PAPER

A Single Agent Exploration in Unknown Undirected Graphs with Whiteboards**

Yuichi SUDO[†,††a)], **Daisuke BABA**[†††], **Junya NAKAMURA**[††††], ***Nonmembers***, **Fukuhito OOSHITA**[††*], ***Member***, **Hirotsugu KAKUGAWA**[††], ***Nonmember***, ***and*** **Toshimitsu MASUZAWA**[††], ***Member***

SUMMARY We consider the exploration problem with a single agent in an undirected graph. The problem requires the agent starting from an arbitrary node to explore all the nodes and edges in the graph and return to the starting node. Our goal is to minimize both the number of agent moves and the memory size of the agent, which dominate the amount of communication during the exploration. We focus on the local memory called the whiteboard of each node. There are several exploration algorithms which are very fast (i.e. the exploration is completed within a small number of agent moves such as $2m$ and $m + 3n$) and do not use whiteboards. These algorithms, however, require large agent memory because the agent must keep the entire information in its memory to explore a graph. We achieve the above goal by reducing the agent memory size of such algorithms with using whiteboards. Specifically, we present two algorithms with no agent memory based on the traditional depth-first traversal and two algorithms with $O(n)$ and $O(n \log n)$ space of agent memory respectively based on the fastest algorithms in the literature by Panaite and Pelc [J. Alg., Vol.33 No.2, 1999].

key words: *graph exploration, mobile agent, whiteboard*

1. Introduction

We consider the exploration problem with a single agent in an undirected graph. We assume that the agent has to explore all the nodes and edges in the graph and return to the starting node. No a priori knowledge about the graph (e.g. topology or the number of nodes) is given to the agent. Exploration is one of the most fundamental problems in agent systems. For example, in a computer network, the agent can search data at unknown computer nodes by visiting all of the nodes, or can find broken communication channels by traversing all of the channels.

We focus on the total amount of communication required for graph exploration. There are two complexities related to the total amount of communication, the number of agent moves and the size of agent memory. In computer networks, a move of the agent between two computers is implemented as a send of a message containing the contents of the agent memory. Therefore, the total amount of communication during exploration over the whole graph is estimated by the product of the number of moves and the size of agent memory. Thus, it is important to minimize both of the two complexities.

In this paper, we propose exploration algorithms for arbitrary undirected graphs with a small number of moves and small agent memory. The algorithms are based on algorithms DFS and PP proposed in [10]. Algorithms DFS and PP work with a small number of moves but require large agent memory to memorize all information needed to explore the whole graph. We reduce the size of the agent memory by adopting a whiteboard (or local memory) at each node and storing the information distributedly over the whiteboards of nodes.

However, dispersion of the information on whiteboards makes the exploration over the whole network difficult. In DFS and PP, the agent always keeps all information it has ever collected (such as a partial map of the graph) in its memory. Hence, at any node, it can determine the next move using the full information. This is the best condition to minimize the number of moves. On the other hand, in our algorithms, the agent stores most of the information distributedly on whiteboards and carries only a small part of the information in its memory. At node v, the agent has to determine the next move depending on only the information stored on its memory and v's whiteboard. In this light, even if infinite space of the whiteboard is available at each node, there is a trade-off between the number of moves and the size of agent memory. This paper tackles an interesting question, "How much agent memory size can we reduce by utilizing whiteboards without increasing the number of moves?".

Related Works

Graph exploration has been widely studied in the literature. The study of graph exploration can be loosely classified by the anonymity and the topology of the graph. If all the nodes in a graph have unique identifiers, the graph is called *labeled*. On the contrary, if any node has no identifier, the graph is called *anonymous*.

When the graph is labeled, the exploration problem can be easily solved. For example, the agent can explore all nodes and edges in an arbitrary labeled graph with

Manuscript received September 20, 2014.
Manuscript revised May 27, 2015.

[†]The author is with NTT Secure Platform Laboratories, NTT Corporation, Musashino-shi, 180-8585 Japan.
[††]The authors are with the Graduate School of Information Science and Technology, Osaka University, Suita-shi, 565-0871 Japan.
[†††]The author is with TIS Inc., Tokyo, 160-0023 Japan.
[††††]The author is with Information and Media Center, Toyohashi University of Technology, Toyohashi-shi, 441-8580 Japan.
[*]Presently, with Graduate School of Information Science, Nara Institute of Science and Technology, Ikoma-shi, 630-0192 Japan.

[**]A preliminary extended abstract [1] of this paper appears in the proceedings of the third international workshop on reliability, availability, and security (WRAS 2010).

a) E-mail: sudo.yuichi@lab.ntt.co.jp

DOI: 10.1587/transfun.E98.A.2117

$2m$ moves by the simple depth-first search (abbreviated to DFS), where m is the number of edges in the graph. Panaite and Pelc [2] improved DFS and proposed an algorithm with a smaller number of moves: the agent explores an arbitrary undirected graph within $m + 3n$ moves, where n is the number of nodes in the graph. We denote this algorithm by PP. When the agent moves through an already traversed edge, the move is said to be a *penalty move*. While DFS requires m penalty moves, PP achieves $O(n)$ penalty moves, which is asymptotically optimal: When the agent begins the exploration at the center node of a line graph, n penalty moves are necessary to visit all nodes. The exploration of labeled directed graphs were studied in [3]–[5].

