# Digraphs Exploration with Little Memory

Pierre Fraigniaud and David Ilcinkas

CNRS, LRI, Université Paris-Sud, 91405 Orsay, France.
{pierre,ilcinkas}@lri.fr

**Abstract.** Under the robot model, we show that a robot needs $\Omega(n \log d)$ bits of memory to perform exploration of digraphs with $n$ nodes and maximum out-degree $d$. We then describe an algorithm that allows exploration of any $n$-node digraph with maximum out-degree $d$ to be accomplished by a robot with a memory of size $O(nd \log n)$ bits. Under the agent model, we show that digraph exploration cannot be achieved by an agent with no memory. We then describe an exploration algorithm for an agent with a constant-size memory, using a whiteboard of size $O(\log d)$ bits at every node of out-degree $d$.

## 1 Introduction

A mobile entity (e.g., a software agent or a robot) has to *explore* a graph by visiting all its nodes and traversing all edges, without any *a priori* knowledge of the topology of the graph nor of its size. Once exploration is completed, the mobile entity has to stop. We also consider the more demanding task of *exploration with return* in which the entity has to return to its original position, and the auxiliary easier task of *perpetual exploration* in which the entity has to traverse all edges of the graph but is not required to stop. The task of visiting all nodes of a network is fundamental in searching for data stored at unknown nodes of a network, and traversing all edges is often required in network maintenance and when looking for defective components. Perpetual exploration may be of independent interest, e.g., if regular control of a network for the presence of faults is required, and all edges must be periodically traversed over long periods of time.

If nodes and edges have unique labels, exploration can be easily done (e.g., by depth-first search). However, in some navigation problems in unknown environments such unique labeling may not be available, or limited sensory capabilities of the mobile entity may prevent it from perceiving such labels. Hence it is important to be able to program the entity to explore *anonymous* graphs, i.e., graphs without unique labeling of nodes or edges. Arbitrary graphs cannot be explored under such weak assumptions, as witnessed by the case of a cycle: without any labels of nodes and without the possibility of putting marks on them, it is clearly impossible to explore a cycle of unknown size and stop. Hence, we assume, as in [5, 6, 11], some ability of marking nodes. More precisely we consider two different models. In the *robot model*, the mobile entity is given the ability of dropping and removing indistinguishable *pebbles* at nodes. This model aims to capture the behavior of a robot in a labyrinth. In the *agent model*, the mobile entity is given the ability to read and write messages at memory locations available at each node, called *whiteboards*. This model aims to capture the behavior of a software agent in a computer network. Observe that the robot model is weaker than the agent model since a robot that is given $k$ pebbles acts as a software agent in a network with whiteboards of size one bit, in which at most $k$ whiteboards can simultaneously contain a 1.

Clearly the robot has to be able to *locally* distinguish ports at a node: otherwise it is impossible to explore even the star with 3 leaves (after visiting the second leaf the robot cannot distinguish the port leading to the first visited leaf from that leading to the unvisited one). Hence we make a natural assumption that all ports at a node are locally labeled $1, \ldots, d$, where $d$ is the degree of the node. No coherence between those local labelings is assumed.

In many applications, robots and mobile agents are meant to be simple, often small, and inexpensive devices which limits the amount of memory with which they can be equipped. As opposed to numerous papers that imposed no restrictions on the memory of the robot and sought exploration algorithms minimizing time, i.e., the number of edge traversals, we investigate the minimum size of the memory of the robot that allows exploration of graphs of given (unknown) size, regardless of the time of exploration. That is, we want to find an algorithm for a mobile entity performing exploration using as little memory as possible, i.e., we want to minimize the memory of the robot in the robot model, and we want to minimize both the amount of information transported by the agent and the size of the whiteboards in the agent model. In the latter case, our specific goal is to design an exploration algorithm for an agent with constant memory size, using small whiteboards.

### 1.1 Our results

Under the robot model, we first prove a lower bound of $\Omega(n \log d)$ bits of memory for perpetual exploration of $n$-node digraphs with maximum out-degree $d$. This lower bound holds even if the robot is given a linear amount of pebbles. We then present two algorithms for exploration with stop in digraphs. One requires $O(nd \log n)$ bits of memory, and uses one pebble. This algorithm is only $O(\log n)$ away from the optimal in constant-degree digraphs. Its time performance is however exponential (again, time is measured by the number of edge traversals). Hence, we also describe another algorithm, which performs exploration with stop in polynomial time, but requires $O(n^2 d \log n)$ bits of memory, and uses $O(\log \log n)$ pebbles. This latter algorithm is a variant of the algorithm in [5], designed for the purpose of compressing the robot memory. Note that it has been proved [5] that $\Omega(\log \log n)$ pebbles are required to explore in polynomial time, thus our algorithm is optimal with regard to the number of pebbles.

