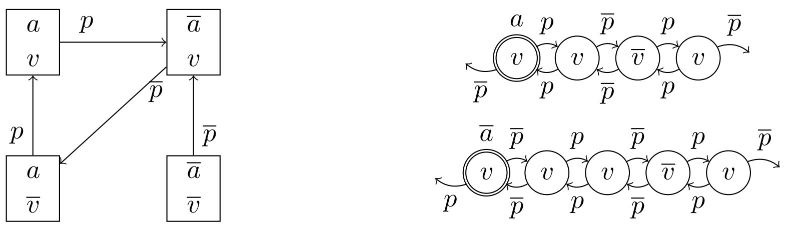

Figure 3 Definition of Algorithm Q (left) and gadgets on which it falls into an infinite loop.

6 Conclusions and open problems

One conclusion from our paper is that certain assumptions of the model of mobile agents can be exchanged. We showed, that in the context of linear time tree exploration, the assumption of clean memory at the nodes can be exchanged for a single token or the knowledge of the incoming port. The paper leaves a number of promising open directions. We showed that token and clean memory allow for linear time exploration of trees, however, we have not ruled out the possibility that linear time exploration of trees is feasible without both these assumptions. Our lower bound suggests that memory $\omega(1)$ at the agent is probably necessary in DirtyMem model. Another open direction would be to consider different graph classes, or perhaps directed graphs. Finally, a very interesting future direction is to study dual-memory exploration with team of multiple mobile agents. Such approach could lead to even smaller exploration time, however, dividing the work between the agents in such models is very challenging since the graph is initially unknown.

References

1 Romas Aleliunas, Richard M. Karp, Richard J. Lipton, László Lovász, and Charles Rackoff. Random Walks, Universal Traversal Sequences, and the Complexity of Maze Problems. In *20th Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 29-31 October 1979*, pages 218–223. IEEE Computer Society, 1979. doi:10.1109/SFCS.1979.34.

2 Evangelos Bampas, Leszek Gąsieniec, Nicolas Hanusse, David Ilcinkas, Ralf Klasing, Adrian Kosowski, and Tomasz Radzik. Robustness of the Rotor-Router Mechanism. *Algorithmica*, 78(3):869–895, 2017. doi:10.1007/s00453-016-0179-y.

3 Michael A. Bender, Antonio Fernández, Dana Ron, Amit Sahai, and Salil P. Vadhan. The Power of a Pebble: Exploring and Mapping Directed Graphs. *Inf. Comput.*, 176(1):1–21, 2002. doi:10.1006/inco.2001.3081.

4 Michael A. Bender and Donna K. Slonim. The Power of Team Exploration: Two Robots Can Learn Unlabeled Directed Graphs. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 75–85, 1994. doi:10.1109/SFCS.1994.365703.

5 Petra Berenbrink, Colin Cooper, and Tom Friedetzky. Random Walks Which Prefer Unvisited Edges: Exploring High Girth Even Degree Expanders in Linear Time. *Random Struct. Algorithms*, 46(1):36–54, 2015. doi:10.1002/rsa.20504.

6 Dominik Bojko, Karol Gotfryd, Dariusz R. Kowalski, and Dominik Pajak. Tree exploration in dual-memory model, 2022. (Full version). arXiv:2112.13449.

7 Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1):107–117, 1998. Proceedings of the Seventh International World Wide Web Conference. doi:10.1016/S0169-7552(98)00110-X.

8 Reuven Cohen, Pierre Fraigniaud, David Ilcinkas, Amos Korman, and David Peleg. Label-Guided Graph Exploration by a Finite Automaton. *ACM Trans. Algorithms*, 4(4):42:1–42:18, August 2008. doi:10.1145/1383369.1383373.

9 H. K. Dai and Kevin E. Flannery. Improved Length Lower Bounds for Reflecting Sequences. In Jin-Yi Cai and Chak Kuen Wong, editors, *Computing and Combinatorics*, pages 56–67. Springer Berlin Heidelberg, 1996. doi:10.1007/3-540-61332-3_139.

10 Shantanu Das. Graph Explorations with Mobile Agents. In Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro, editors, *Distributed Computing by Mobile Entities: Current Research in Moving and Computing*, Lecture Notes in Computer Science, pages 403–422. Springer International Publishing, Cham, 2019. doi:10.1007/978-3-030-11072-7_16.