Exploring an anonymous graph is a more demanding task. Budach [6] proved that the agent cannot explore an arbitrary anonymous graph without the ability to mark nodes in some way. Therefore, anonymous graph exploration has been studied with assuming that the agent can mark nodes in some way [7]–[10] or that the topology of the graph is restricted [11], [12]. In the model where the agent can put and retrieve a finite number of *pebbles* on nodes, Bender et al. [7] analyzed the necessary and sufficient number of pebbles to explore an arbitrary directed graph with a polynomial number of moves. In the model where an unmovable token is located on the starting node and the agent can recognize the token, Chalopin et al. [8] presented an exploration algorithm for any arbitrary undirected graph. Dieudonné et al. [13] studied the extended model where (i) unmovable tokens are located on some nodes of the graph, and (ii) some of the tokens are Byzantine, that is, sometimes invisible to the agent. (The adversary decides the visibility of Byzantine tokens at each step.) The closest study to ours is the one of Fraigniaud et al. [9]. They minimized the agent memory size by using whiteboards and proposed an algorithm with a constant size agent memory which explores arbitrary directed graphs. However, they did not pay much attention to the number of moves; In a directed graph, the agent cannot perform backtracking, and this fact makes it very difficult to devise an exploration algorithm with a small number of moves. Das et al. [10] considered the exploration by k agents in the whiteboard model. They proposed an exploration algorithm that costs only $O(m \log k)$ agent moves. Here the agents have to accomplish not only graph exploration but also constructing the same map of the graph. Hence, the size of agent memory was not their concern.

The rotor-router model or the rotor-router mechanism, proposed by Priezzhev et al. [14], achieves a perpetual graph exploration i.e. repeated graph explorations without stop. In this mechanism, all edges incident to each node v are locally referred at v by ports $0, \ldots, \delta(v) - 1$ where $\delta(v)$ is the degree of node v, and each node v memorizes the port π_v through which the agent moves on the last exit from v. When the agent visits node v, it moves to the next node through port $(\pi_v + 1) \mod \delta(v)$. This simple mechanism achieves a perpetual graph exploration for any anonymous undirected graph. Regardless of initial value of π_v for all nodes v, from some point, the agent periodically moves through an Eulerian cycle, i.e. a cycle of length $2m$ where each edge (u, v) appears exactly twice, once in each direction. It takes $O(mD)$ moves until the agent starts perpetual traversals on the Eulerian cycle where D is the diameter of the graph [15]. Obviously, this mechanism is implemented with the agent of zero memory and the whiteboards on each node v with $O(\log \delta(v))$ space of memory. It is interesting that the rotor-router mechanism also makes a dynamic spanning tree; at any step after the agent enters an Eulerian cycle, the $n - 1$ edges indicated by π_v of all nodes v other than v_{cur} forms a spanning tree rooted at v_{cur} where v_{cur} is the node that the agent currently exists on.

As well as this paper, some studies aimed to minimize both the number of agent moves and the agent memory space on variant models of graph exploration. For example, Cohen et al. [16] showed that the agent can explore any anonymous undirected graph within $O(m)$ agent moves, with $O(1)$ space of agent memory, and without using whiteboards if the nodes of the graph are appropriately colored with only three colors. Menc et al. [17] focused on the model where the agent is not able to know the incoming port of the current node (i.e. the agent cannot know from which port it came into the current node), hence the agent cannot backtrack. In this model, they presented an exploration algorithm for any anonymous undirected graph such that the number of agent move is $O(m)$, the agent memory space is 1 (i.e. only one bit), and the whiteboard memory space of node v is $O(\log \delta(v))$.

A Universal Exploration Sequence (UXS) is a concept which is closely related to graph exploration. A UXS for a class $\mathcal{G}$ of graphs is a sequence of integers $x = (x_1, x_2, ..., x_k)$ that enables the agent to explore any graph G of $\mathcal{G}$ starting on any node v_1 in G. At the first move, the agent moves though the first edge[†] of the starting node v_1. At the i-th move $(2 \leq i \leq k + 1)$, the agent moves through the $((l_i + x_{i-1} - 1) \bmod \delta(v_i) + 1)$-th edge of the current node v_i, where l_i is the label of the edge $\{v_{i-1}, v_i\}$ at v_i. After the agent performs $k+1$ moves and reaches node v_{k+2}, it is guaranteed that all nodes and edges are explored by the agent. Reingold [18] develops an algorithm that computes, with $O(\log N)$ memory space, a UXS for any undirected graph with at most N nodes. This result directly implies that an agent exploration for any anonymous undirected graph can be performed with $O(\log N)$ space of agent memory when an upper bound N of the graph size is known to the agent. Furthermore, since the length of a UXS that Reingold's algorithm generates is polynomial of N, the exploration costs only a polynomial number of agent moves.

Our Contribution

In this paper, we present four algorithms WDFS1, WDFS2, WPP1 and WPP2. These algorithms are based on the existing algorithms DFS and PP [2]. We reduce their agent memory space with the help of whiteboards. Algorithms

[†]It is assumed that all edges incident to node v are locally labeled at v. (We shall see later in Sect. 2.)

Table 1 Performances of algorithms. ($\delta(v)$ is the degree of node v.)

	#moves	agent memory	memory of node v
DFS	$2m$	$O(m + n \log n)$	-
PP[2]	$m + 3n$	$O(m \log n)$	-
WDFS1	$2m$	0	$O(\delta(v))$
WDFS2	$4m$	0	$O(\log \delta(v))$
WPP1	$m + 3n$	$O(n)$	$O(n)$
WPP2	$m + 3n$	$O(n \log n)$	$O(\delta(v) + \log n)$
DFS2	$4m$	$O(n \log n)$	-
PP2	$m + 3n$	$O(m + n \log n)$	-

WDFS1 and WDFS2 are designed based on DFS, and algorithms WPP1 and WPP2 are designed based on PP.