Under the agent model, we first prove that exploration with stop cannot be achieved by an oblivious agent, i.e., an agent carrying no information when moving from one node to another. However, we describe an algorithm for an agent with constant size memory. It performs exploration with return in all digraphs, using a whiteboard of size $O(\log d)$ at every node of out-degree $d$. Note that $\Omega(\log d)$ bits is a lower bound for the size of the whiteboards when using an agent with constant-size memory. Indeed, an agent with a memory of size $k = O(1)$ and a whiteboard of size $k' = o(\log d)$ generate at most $2^{k+k'} < d$ states, and hence not all out-going edges can be distinguished. Our algorithm is also optimal according to the following criteria. We mentioned before the lower bound of $\Omega(n \log d)$ bits of memory for exploration in digraphs under the robot model. Therefore our algorithm under the agent model demonstrates that the memory of the robot can be optimally distributed among the $n$ nodes of the digraph. This is in contrast with other contexts (e.g., compact routing) in which there is a penalty for the distribution of a centralized data structure.

### 1.2 Related work

Exploration and navigation problems for robots in an unknown environment have been extensively studied in the literature (cf. [14]). There are two groups of models for these problems. In one of them a particular geometric setting is assumed. Another approach is to model the environment as a graph, assuming that the robot may only move along its edges. The graph setting can be further specified in two different ways. In [1, 5, 6, 9] the robot explores strongly connected directed graphs and it can move only in the direction from head to tail of an edge, not vice-versa. In [2, 7, 10–12, 16] the explored graph is undirected and the robot can traverse edges in both directions. (See also [13] an the references therein where parallel search is investigated.) In the graph setting it is often required that apart from completing exploration the robot has to draw a map of the graph, i.e., output an isomorphic copy of it.

Graph exploration scenarios considered in the literature differ in an important way: it is either assumed that nodes of the graph have unique labels which the robot can recognize, or it is assumed that nodes are anonymous. It is impossible to explore arbitrary anonymous graphs if no marking of nodes is allowed. Hence the scenario adopted in [5, 6] was to allow *pebbles* which the robot can drop on nodes to recognize already visited ones, and then remove them and drop in other places. The authors concentrated attention on the minimum number of pebbles allowing efficient exploration and mapping of arbitrary directed $n$-node graphs. (In the case of undirected graphs, one pebble suffices for efficient exploration [11].) In [6] the authors compared exploration power of one robot with pebbles to that of two cooperating robots. In [5] it was shown that, to perform exploration in polynomial time, one pebble is enough if the robot knows an upper bound on the size of the graph. However, without the knowledge of any bound on the size of the graph, $\Theta(\log \log n)$ pebbles are necessary and sufficient for exploration in polynomial time.

The efficiency measure adopted in most papers dealing with graph exploration is the completion time of this task, measured by the number of edge traversals. On the other hand, there are no restrictions imposed on the memory of the robot. Minimizing the memory of the robot for the exploration of anonymous non-directed graphs has been addressed in, e.g., [8, 10, 16, 17]. Most of previous works deal with perpetual exploration. For instance, it is shown in [17] that, with no pebble, no finite set of finite automata can perform perpetual exploration of all cubic planar graphs. Using a pebble, exploration with stop of undirected graphs is much facilitated by the ability of backtracking. In particular, it is easy to design an exploration algorithm for a robot with $O(D \log d)$ bits of memory, where $D$ denotes the diameter of the graph. Also, a simple variant of the algorithm in [11] yields a bound $O(n \log d)$ bits. Better bounds are known for specific families of graphs. For instance, it is shown in [10] that exploration with stop in $n$-node trees requires a robot with memory size $\Omega(\log\log\log n)$, and that exploration with return in $n$-node trees can be achieved by a robot with $O(\log^2 n)$ bits of memory. Our paper focuses on directed graphs.