11 Shantanu Das and Nicola Santoro. Moving and Computing Models: Agents. In Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro, editors, *Distributed Computing by Mobile Entities: Current Research in Moving and Computing*, Lecture Notes in Computer Science, pages 15–34. Springer International Publishing, Cham, 2019. doi:10.1007/978-3-030-11072-7_2.

12 Dariusz Dereniowski, Yann Disser, Adrian Kosowski, Dominik Pająk, and Przemysław Uznański. Fast collaborative graph exploration. *Inf. Comput.*, 243:37–49, 2015. doi:10.1016/j.ic.2014.12.005.

13 Dariusz Dereniowski, Adrian Kosowski, Dominik Pająk, and Przemyslaw Uznanski. Bounds on the cover time of parallel rotor walks. *J. Comput. Syst. Sci.*, 82(5):802–816, 2016. doi:10.1016/j.jcss.2016.01.004.

14 Krzysztof Diks, Pierre Fraigniaud, Evangelos Kranakis, and Andrzej Pelc. Tree exploration with little memory. *J. Algorithms*, 51(1):38–63, 2004. doi:10.1016/j.jalgor.2003.10.002.

15 Yann Disser, Jan Hackfeld, and Max Klimm. Tight Bounds for Undirected Graph Exploration with Pebbles and Multiple Agents. *J. ACM*, 66(6), October 2019. doi:10.1145/3356883.

16 Yann Disser, Frank Mousset, Andreas Noever, Nemanja Škorić, and Angelika Steger. A general lower bound for collaborative tree exploration. *Theoretical Computer Science*, 811:70–78, 2020. doi:10.1016/j.tcs.2018.03.006.

17 Mirosław Dynia, Jakub Łopuszański, and Christian Schindelhauer. Why Robots Need Maps. In G. Prencipe and S. Zaks, editors, *Structural Information and Communication Complexity*, volume 4474 of *Lecture Notes in Computer Science*, pages 41–50. Springer Berlin Heidelberg, 2007. doi:10.1007/978-3-540-72951-8_5.

18 Klim Efremenko and Omer Reingold. How Well Do Random Walks Parallelize? In Irit Dinur, Klaus Jansen, Joseph Naor, and José Rolim, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, volume 5687 of *Lecture Notes in Computer Science*, pages 476–489. Springer Berlin Heidelberg, 2009. doi:10.1007/978-3-642-03685-9_36.

19 Robert Elsässer and Thomas Sauerwald. Tight bounds for the cover time of multiple random walks. *Theor. Comput. Sci.*, 412(24):2623–2641, 2011. doi:10.1016/j.tcs.2010.08.010.

20 Uriel Feige. A Tight Lower Bound on the Cover Time for Random Walks on Graphs. *Random Struct. Algorithms*, 6(4):433–438, 1995. doi:10.1002/rsa.3240060406.

21 Uriel Feige. A Tight Upper Bound on the Cover Time for Random Walks on Graphs. *Random Struct. Algorithms*, 6(1):51–54, 1995. doi:10.1002/rsa.3240060106.

22 Pierre Fraigniaud, Leszek Gąsieniec, Dariusz R. Kowalski, and Andrzej Pelc. Collective Tree Exploration. *Networks*, 48(3):166–177, 2006. doi:10.1002/net.20127.

23 Pierre Fraigniaud and David Ilcinkas. Digraphs Exploration with Little Memory. In V. Diekert and M. Habib, editors, *STACS 2004*, volume 2996 of *Lecture Notes in Computer Science*, pages 246–257. Springer Berlin Heidelberg, 2004. doi:10.1007/978-3-540-24749-4_22.

24 Pierre Fraigniaud, David Ilcinkas, Guy Peer, Andrzej Pelc, and David Peleg. Graph exploration by a finite automaton. *Theor. Comput. Sci.*, 345(2-3):331–344, 2005. doi:10.1016/j.tcs.2005.07.014.

25 Pierre Fraigniaud, David Ilcinkas, Sergio Rajsbaum, and Sébastien Tixeuil. Space Lower Bounds for Graph Exploration via Reduced Automata. In A. Pelc and M. Raynal, editors, *Structural Information and Communication Complexity*, volume 3499 of *Lecture Notes in Computer Science*, pages 140–154. Springer Berlin Heidelberg, 2005. doi:10.1007/11429647_13.