The performances of these algorithms are summarized in Table 1. The two algorithms DFS and PP guarantee a small number of moves without using whiteboards but require relatively large agent memory. In algorithm DFS, the agent uses $O(m + n \log n)$ bits of its memory to remember all the already visited nodes and all the already traversed edges. In algorithm PP, the agent uses $O(m \log n)$ bits of its memory to remember the map of the explored part of the graph. Our proposed algorithms reduce these relatively large spaces of the agent memory by utilizing whiteboards. Algorithms WDFS1 and WDFS2 simulate DFS without requiring agent memory, and algorithms WPP1 and WPP2 simulate PP with $O(n)$ and $O(n \log n)$ bits of agent memory respectively. All of them, except for WDFS2, keep the same number of moves as the original existing algorithms while the number of moves of WDFS2 is at most twice as that of DFS. There are two trade-offs; one is between WDFS1 and WDFS2 and the other is between WPP1 and WPP2. Algorithm WDFS2 has a smaller whiteboard memory but costs a larger number of moves compared to WDFS1. Algorithm WPP2 has a smaller whiteboard memory but requires a larger agent memory compared to WPP1.

Our main contributions are WPP1 and WPP2. Original PP is the fastest in the literature but highly complicated algorithm. It is hard to distribute the contents of the agent memory of such a complicated algorithm adequately over the whiteboards. WPP1 and WPP2 defeat this difficulty. On the other hand, the aim of presenting WDFS1 and WDFS2 is to provide good examples showing how much agent memory space can be reduced by utilizing whiteboards. Since WDFS1 and WDFS2 are naive algorithms, they may have existed as informal results while the authors have not found a publication that presents them.

All algorithms, WDFS1, WDFS2, WPP1 and WPP2, work even on an anonymous graph while the two existing algorithms require unique labels of all the nodes. In WDFS1 and WDFS2, the agent does not use labels. In WPP1 and WPP2, the agent can easily assign unique labels to all nodes using $O(\log n)$ space of both the agent memory and the whiteboard of each node. However, for simplicity, we assume the unique node labels in this paper.

It is worth mentioning that a space-efficient algorithm that does not use whiteboards can be obtained from the space-efficient one that uses whiteboards. Assume that we have an algorithm that uses whiteboards, and let x, y and $z(v)$ denote the number of moves, the agent memory space and the whiteboard memory space of node v of the algorithm. Then, we can transform this algorithm into an algorithm that does not use whiteboards such that the number of moves and the agent memory space are x and $y + \sum_v z(v) + n \log n$ respectively. In the transformed algorithm, the agent simulates the moves of the original algorithm by remembering its original memory of size y and the contents of all the whiteboards. The additional $n \log n$ space is used to store the identifiers of n nodes.

By applying this transformation over WDFS2 and WPP2, we can obtain new no-whiteboard algorithms DFS2 and PP2 respectively (Table 1). Algorithm DFS2 reduces the agent memory of DFS from $O(m + n \log n)$ to $O(n \log n)$ at the expense of the number of moves (from $2m$ to $4m$). Algorithm PP2 decreases the agent memory of PP from $O(m \log n)$ to $O(m + n \log n)$ and does not increase any complexities of PP. Note that algorithms obtained by this transformation, such as DFS2 and PP2, require unique labels of each node.

2. Preliminaries

The environment is represented by a simple undirected connected graph $G = (V, E, p)$ where V is the set of nodes and E is the set of edges. We denote $|V|$ and $|E|$ by n and m respectively. The set $N(v)$ of neighboring nodes of v is defined by $\{u \in V \mid \{v, u\} \in E\}$. We denote the degree of v (i.e. the number of edges incident to v) by $\delta(v)$. A *port labeling* p is a collection of functions $(p_v)_{v \in V}$ where each p_v uniquely assigns a *port number* in $\{1, 2, \ldots, \delta(v)\}$ to every edge incident to node v. The agent needs these port numbers to distinguish edges incident to v when located at v. The port labeling p is locally independent: two port numbers $p_u(e)$ and $p_v(e)$ are independent for any edge $e = \{u, v\} \in E$. Every node $v \in V$ has the unique identifier $id(v) \in \mathbb{N}$. The size of the identifier space is polynomial in n, that is, $\max_{v \in V} id(v) \in O(n^c)$ is assumed for some constant c.

An *agent* $\mathcal{A} = (\mathcal{P}, \mathcal{M})$ consists of a constant-size program (algorithm) $\mathcal{P}$ and a finite memory $\mathcal{M}$. The agent exists on exactly one node $v \in V$ at any time, and moves through an edge incident to v. Program $\mathcal{P}$ specifies the movement of the agent. The *current node* that the agent currently exists on is denoted by v_{cur}. The previous node that the agent existed just before moving to v_{cur} is denoted by v_{pre}. Port p_{in} is defined as $p_{v_{\text{cur}}}(\{v_{\text{pre}}, v_{\text{cur}}\})$, via which the agent comes to v_{cur}. For simplicity, we suppose $v_{\text{pre}} = \mathit{null}$ and $p_{\text{in}} = 0$ at the beginning of exploration. Every node $v \in V$ has a *whiteboard* $w(v)$, which the agent can access freely at the visit of v. Agent memory $\mathcal{M}$ and whiteboard $w(v)$ consists of bit sequences. Initially, $\mathcal{M} = \varepsilon$ and $w(v) = \varepsilon$ for any $v \in V$ hold where ε represents the null string. Program $\mathcal{P}$ is invoked every time the agent visits a node or when the exploration begins. It takes 5-tuple $(\delta(v_{\text{cur}}), p_{\text{in}}, \mathcal{M}, w(v_{\text{cur}}), id(v_{\text{cur}}))$ as the input, and returns 3-

tuple $(p_{\text{out}}, \mathcal{M}', w')$ as the output. Here p_{out} is a port number of v, and both $\mathcal{M}'$ and w' are bit sequences. After obtaining the output from $\mathcal{P}$, the agent performs two substitutions $\mathcal{M} := \mathcal{M}'$ and $w(v_{\text{cur}}) := w'$, and then, moves to the next node through port p_{out}. The agent terminates when $p_{\text{out}} = 0$.