It is worth mentioning that our work has connections with derandomized random walks (cf. [10] and the references therein). There, the objective is to produce an explicit universal traversal sequence (UTS), i.e., a sequence of port labels, such that the path guided by this sequence visits all edges of any graph. However, without the *a priori* knowledge of $n$, non of these UTS allows the robot to stop. Moreover, even if bounds on the length of these sequences have been derived, they provide little knowledge on the minimum number of states for graph exploration by a robot. For instance, sequences of length $\Omega(n \log n)$ are required to traverse all degree 2 graphs with $n$ nodes [3], although a 2-state robot can explore all degree-2 graphs.

## 2 Terminology and models

An anonymous graph (resp., digraph) with locally labeled ports is a connected graph (resp., strongly connected digraph) whose nodes are unlabeled, and edges incident to a node $v$ have distinct labels 1,...,$d$, where $d$ is the degree of $v$. Thus every undirected edge $\{u, v\}$ has two labels which are called its port numbers at $u$ and at $v$. Port numbering is local: there is no relation between labels at $u$ and at $v$. In digraphs, edges out-going from a node $v$ have distinct labels 1,...,$d$, where $d$ is the out-degree of $v$. Edges incoming to a node $v$ are not labeled at $v$.

We are given a mobile entity traveling in an anonymous (di)graph with locally labeled ports. The graph and its size are *a priori* unknown to the entity. We consider the two following models.

*Robot model.* The mobile entity is called a *robot*. A robot with $k$-bit memory is a finite automaton of $K = 2^k$ states among which a specified state $S_0$ is called *initial* and some specified states are called *final*. The robot is originally given a source of indistinguishable pebbles. If the robot is in a node $v$ in a non-final state $S$, this state determines a local port number $p$, and the decision of dropping a pebble at $v$, removing a pebble from $v$ (if such a pebble is currently present at $v$), or doing nothing. Then the robot leaves the node by port $p$. Upon traversing the corresponding edge, the behavior of the robot differs depending whether the graph is directed or not.

In graphs, the robot reads the port number $i$ at the node it enters, and the degree $d$ of this node. It also detect the presence or not of a pebble at this node,

$b = 0$ if no pebble, and $b = 1$ otherwise. The triple $(i, d, b)$ is an input symbol that causes the transition from state $S$ to $S'$.

In digraphs, the robot reads the out-degree $d$ of the node it enters, and check the presence or not of a pebble at this node. The pair $(d, b)$ is an input symbol that causes the transition from state $S$ to $S'$.

In both cases, the robot continues moving in this way until it enters a final state for the first time. Then it stops.

*Agent model.* The mobile entity is called *agent*. An agent with $k$-bit memory is a pair $(\mathcal{P}, \mathcal{M})$ where $\mathcal{P}$ is a constant size program, and $\mathcal{M}$ is a memory of size $k$ bits. In the agent model, every node is given computing facilities, including CPU and $q$ bits of local memory. The local memory is called *whiteboard*. Initially, all whiteboards are empty, and the agent memory contains an *initial* $k$-bit binary string $s_0$. Every pair agent-node forms a system that acts as a finite automaton of $2^{k+q}$ states. A state of the system is a pair $S = (s, \omega)$ where $s$ is the content of the agent memory, and $\omega$ is the content of the whiteboard. This includes some specified states called *final*. When the system is in a non-final state $S$, the agent is operated as follows. The state $S$ determines a local port number $p$, a $k$-bit binary string $s$, and a $q$-bit binary string $\omega$. Then $\omega$ is written on the whiteboard, $s$ is stored in the agent memory, and the agent is sent through port $p$. Upon reception of the agent by a node, the operation performed by that node differs depending whether the graph is directed or not.

In graphs, let $i$ be the port number through which the agent enters the current node. Let $d$ be the degree of that node, and let $s$ and $\omega$ be the current contents of the agent memory and the node whiteboard. The pair $(i, d)$ is an input symbol that causes transition of the system from state $S = (s, \omega)$ to $S' = (s', \omega')$ by application of program $\mathcal{P}$.

In digraphs, the out-degree $d$ is an input symbol that causes the transition from state $S = (s, \omega)$ to $S' = (s', \omega')$ by application of program $\mathcal{P}$. (There is no access to the input port number.)

In both cases, the agent continues moving in this way until it enters a final state for the first time. Then it stops.

*Remark.* Most of our exploration algorithms under the robot model actually perform in the weakest version of the model, i.e., when the robot is given a unique pebble.