26 Leszek Gąsieniec and Tomasz Radzik. Memory Efficient Anonymous Graph Exploration. In H. Broersma, T. Erlebach, T. Friedetzky, and D. Paulusma, editors, *Graph-Theoretic Concepts in Computer Science*, volume 5344 of *Lecture Notes in Computer Science*, pages 14–29. Springer Berlin Heidelberg, 2008. doi:10.1007/978-3-540-92248-3_2.

27 Ralf Klasing, Adrian Kosowski, Dominik Pająk, and Thomas Sauerwald. The multi-agent rotor-router on the ring: a deterministic alternative to parallel random walks. *Distributed Comput.*, 30(2):127–148, 2017. doi:10.1007/s00446-016-0282-y.

28 Adrian Kosowski. Faster Walks in Graphs: A $\widetilde{O}(n^2)$ Time-Space Trade-off for Undirected s-t Connectivity. In *Proceedings of the 2013 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1873–1883. SIAM, 2013. doi:10.1137/1.9781611973105.133.

29 Adrian Kosowski and Dominik Pająk. Does adding more agents make a difference? A case study of cover time for the rotor-router. *J. Comput. Syst. Sci.*, 106:80–93, 2019. `doi:10.1016/j.jcss.2019.07.001`.

30 Michal Koucký. Universal traversal sequences with backtracking. *Journal of Computer and System Sciences*, 65(4):717–726, 2002. `doi:10.1016/S0022-0000(02)00023-5`.

31 Michal Koucký. Log-space constructible universal traversal sequences for cycles of length $O(n^{4.03})$. *Theor. Comput. Sci.*, 296(1):117–144, 2003. `doi:10.1016/S0304-3975(02)00436-X`.

32 László Lovász. Random Walks on Graphs: A Survey. Combinatorics, Paul Erdos is Eighty. *Bolyai Soc. Math. Stud.*, 2:1–46, 1993.

33 Nicole Megow, Kurt Mehlhorn, and Pascal Schweitzer. Online graph exploration: New results on old and new algorithms. *Theoretical Computer Science*, 463:62–72, 2012. Special Issue on Theory and Applications of Graph Searching Problems. `doi:10.1016/j.tcs.2012.06.034`.

34 Artur Menc, Dominik Pająk, and Przemysław Uznański. Time and space optimality of rotor-router graph exploration. *Inf. Process. Lett.*, 127:17–20, 2017. `doi:10.1016/j.ipl.2017.06.010`.

35 Nathan Michael et al. Collaborative Mapping of an Earthquake-Damaged Building via Ground and Aerial Robots. *Journal of Field Robotics*, 29(5):832–841, 2012. `doi:10.1002/rob.21436`.

36 Yoshiaki Nonaka, Hirotaka Ono, Kunihiko Sadakane, and Masafumi Yamashita. The hitting and cover times of Metropolis walks. *Theor. Comput. Sci.*, 411(16-18):1889–1894, 2010. `doi:10.1016/j.tcs.2010.01.032`.

37 Christian Ortolf and Christian Schindelhauer. A Recursive Approach to Multi-robot Exploration of Trees. In Magnús M. Halldórsson, editor, *Structural Information and Communication Complexity*, volume 8576 of *Lecture Notes in Computer Science*, pages 343–354. Springer International Publishing, 2014. `doi:10.1007/978-3-319-09620-9_26`.

38 Omer Reingold. Undirected Connectivity in Log-Space. *J. ACM*, 55(4), September 2008. `doi:10.1145/1391289.1391291`.

39 H. A. Rollik. Automaten in planaren Graphen. *Acta Inf.*, 13(3):287–298, March 1980. `doi:10.1007/BF00288647`.

40 Yuichi Sudo, Fukuhito Ooshita, and Sayaka Kamei. Self-stabilizing Graph Exploration by a Single Agent. *CoRR*, abs/2010.08929, 2020. `arXiv:2010.08929`.

41 Vladimir Yanovski, Israel A. Wagner, and Alfred M. Bruckstein. A Distributed Ant Algorithm for Efficiently Patrolling a Network. *Algorithmica*, 37(3):165–186, 2003. `doi:10.1007/s00453-003-1030-9`.