Exploration Problem

The agent starts exploration on an arbitrary node v_{st} of V, which we call the starting node. The goal of the agent is to traverse all edges[†] in the graph and return to the starting node. More precisely, we say that algorithm $\mathcal{P}$ solves the exploration problem if the following conditions hold for any graph G and any starting node v_{st}: (i) agent $\mathcal{A} = (\mathcal{P}, \mathcal{M})$ eventually terminates, (ii) the agent traverses every edge at least once before it terminates, and (iii) $v_{\text{cur}} = v_{\text{st}}$ holds when the agent terminates.

We measure efficiency of program (algorithm) $\mathcal{P}$ by three metrics: the number of moves, the agent memory space, and the whiteboard memory space. The first one is defined as the number of moves that the agent makes during the exploration. The memory spaces of the agent and the whiteboard on node v are defined as the maximum number of bits used for $\mathcal{M}$ and $w(v)$ respectively, taken over all of the exploration. All the above metrics are evaluated in the worst-case manner with respect to G and v_{st}.

In what follows, we say that a node (or edge) is *explored* when the node (edge) is already visited (traversed) at the time. Otherwise, the node (edge) is *unexplored*. We say that port $q \in \{1, 2, \ldots, \delta(v)\}$ of node v is explored if edge $p_v^{-1}(q)$ is explored[††]. A node v is *saturated* if all edges incident to v are explored.

Algorithm Description

This paper presents seven algorithm descriptions. Each description specifies agent variables, whiteboard variables and pseudo code of instructions the agent performs. Agent variables are stored in the agent memory. If initialization is specified for an agent variable (e.g. *flag*(v) in Algorithm 6), the variable is set to the specified value at the start of exploration. Whiteboard variables are stored in the whiteboard of every node. In our algorithms, whiteboards of all nodes have the same set of variables. If initialization is specified for a whiteboard variable (e.g. $p_{recent}(v)$ in Algorithm 2), the variable is set to the specified value when the agent makes the first visit on node v.[†††] If initialization is not specified for an agent or whiteboard variable, the value of the variable is undefined (or "DON'T CARE") before the first substitution to the variable.

3. Algorithms Based on DFS

In this section, we present algorithms WDFS1 and WDFS2, both of which are based on algorithm DFS. In the original DFS, the agent memorizes, for each node $v \in V$ and each port $q \in \{1, 2, \ldots, \delta(v)\}$, whether the agent has visited v before or not and whether the agent has moved through q from v or not. Clearly, $O(m + n \log n)$ space of agent memory is enough to store these information.

With these information, the agent performs the well known depth-first search on the graph. Figures 1 and 2 show the flow chart of DFS and an example of the depth first search on G respectively. As long as an unexplored port exists at the current node v_{cur}, the agent keeps on moving forward through the port (moves of type (a) in Fig. 1, and the 1-4th, 8-9th and 11th moves in Fig. 2). On moving forward, if the arrival node has already been visited before, the agent backtracks to the last visited node (moves of type (c) in Fig. 1, and the 5th and 12th moves in Fig. 2). If it is the first time to visit v_{cur}, the agent memorizes port p_{in} in a variable $p_{return}(v_{\text{cur}})$ of agent memory. The agent backtracks through this port $p_{return}(v_{\text{cur}})$ when the agent cannot find any unexplored port at v_{cur} (moves of type (b) in Fig. 1, and the 6-7th, 10th and 13-14th moves in Fig. 2). The agent terminates when it goes back to v_{st} and v_{st} is saturated. The number of agent moves is exactly $2m$ since the agent makes exactly one forward move and exactly one backtracking over every edge in the graph.

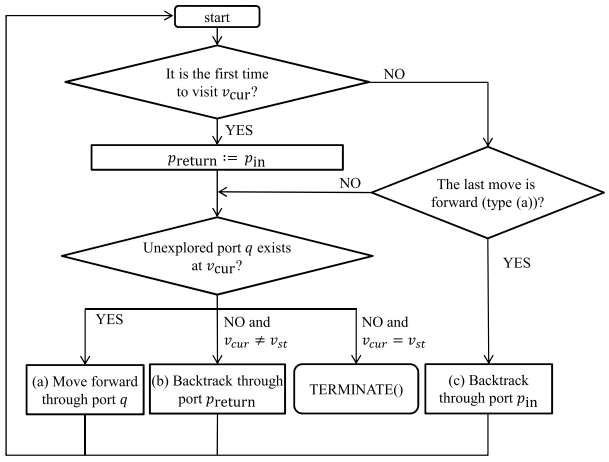

Fig. 1 The flow chart of DFS.

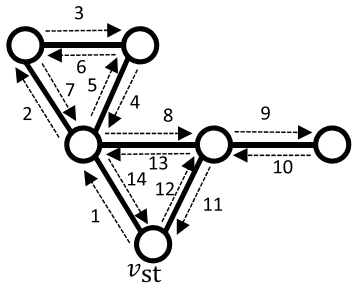

Fig. 2 Example of depth-first search on G. The numbers represent the order of moves. (e.g. Number "7" represents the 7-th move.)

[†]Then, it is guaranteed that all nodes are also visited.

[††]Function p_v^{-1} is the inverse of p_v. Hence, $p_v^{-1}(q)$ denotes edge e that satisfies $p_v(e) = q$.

[†††]The agent can easily detect the first visit on v because $w(v) = \epsilon$ holds. Then, the agent writes "INITIALIZATION DONE" in $w(v)$.