We consider three tasks of increasing difficulty: *perpetual exploration* in which the mobile entity has to traverse all edges of the (di)graph but is not required to stop, *exploration with stop* (often simply called *exploration* in this paper) in which starting at any node of the graph, the entity has to traverse all edges and stop at some node, and *exploration with return* in which starting at any node of the graph, the entity has to traverse all edges and stop at the starting node. An entity is said to perform one of the above tasks in a (di)graph, if starting at *any* node of this graph in the initial state, it completes this task in finitely many steps. (Notice that in the case of perpetual exploration, completing this task after finitely many steps means only traversing all edges, not necessarily stopping after it.) We compute the memory requirement of an exploration algorithm by measuring either the size of the robot memory in the robot model, or both the size of the agent memory and the size of the whiteboards in the agent model.

*Terminology.* A one-to-one and onto mapping $f$ between the two sets of nodes $V$ and $V'$ of two edge-labeled graphs $G = (V, E)$ and $G' = (V', E')$ is an *isomorphism* if, for every two nodes $x$ and $y$ in $V$: $(x, y) \in E \Leftrightarrow (f(x), f(y)) \in E'$, and the two edges have the same label. In the *map drawing* problem, the robot (resp., the agent) has to compute an edge-labeled graph $G$ such that $G$ is isomorphic to $X$ where $X$ is the unknown edge-labeled graph that the robot (resp., the agent) is exploring. Given two digraphs $G$ and $X$, and two nodes $u$ and $x$ of $G$ and $X$, respectively, we note $(G, u) \cong (X, x)$ if there exists an isomorphism $f$ between $G$ and $X$, such that $f(u) = x$.

## 3 Exploration of directed graphs under the robot model

We first prove a lower bound on the size of the robot memory. The proof uses the digraph *combination lock* (see, e.g., [15]) defined as follows.

**Definition 1.** *The* combination lock $L_{d,n}$ *is a regular digraph of out-degree $d$, and order $n$. The $n$ nodes $u_0, u_1, \ldots, u_{n-1}$ are connected as follows. For every $i < n-1$, node $u_i$ has one out-going edge pointing to $u_{i+1}$, and $d-1$ out-going edges pointing to $u_0$. Node $u_{n-1}$ has all its $d$ out-going edges pointing to $u_0$.*

**Theorem 1.** *Perpetual exploration in $n$-node digraphs of maximum out-degree $d > 2$ cannot be accomplished by a robot with less than $\Omega(n \log d)$ bits of memory, even if it is given up to $n$ pebbles. For $d = 2$, the result holds even if the robot is given up to $n/2$ pebbles.*

*Proof.* Let us given $d$ and $n$, and a robot able to explore all $n$-node digraphs of maximum out-degree $d$, thus including all distinct edge-labeled combination locks $L_{d,n}$. Assume that the robot is given $k$ pebbles, $k \geq 1$. A *full run* of the robot in $L_{d,n}$ is a run of the robot along the path $u_0, u_1, \ldots, u_{n-1}$. For every edge-labeled combination lock, place the robot at node $u_0$, and let us consider the state of the robot at $u_0$ before its first full run. (For each exploration, there are at least $d$ full runs since node $u_{n-1}$ must be reached at least $d$ times to traverse its $d$ out-going edges.) Since the $n$ nodes $u_i$, $i = 0, \ldots, n-1$, look identical to the robot up to the presence of a pebble, the ability to perform a full run is determined by the state of the robot just before leaving node $u_0$, and by the positions of the $k$ pebbles. There are $d^{n-1}$ different labelings of the edges $(u_i, u_{i+1})$, $i = 0, \ldots, n-2$, and $p = \sum_{i=0}^{k} \binom{i}{n}$ possible positions for the pebbles. Therefore, the robot must be able to be in at least $d^{n-1}/p$ different states at $u_0$. Thus it must have at least $\lceil (n-1)\log d - \log p \rceil$ bits of memory. Since $p \leq 2^n$, the result follows for $d > 2$. For $d = 2$, we use the fact that $\binom{a}{b} \leq (\frac{ae}{b})^b$ for $0 < a < b$, where $\ln e = 1$. Since $k \leq n/2$, we have $p \leq k(\frac{ke}{n})^n$, and thus $\log p \leq \log k + n \log(\frac{ke}{n})$. We have $k \leq n/2 < 2n/e$, thus $\log p < \log n + \alpha n$ with $\alpha < 1$, which completes the proof. $\square$

Note that our exploration algorithms use much less than $O(n)$ pebbles. One of them uses only one pebble, and the other uses $O(\log\log n)$ pebbles. We first sketch the description of an exploration algorithm, called `Test-all-maps`, satisfying the following:

**Theorem 2.** *Under the robot model, Algorithm* `Test-all-maps` *accomplishes exploration with stop in any digraph with a robot using one pebble, and whose memory does not exceed $O(nd\log n)$ bits in $n$-node digraphs of maximum out-degree $d$.*

We first sketch the description of Algorithm `Test-all-maps` and later prove that it satisfies the statement of Theorem 2.

*Algorithm* `Test-all-maps`. The robot successively tries every value for $n$, starting at $n = 1$. For a fixed $n$, the robot tries all possible maps of edge-labeled digraphs of order $n$. For a given map $G = (V, E)$, with $V = \{v_1, \ldots, v_n\}$, the robot proceeds as follows. Let $x$ be the current position of the robot in the unknown digraph $X$, and assume that the robot holds the pebble. The robot chooses node $v_1 \in V$, and tests whether it is standing on node $v_1$ of $G$, i.e., whether $(G, v_1) \cong (X, x)$. This is done thanks to the use of Procedure `Check-Consistency` that will be detailed later in the text. This procedure takes as input a graph $G$ and a node $v$ of $G$, and tests whether the robot is currently standing at $v$ in $G$. If the test succeeds, then the exploration stops. Otherwise, the robot chooses another node $v_2 \in V$, and tests whether $(G, v_2) \cong (X, x)$. Observe that during Procedure `Check-Consistency`, the robot moves in the graph $X$, and thus, since the procedure failed for $v_1$, there is no guarantee that the robot is yet standing at node $x$ of $X$. Hence, the robot uses a linear array `position`, of size $n$, such that `position`$[i]$ is the index $j$ of the node $v_j \in V$ where the robot would be now standing *if* the original position $x$ of the robot would satisfy $x = v_i$. Assuming $x$ is node $v_2$ of $G$, the robot would now stand on node $v_j$, $j =$ `position`$[2]$. The robot thus executes procedure `Check-Consistency` with input $(G, v_j)$. If the procedure succeeds, then the exploration stops. Otherwise, the robot chooses the next node $v_3$, and tests whether $(G, v_3) \cong (X, x)$. The robot thus executes procedure `Check-Consistency` with input $(G, v_j)$ where $j =$ `position`$[3]$. This process is carried on until either a test is eventually satisfied, or all nodes of $G$ have been exhausted. In the latter case, the robot picks the next map, and repeats the same scenario until if finds the map of the *a priori* unknown explored digraph. Now, we describe procedure `Check-Consistency`.

*Procedure* `Check-Consistency`$(G, u)$. Given the map of an edge-labeled graph $G = (V, E)$, with $n$ nodes and maximum out-degree $d$, and given a node $u$ of $G$, Procedure `Check-Consistency` checks whether the robot is currently standing at node $u$ of $G$, i.e., whether $(G, u) \cong (X, x)$ where $x$ is the current position of the robot in the unknown digraph $X$. The procedure borrows from [5] the technique of marking nodes of a cycle. However, this technique is implemented without the use of a large data-structure. More precisely, the robot assigns numbers, from 1 to $m$, to all the $m \leq nd$ edges of the map $G$, with the additional condition that the edge labeled 1 is out-going from $u$, and the edge labeled $m$ is incoming to $u$ (such an edge does exist because $G$ is strongly connected). Thus $E = \{e_1, \ldots, e_m\}$. For every $i \in \{1, \ldots, m-1\}$, the robot computes a shortest path $P_i$ in the map $G$ starting from the head of edge $e_i$ to the tail of edge $e_{i+1}$. During Procedure `Check-Consistency`, the paths $P_i$'s are computed on-line, and at most one path is stored at any given time in the robot memory. Let $C$ be the following closed walk starting and ending at $u$: $C = e_1, P_1, \ldots, e_{m-1}, P_{m-1}, e_m$. This walk will be traversed several times during the execution of Procedure `Check-Consistency`. $C$ is thus recomputed several times by the robot, and when $P_i$ is computed, the robot forgets about path $P_{i-1}$.