Algorithm 1 WDFS1

Variable in v's Whiteboard

$explored(v) \in \{0, 1\}$ // Initially, $explored(v) = 0$
$P_{exp}(v) \in 2^{\{1,2,\ldots,\delta(v)\}}$ // Initially, $P_{exp}(v) = \emptyset$
$p_{return}(v) \in \{0, 1, \ldots, \delta(v)\}$

Program

```
1: while true do                    // Always start from this line on visiting v
2:     FIRST_TIME_TO_VISIT := (explored(v_cur) = 0)?
3:     LAST_MOVE_IS_FORWARD := (p_in ∉ P_exp(v_cur))?
4:     explored(v_cur) := 1
5:     if p_in ≠ 0 then P_exp(v_cur) := P_exp(v_cur) ∪ {p_in}
6:     if FIRST_TIME_TO_VISIT is true then
7:         p_return(v_cur) := p_in                    // p_in = 0 if v_cur = v_st
8:     else if LAST_MOVE_IS_FORWARD is true then
9:         Move through p_in and go to Line 1         // backtracking (type (c))
10:    end if
11:    if ∃q ∈ {1, 2, ..., δ(v_cur)} \ P_exp(v_cur) then
12:        P_exp(v_cur) := P_exp(v_cur) ∪ {q}
13:        Move through port q                        // forward move (type (a))
14:    else if p_return(v_cur) ≠ 0 then               // v_cur ≠ v_st ⇔ p_return(v_cur) ≠ 0
15:        Move through port p_return(v_cur)          // backtracking (type (b))
16:    else
17:        TERMINATE()                 // v_cur = v_st holds and v_st is saturated
18:    end if
19: end while
```

Algorithm 2 WDFS2

Variable in v's Whiteboard

$explored(v) \in \{0, 1\}$ // Initially, $explored(v) = 0$
$p_{recent}(v) \in \{0, 1, \ldots, \delta(v)\}$ // Initially, $p_{recent}(v) = 0$
$p_{return}(v) \in \{0, 1, \ldots, \delta(v)\}$

Program (Main changes from WDFS1 are Lines 3, 10-12.)

```
1: while true do                    // Always start from this line on visiting v
2:     FIRST_TIME_TO_VISIT := (explored(v_cur) = 0)?
3:     LAST_MOVE_IS_FORWARD := (p_in ≠ p_recent(v_cur))?
4:     explored(v_cur) := 1
5:     if FIRST_TIME_TO_VISIT is true then
6:         p_return(v_cur) := p_in                    // p_in = 0 if v_cur = v_st
7:     else if LAST_MOVE_IS_FORWARD is true then
8:         Move through p_in and go to Line 1         // backtracking (type(c))
9:     end if
10:    if ∃q, p_recent(v_cur) < q ≤ δ(v_cur) ∧ q ≠ p_return(v_cur) then
11:        p_recent(v_cur) := (minimum number of such q)
12:        Move through port p_recent(v_cur)          // forward move (type (a))
13:    else if p_return(v_cur) ≠ 0 then               // v_cur ≠ v_st ⇔ p_return(v_cur) ≠ 0
14:        Move through port p_return(v_cur)          // backtracking (type (b))
15:    else
16:        TERMINATE()                 // v_cur = v_st holds and v_st is saturated
17:    end if
18: end while
```

WDFS1

In WDFS1, the agent completely simulates the move of DFS with no agent memory by using $O(\delta(v))$ space of $w(v)$ (Algorithm 1). To this end, it is sufficient for the agent to get the following information locally at node v: (i) whether v is explored or not, (ii) whether each port at v is explored or not, (iii) the value of $p_{return}(v)$ and (iv) whether the last move is forward (a move by Line 13) or not. The agent record the information of (i), (ii) and (iii) on whiteboard variables $explored(v)$, $P_{exp}(v)$ and $p_{return}(v)$ (Lines 4, 7 and 12). Clearly, $O(\delta(v))$ space of $w(v)$ is enough to store them. Furthermore, the agent can evaluate the condition of (iv) with only local information: the last move is forward if and only if edge $p_{v_{cur}}^{-1}(p_{in})$ (= $\{v_{pre}, v_{cur}\}$) was unexplored before the last move (Line 3). Note that the agent does not store any information on its own memory (Variables FIRST_TIME_TO_VISIT and LAST_MOVE_IS_FORWARD are used only for local computation and their values are never moved between nodes with the agent).

Theorem 1: WDFS1 solves the exploration problem for any undirected graph. The number of moves, the agent memory space, and the whiteboard memory space of node v are $2m$, 0, and $O(\delta(v))$ respectively.

WDFS2

In WDFS2 (Algorithm 2), the agent uses only $O(\log \delta(v))$ space on whiteboard $w(v)$. With such small space, the agent can store information (i) and (iii) but cannot record information (ii), that is, whether each port of v is explored or not. To explore all edges incident to v without this information, the agent uses whiteboard variable $p_{recent}(v)$, which memorizes the most recently used port to move *forward* from v. By using variable $p_{recent}(v)$, the agent makes a forward move through every port $p \in \{1, 2, \ldots, \delta(v)\}$ other than $p_{return}(v)$ in ascending order (Lines 10-12 in Algorithm 2). Note that, with $p_{recent}(v)$, the agent knows the set of explored ports of v through which the agent made forward moves from v to v's neighbors, but the agent does not know the set of explored ports of v through which the agent made forward moves from v's neighbors to v (and backtracks from v), which leads to more penalty moves than WDFS1. Variable $p_{recent}(v)$ is also used to evaluate (iv): the last move is a forward move if and only if $p_{in} \neq p_{recent}(v_{cur})$ holds (Line 3).

Indeed WDFS2 solves the exploration problem for any undirected graph, but it costs a larger number of agent moves than DFS (and WDFS1). The agent may traverse some edge four times in WDFS2 while it traverses every edge exactly twice in DFS. For example, consider the situation that the agent moves from node u to v through edge $\{u, v\}$, and then, v was already visited before (e.g. the fourth move in Fig. 2). Then, the agent knows that v is already explored and backtracks to node u. However, unlike in WDFS1, the agent does not record in $w(v)$ that port $p_v(\{u, v\})$ has been explored. If $p_{recent}(v) < p_v(\{u, v\})$ holds at this time, the agent will eventually move forward from v to u, and soon after that, backtrack from u to v. Thus, the agent traverses edge $\{u, v\}$ four times.