There are at most $n$ phases in Procedure `Check-Consistency`, one for every node of $G$. (The procedure assumes that the robot holds the pebble. Otherwise, the robot runs Procedure `Find-Pebble` described later.) For every phase there is a new *considered* node. During Phase $i$, the robot leaves $u$ with the pebble, and follows the edges of $C$ until it visits a node $v$ in $G$ that has not yet been *considered* during the $i-1$ previous phases. This node is marked *considered* on the map of $G$, and the pebble is dropped there. (Hence the first considered node is node $u$.) Then the robot carries on its walk guided by $C$ until, according to the map, it is back at $u$. Now, the robot traverses $C$ again. During its way along $C$, it checks the following property $\mathcal{P}$: the token is at the current node $x$ if and only if $x$ is the considered node $v$, according to the map of $G$. If property $\mathcal{P}$ is satisfied for every node of $C$, then the robot follows $C$ once again to bring the pebble back to $u$. If there is yet another node to be considered, then the next phase proceeds with this node. Otherwise the robot completes Procedure `Check-Consistency` as follows. It executes a last journey along $C$ to check whether there is equality between the degree of each node in the map $G$, and the degree of the corresponding node in the explored graph $X$. If so, the robot returns `success`. The robot turns into state `failure` as soon as it detects a problem at any step (e.g., the pebble is not where it should be, the pebble is where it should not be, the degree-sequences are different in the map and in the explored graph, etc.). As in [5], we have:

**Lemma 1.** *Given a robot at node $x$ of an anonymous digraph $X$, Procedure* `Check-Consistency` *returns* `success` *for $(G, u)$ if and only if $(G, u) \cong (X, x)$.*

If the robot loses the pebble during the execution of Procedure `Check-Consistency`$(G, u)$, then either the map $G$ is not correct, or it is correct but the robot was not at $u$. The robot then looks for the pebble by running the following procedure:

*Procedure* `Find-Pebble`. The robot computes a (non necessarily simple) closed path $P$ in the map $G$, visiting all nodes $\{v_1, \ldots, v_n\}$ of $G$. $P$ is computed on-line, e.g., $P$ is a sequence of sub-paths $P_i$ from $v_i$ to $v_{i+1}$, $i = 1, \ldots, n-1$, and the $P_i$'s are computed one after the other. The robot traverses the path $P$ several times, successively assuming that it starts from a node $v_i$ of the map,

$i = 1, \ldots, n$, and using an array `position` as in Procedure `Test-all-maps`. If the robot does not find the pebble, then the current map $G$ is for sure not a map of the explored digraph $X$. Therefore the robot considers the next map, and looks for the pebble in this new map using the same strategy as above. The robot proceeds this way until it finds the pebble when considering some map $H$. Once the pebble is found, the robot returns to the execution of Procedure `Test-all-maps`, and tests the current map $H$.

*Proof of Theorem 2.* We prove that the algorithm `Test-all-maps` can be implemented so not to use more than $O(nd \log n)$ bits of memory in $n$-node digraphs of maximum out-degree $d$. It is easy to list all edge-labeled digraphs with at most $n$ nodes and maximum out-degree $d$ using an array of $O(nd \log n)$ bits. Since the cycle $C = e_1, P_1, e_2, P_2, \ldots, e_{m-1}, P_{m-1}, e_m$ visiting all edges of a given map is computed on the fly, and since any path $P_i$ can be encoded by a sequence of at most $D$ labels, where $D$ is the diameter of $G$, we get that Procedure `Check-Consistency` requires $O(D \log d) \leq O(n \log d)$ bits of memory for the storage of $C$. The same holds in Procedure `Find-Pebble` for the storage of $P$. Thus the robot does not use more than $O(nd \log n)$ bits of memory in total. □

The algorithm `Test-all-maps` performs exploration in exponential time in the worst case (recall that time is counted as the number of edge traversals). Nevertheless, we can describe a variant of Algorithm `Explore-and-Map` presented in [5]. Although polynomial in time, `Explore-and-Map` is costly in term of memory space: a rough analysis shows that it requires a memory of $O(n^5 d \log n)$ bits. Our variant is called `Compacted-Explore-and-Map`. We summarize its performances by the following:

**Theorem 3.** *For any $n$-node digraph of maximum out-degree $d$, Algorithm* `Compacted-Explore-and-Map` *accomplishes exploration with stop in polynomial time under the robot model, with a robot using $O(\log \log n)$ pebbles and a memory of size $O(n^2 d \log n)$ bits.*

## 4 Exploration of directed graphs under the agent model

This section is dedicated to the agent model, i.e., nodes are given whiteboards on which the agent can read, erase, and write messages. The goal is to limit the sizes of both the agent memory, and the nodes' whiteboards. We first observe that exploration is impossible with an agent that performs obliviously, that is carrying no information from node to node.

**Theorem 4.** *Under the agent model, exploration with stop cannot be achieved by an agent with zero bit of memory.*

*Proof.* Assume for the purpose of contradiction that exploration with stop can be achieved by an agent with zero bit of memory. Then consider regular digraphs of out-degree $d \geq 2$. The content $\omega_i$ of the whiteboard of a node $u$ at the $i$th visit of that node by the agent is independent of $u$. Therefore $\omega_{i+1} = f(\omega_i)$, where

$f$ is a function that is uniquely defined by the program $\mathcal{P}$ of the agent. Thus the decision to stop depends only of the number of times the agent visits the same node. Let $k$ be the smallest integer such that $\omega_k$ is a final state. Let $L_{d,k+1}$ be the combination lock of out-degree $d$ and order $k+1$. To traverse all edges incoming to the first node $u_0$ of $L_{d,k+1}$, the agent must visit node $u_0$ at least $k+1$ times. Since it stops at the $k$th visit, not all edges have been traversed, and thus exploration is not completed, a contradiction. $\square$

**Theorem 5.** *Under the agent model, Algorithm* `DFS` *accomplishes exploration with return in any digraph using an agent with $O(1)$ bits of memory.* `DFS` *uses $O(\log d)$ bits of memory per node of out-degree $d$.*

We first describe Algorithm `Next-Port` that performs perpetual exploration in any digraph.

*Algorithm* `Next-Port`.

1. If the current node whiteboard is empty, then the agent writes 1 on it, and leaves the node through port 1;
2. Otherwise let $i$ be the integer written on the whiteboard, and let $d$ be the out-degree of the node. The program erases the whiteboard, writes $j = (i \bmod d) + 1$ on it, and the agent leaves the node through port $j$;

**Lemma 2.** *Algorithm* `Next-Port` *accomplishes perpetual exploration of any digraph using an agent with zero bit of memory, and uses $O(\log d)$ bits of memory per node of out-degree $d$.*

*Remark.* Algorithm `Next-Port` is used several times as a sub-routine in Algorithm `DFS`, and thus will be called with non-empty whiteboards. Nevertheless, it was shown [4] that Algorithm `Next-Port` is self-stabilizing and thus does not require the whiteboards to be initially empty to eventually perform correctly.

*Algorithm* `DFS`. Algorithm `DFS` performs a depth-first search (DFS) in the graph, using Algorithm `Next-Port` as a sub-routine. Nodes visited during the DFS are marked `visited` on their whiteboards. The last visited node is marked `last`. There is at most one node marked `last` during the execution of `DFS`. When exploration starts, the node on which is placed the agent is marked `visited` and `last`. It is also marked `root`. The path from the root to the last node is maintained thanks to port numbers that are stored on the whiteboards during the exploration. This path is called the *main path*. The agent leaves the root through port number 1. The DFS will proceeds by successively traversing incident edges of any node $u$ in order $1, 2, \ldots, d$ where $d$ is the out-degree of $u$. Before leaving the last node $u$, the port number through which the agent leaves is stored on $u$'s whiteboard. Assume that the agent then reaches node $v$. There are two cases, depending on whether node $v$ has been visited or not.

If $v$ has not yet been visited, it is marked `visited`. The agent then starts Algorithm `Next-Port` to find the root. From Lemma 2, this task will eventually succeed. From the root, the agent follows the main path and eventually reaches the last node $u$. There, the mark `last` is erased from $u$'s whiteboard. The agent then leaves $u$ by the port whose number is stored on $u$'s whiteboard, to reach $v$ again. Node $v$ is marked `last`. This sequence of instructions is repeated until the agent reaches a node $v$ that has been previously visited during the DFS.

If the agent reaches a node $v$ that is marked `visited`, it runs Algorithm `Next-Port` to find the root, and follows the main path from the root to the last node $u$. Once back at $u$, there are two sub-cases. If the port number $p$ of the edge leading to $v$ is smaller than the out-degree $d$ of $u$, then the agent leaves $u$ through port $p+1$, and repeats the same sequence of instructions as described before. If $p = d$, then the agent aims to backtrack. For that purpose, it runs Algorithm `Next-Port` to return to the root. The goal of the agent is to find the node of the main path that stands just before the node marked `last`. It marks the root as `next`, and proceed as follows. From the node marked `next`, the agent goes down one step along the main path to reach some node $w$. If $w$ is not marked `last`, the agent goes back to the root, follows the main path to the node marked `next`, erases `next` from the whiteboard of that node, moves to $w$, and mark $w$ as `next`. This is repeated until the agent finds the last node. Then it erases the mark `last` from the whiteboard of that node, goes back to the root using `Next-Port`, follows the main path until the node marked `next`, and replaces the mark `next` by `last`.