Theorem 2: WDFS2 solves the exploration problem for any undirected graph. The number of moves, the agent memory space, and the whiteboard memory space of each node v are at most $4m$, 0, and $O(\log \delta(v))$ respectively.

Proof: Since the agent memory space and the whiteboard memory space are clearly 0 and $O(\log \delta(v))$, we prove that

the number of moves is at most $4m$. The number of *forward* moves from node v is at most $\delta(v)$ since $p_{recent}(v)$ is monotonically increasing over the execution. Hence, the total number of forward moves is at most $2m$. Since the number of backward moves is equal to the number of forward moves, the number of moves of WDFS2 is at most $4m$. □

4. Algorithms Based on PP

In this section, we present algorithms WPP1 and WPP2 both of which are based on algorithm PP developed by Panaite and Pelc [2]. At first, we introduce the original algorithm PP.

4.1 Algorithm PP [2]

The algorithm PP solves the exploration problem in any undirected graph with no whiteboard and $O(m \log n)$ agent memory. During the execution of PP, the agent constructs the map of the explored part of graph G and stores it in its memory of $O(m \log n)$ space. The map $H = (V_H, E_H, p_H)$ consists of the explored nodes and edges, that is, $V_H = \{v \in V \mid v \text{ is explored}\}$ and $E_H = \{e \in E \mid e \text{ is explored}\}$. In addition, for every edge $\{u, v\} \in E_H$, the corresponding port numbers $p_u(\{u, v\})$ and $p_v(\{u, v\})$ are stored in p_H. The agent can easily construct the map H thanks to identifiers of nodes and port numbers that the agent can obtain locally. We omit the map construction part of algorithm PP from the pseudo code (Algorithm 3).

The algorithm PP, presented in Algorithm 3, consists of two parts: the main routine and subroutine Saturate(r). The agent can invoke Saturate(r) on node r. As we shall see later, Saturate(r) makes the agent explore all the unexplored edges incident to r and return to r. When the execution of Saturate(r) finishes, it is guaranteed that r is saturated and the agent exists on r. Importantly, if node v is saturated, map H is complete around v. Thus, for any port q of v, the agent knows which node the agent will reach if it leaves through q.

In the main routine, the agent performs the depth-first traversal on the graph with at most n invocations of Saturate(). When an exploration starts or every time the agent visits a node in the main routine, the agent marks v_{cur} on variable $mark(v_{\text{cur}})$ and invokes Saturate(v_{cur}) (Lines 2-3). If v_{cur} is saturated, the invocation is omitted. Thereafter, map H is complete around v_{cur} and the agent knows whether an unmarked neighboring node exists or not. If such a node u exists, it moves to u and memorizes port number $p_{\text{in}} = p_u(\{v_{\text{cur}}, u\})$ in variable $p_{parent}(u)$ for backtracking (Lines 4-6). If such node does not exist, the agent goes back through port $p_{parent}(v_{\text{cur}})$ (Lines 7-8). The agent repeats the above operations until the agent goes back to the starting node v_{st} and all the nodes around v_{st} are marked. Clearly, all the nodes and edges are explored when this repetition ends.

Let us observe how Saturate(r) saturates node r. All the moves of the agent are performed by two subroutines, GetForward(e) and GoBack(). During the execution of Saturate(r), the agent keeps the *return path* $RP = (v_0, e_1, v_1, e_2, \ldots, v_k)$, $v_0 = r$, $v_k = v_{\text{cur}}$ in its memory $\mathcal{M}$ (see the above part of Fig. 3), which is updated and used in GetForward(e) and GoBack(). The initial value of RP is (r). When GetForward(e) is invoked, the agent moves through edge e, and then appends (e, v_{cur}) to the tail of RP (Line 20). If this makes a cycle in RP, the agent deletes the cycle to shorten the return path RP. More precisely, if $v_{\text{cur}} = v_i$ holds for some i $(0 \le i \le k)$ in $RP = (v_0, e_1, v_1, \ldots, e_k, v_k, e, v_{\text{cur}})$, then the agent assigns $(v_0, e_1, v_1, \ldots, e_i, v_i)$ to RP (Line 22). When GoBack() is invoked, the agent retrieves and deletes the last two elements $(e_{k'}, v_{k'})$ from $RP = (v_0, e_1, v_1, \ldots, e_{k'}, v_{k'})$ and moves through edge $e_{k'}$ (Lines 24-25). Note that GoBack() is well-defined: GoBack() is invoked only when $v_{\text{cur}} \neq r$, and thus, $k' \ge 1$ is satisfied when GoBack() is invoked. The agent keeps on traversing unexplored edge e by invoking GetForward(e) as long as an unexplored edge incident to v_{cur} exists (Lines 13-14). The agent backtracks along RP

Algorithm 3 PP [2]

Variable in Agent

RP // a path from node r to v_{cur} used in Saturate(r)
H // a map of the graph constructed during exploration
$p_{parent}(v) \in \{1, 2, \ldots, \delta(v)\}$ // store the port leading to the parent node
$mark : V \to \{0, 1\}$ // Initially, $mark(v) = 0$ for any node v

{The conditions in Lines 3, 4, 10, 12 and 13 are evaluated with map H.}

Main Routine:

```
1: repeat
2:     mark(v_cur) := 1
3:     if v_cur is not saturated then Saturate(v_cur)
4:     if ∃u ∈ N(v_cur), mark(u) = 0 then
5:         Move through edge {v_cur, u}
6:         p_parent(v_cur) := p_in                    // v_cur = u at this line
7:     else
8:         Move through port p_parent(v_cur)
9:     end if
10: until v_cur = v_st and ∀u ∈ N(v_cur), mark(u) = 1
```

Saturate(r):