The process above is repeated until all edges out-going from the root have been visited, and the last backtrack leads to the root. Then the robot stops.

*Proof of Theorem 5.* During the execution of Algorithm `DFS`, the agent is clearly in a constant number of different states, hence a memory of $O(1)$ bits is enough for the agent. There is a constant number of marks written on each whiteboards. However, the storage of the port numbers of the main path, as well as the local storage used by Algorithm `Next-Port` (cf. Lemma 2) require whiteboards of size $O(\log d)$ bits. $\square$

*Remark.* It is possible to call Algorithm `Next-Port` only once (amortized), and to use it to construct a tree whose edges are pointing toward the root. Then returning to the root in Algorithm `DFS` takes a linear time after the first run of Algorithm `Next-Port`.

## 5 Conclusion and Further Works

Our algorithm `Test-all-maps` requires the storage of a test map of the unknown explored digraph. Graph exploration is however a weaker task than map drawing. One may thus expect to find an algorithm using a memory smaller than the size of a map. Another interesting direction of research is the investigation of compact exploration under the constraint that the algorithm must perform in polynomial time (i.e., the mobile entity must perform a polynomial number of edge-traversals). We described an algorithm for polynomial-time exploration, using a robot with a memory of size $O(n^2 d \log n)$ bits. This is however far from the $\Omega(n \log d)$ lower bound, and it would be interesting to determine the exact trade-off between time and memory space for graph exploration.

*Acknowledgement.* Both authors are supported by the Actions Spécifiques CNRS "Dynamo" and "Algorithmique des grands graphes", and by the project "PairA-Pair" of the ACI Masses de Données.

## References

1. S. Albers and M. R. Henzinger, Exploring unknown environments, SIAM Journal on Computing 29:1164-1188, 2000.
2. B. Awerbuch, M. Betke, R. Rivest and M. Singh, Piecemeal graph learning by a mobile robot, Proc. 8th Conf. on Comput. Learning Theory, pages 321-328, 1995.
3. A. Bar-Noy, A. Borodin, M. Karchmer, N. Linial, and M.Werman, Bounds on universal sequences, SIAM J. Computing, 18(2):268-277, 1989.
4. J. Beauquier, T. Hérault, and E. Schiller, Easy stabilization with an agent, In 5th Workshop on Self-Stabilizing Systems (WSS), Vol. 2194 of LNCS, pages 35-51, Springer-Verlag, 2001.
5. M. 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), pages 269-278, 1998.
6. M. 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), pages 75-85, 1994.
7. M. Betke, R. Rivest and M. Singh, Piecemeal learning of an unknown environment, Machine Learning 18:231-254, 1995.
8. L. Budach. Automata and labyrinths. Math. Nachrichten 86:195-282, 1978.
9. X. Deng and C. H. Papadimitriou, Exploring an unknown graph, Journal of Graph Theory 32:265-297, 1999.
10. K. Diks, P. Fraigniaud, E. Kranakis, and A. Pelc. Tree Exploration with Little Memory. To appear in Journal of Algorithms (see also proceedings of the 13th Annual ACM-SIAM Symp. on Discrete Algorithms (SODA), pages 588-597, 2002).
11. G. Dudek, M. Jenkins, E. Milios, and D. Wilkes. Robotic Exploration as Graph Construction. IEEE Transaction on Robotics and Automation 7(6):859-865, 1991.
12. C. Duncan, S. Kobourov and V. Kumar, Optimal constrained graph exploration. In 12th Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA), pages 807-814, 2001.
13. P. Fraigniaud, L. Gasieniec, D. Kowalski, and A. Pelc. Collective Tree Exploration. In 6th Latin American Theoretical Informatics Symposium (LATIN), Buenos Aires, April 2004.
14. A. Hemmerling. Labyrinth Problems. Teubner-Texte zur Mathematik, Bd. 114, Leipzig, 1989.
15. E. Moore. Gedanken-Experiments on Sequential Machines. In Automata Studies, pages 129-153, C. Shannon and J. McCarthy (Eds.), Princeton University Press, 1956.
16. P. Panaite and A. Pelc, Exploring unknown undirected graphs, Journal of Algorithms 33:281-295, 1999.
17. H.-A. Rollik. Automaten in planaren graphen. Acta Informatica 13:287-298, 1980.