{$v_{\text{cur}} = r$ must hold when Saturate(r) starts.}

```
11: RP := (r)                                         // initialize return path RP
12: while not (v_cur = r and v_cur is saturated) do
13:     if unexplored edge e incident to v_cur exists then
14:         GetForward(e)
15:     else
16:         GoBack()
17:     end if
18: end while
```

GetForward(e):

{Let $RP = (v_0, e_1, v_1, \ldots, e_k, v_k)$ $(k \ge 0)$.}

```
19: Move through edge e
20: RP := (v_0, e_1, v_1, ..., e_k, v_k, e, v_cur) // Append (e, v_cur) to the tail of RP
21: if v_i = v_cur for some i (0 ≤ i ≤ k) then
22:     RP := (v_0, e_1, v_1, ..., e_i, v_i)          // Delete a cycle from RP
23: end if
```

GoBack():

{Let $RP = (v_0, e_1, v_1, \ldots, e_{k'}, v_{k'})$ $(k' \ge 1)$.}

```
24: RP := (v_0, e_1, v_1, ..., e_{k'-1}, v_{k'-1})    // Delete (e_k', v_k') from RP
25: Move through e_k'
```

by invoking GoBack() when no unexplored edge incident to v_{cur} exists(Line 16). Since the agent always maintains return path RP and invokes GetForward() only finite times (at most m), the agent eventually goes back to r, and eventually saturates r and terminates.

We analyze the number of moves required for exploration by PP in what follows. Note that the agent invokes GoBack() only when the current node is saturated. Consider the case that the agent has just performed GoBack() and backtracked from node v. Then, v is saturated and RP does not include v. Hence, after that, the agent never visits v during the execution of Saturate(u) for any $u \in V$. This means that the total number of invoking GoBack() is at most n in all the executions of Saturate() during the exploration. The number of invoking GetForward() is at most m. (Every invocation consumes one unexplored edge). Hence, the number of moves performed during the executions of Saturate() is at most $m + n$. The number of moves during the execution of the main routine is exactly $2(n-1)$ since the agent moves to an unmarked node (by Line 5) $n-1$ times and goes back to the parent node (by Line 8) $n-1$ times. Summing up these upper bounds, we see that the number of moves of PP is at most $m + 3n$.

Proposition 1 (Panaite and Pelc [2]): Without whiteboards, an agent $\mathcal{A} = (\mathsf{PP}, \mathcal{M})$ completes exploration for any undirected graph $G = (V, E)$ and any starting node $v \in V$ with at most $m + 3n$ moves.

4.2 Algorithms WPP1 and WPP2

In this section, we present our two algorithms, WPP1 and WPP2. Using whiteboards, these two algorithms simulate PP with less agent memory space. Algorithm WPP1 uses $O(n)$ agent memory space and $O(n)$ space on each whiteboard while algorithm WPP2 uses $O(n \log n)$ agent memory space and $O(\delta(v) + \log n)$ space on each whiteboard $w(v)$.

In both the two algorithms, we assume the followings:

(i) the agent knows if it already visited v_{cur} before or not,
(ii) the agent knows which ports of v are explored, and
(iii) $1 \le id(v) \le n$ holds for $v \in V$.

These assumptions do not lose generality. For assumption (i), it is sufficient that the agent marks the current node as "an explored node" on $w(v_{\text{cur}})$ at the start of exploration and every time it visits an unmarked node. For assumption (ii), it is sufficient that the agent marks ports $p_u(u, v)$ and $p_v(u, v)$ as "an explored port" on $w(u)$ and $w(v)$ respectively when it moves from u to v. For assumption (iii), it is sufficient that the agent records the number of times it visits unexplored nodes on agent variable *count* and stores *count* + 1 on $w(v)$ as the new label of v when it visits an unexplored node v. These procedure can be performed with $O(\log n)$ space of agent memory and $O(\log n + \delta(v))$ space of each whiteboard $w(v)$.

In the rest of this section, we illustrate how WPP1 and WPP2 simulate the subroutine Saturate(r) (Sect. 4.2.1) and the main routine of PP (Sect. 4.2.2).

4.2.1 Saturate(r) with Whiteboards

The algorithms to simulate Saturate(r) in WPP2 and WPP1 are shown in Algorithms 4 and 5 respectively. The difficulty of simulating Saturate(r) exists only in maintaining the return path RP in the subroutines GetForward() and GoBack(). Other instructions (i.e. Lines 11-18 in Algorithm 3) can be easily simulated with node identifiers and knowledge of explored ports (assumption (ii)) as in Lines 2-8 in Algorithm 4 and Lines 3-9 in Algorithm 5. Note that there is no difference in the agent moves between moving through unexplored edge e incident to v_{cur} in Algorithm 3 and moving through unexplored port q of v_{cur} in Algorithms 4 and 5. Therefore, the simulation succeeds if the agent maintains RP correctly. In what follows, we explain how the agent maintains RP with the agent memory restrictions in WPP1 and WPP2. Remind that $RP = (v_0, e_1, v_1, \ldots, e_k, v_k)$, $v_0 = r$, $v_k = v_{\text{cur}}$ is the path such that:

- $RP = (r)$ holds at the start of Saturate(r),
- when the agent moves from v to u in GetForward($\{v, u\}$), $(\{v, u\}, u)$ is appended to the tail of RP,
- if the above creates a cycle in RP, that is, $u = v_i$ holds for some i $(0 \le i < k)$, then the cycle is deleted and RP becomes $(v_0, e_1, v_1, \ldots, e_i, v_i)$, and
- when the agent backtracks from v_k to v_{k-1} in GoBack(), (e_k, v_k) is deleted from the tail of RP.

The two sub-routines, BeforeMove() and AfterMove() (Lines 9, 11 and 18 in Algorithm 4 and Lines 10, 12 and 20 in Algorithm 5) are invoked to construct a spanning tree for the main-routine and do not affect the moves of Saturate() at all. These sub-routines are explained in Sect. 4.2.2.

Procedure in WPP2

As for WPP2, the solution is easy (Algorithm 4). The agent can memorize the entire return path in its memory $\mathcal{M}$ of $O(n \log n)$ space. The agent keeps the return path $RP = (v_0, e_1, \ldots, e_k, v_k)$ as an alternating sequence of node identifiers and port numbers $RP_2 = (id(v_0), q_1, \ldots, q_k, id(v_k))$, where $q_i = p_{v_i}(e_i)$ holds for every i, $1 \le i \le k$. Since the length of the return path is at most $n-1$, agent memory of $O(n \log n)$ space is enough to keep RP_2. Since the agent has RP_2 in its memory, the agent can maintain RP_2 in exactly the same way as the agent does in PP: the agent appends the incoming port and the current node to RP_2 (Line 12), deletes a cycle if created (Lines 13-15), and deletes the last two elements (a port number and a node identifier) from RP_2 (Line 16). Hence, the agent in Saturate(r) of WPP2 moves in the exactly same way as the agent does in Saturate(r) of PP.

Lemma 1: The subroutine Saturate(r) of WPP2 simulates Saturate(r) of PP. It uses $O(n \log n)$ memory space of the agent memory and $O(\log n + \delta(v))$ memory space of $w(v)$.

Procedure in WPP1

Since WPP1 allows the agent to have only $O(n)$ memory

Algorithm 4 Saturate(r) in WPP2

Variable in Agent

RP_2 : an alternating sequence of node identifiers and port numbers

Saturate(r):
{$v_{\mathrm{cur}} = r$ must hold when Saturate(r) starts.}
1: $RP_2 := (id(r))$ // initialize return path RP
2: **while not** ($id(v_{\mathrm{cur}}) = id(r)$ **and** all ports of v_{cur} are explored) **do**
3: **if** an unexplored port q of v_{cur} exists **then**
4: GetForward(q)
5: **else**
6: GoBack()
7: **end if**
8: **end while**

GetForward(q):
{Let $RP_2 = (id(v_0), q_1, id(v_1), \ldots, q_k, id(v_k))$ $(k \geq 0)$.}
9: BeforeMove(q)
10: Move through port q
11: AfterMove()
12: $RP_2 := (id(v_0), q_1, id(v_1), \ldots, q_k, id(v_k), p_{\mathrm{in}}, id(v_{\mathrm{cur}}))$
13: **if** $id(v_i) = id(v_{\mathrm{cur}})$ for some i $(0 \leq i \leq k)$ **then**
14: $RP_2 := (id(v_0), q_1, id(v_1), \ldots, q_i, id(v_i))$ // Delete a cycle from RP_2
15: **end if**

GoBack():
{Let $RP_2 = (id(v_0), q_1, id(v_1), \ldots, q_{k'}, id(v_{k'}))$ $(k' \geq 1)$.}
16: $RP_2 := (id(v_0), q_1, id(v_1), \ldots, q_{k'-1}, id(v_{k'-1}))$
17: Move through port $q_{k'}$
18: AfterMove()

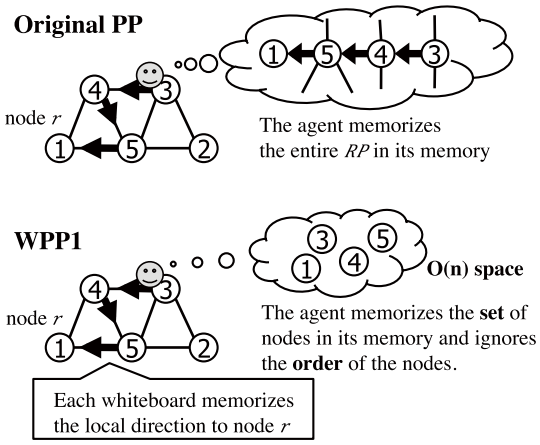

Fig. 3 How the return path is memorized in PP and WPP1.

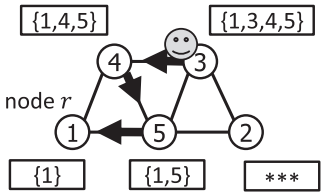

Fig. 4 Histories at whiteboards of nodes.

space, the agent cannot keep the return path RP. Thus, the information of RP should be dispersed among nodes. But this makes it difficult to detect a cycle in RP. Readers may think of the following simple mechanism for the cycle detection of RP: When the agent moves from u to v in GetForward(), it writes the mark as "this node belongs to RP" on $w(v)$, and when the agent backtracks from v, it deletes the mark from $w(v)$. The agent knows that a cycle is created in RP if it finds this mark just after the move in GetForward(). But this mechanism does not work well: to avoid subsequent false detection of a cycle, the agent has to remove the marks of the nodes in the deleted cycle, which requires additional moves of the agent.

Algorithm WPP1 maintains RP in a sophisticated way. With $O(n)$ space of agent memory, the agent cannot store the entire return path $RP = (v_0, e_1, v_1, \ldots, e_k, v_k)$ in its memory $\mathcal{M}$. Hence, WPP1 maintains RP by storing it separately on the agent memory and the whiteboards of all the nodes in RP (Fig. 3). The agent memorizes the *set* of node identifiers $\{id(v_j) \mid 0 \leq j \leq k\}$ in the variable RP_1 instead of the *sequence* of the identifiers. (Memory space required to store the set of identifiers is n since every identifier $id(v)$ satisfies $1 \leq id(v) \leq n$.) Each whiteboard $w(v_i)$ contains port number $p_{v_i}(e_i)$ in the variables $p_{return}(v_i)$. The set RP_1 is used to detect a cycle in RP in GetForward(e), and $p_{return}(v)$ is used to come back along RP in GoBack(). In addition, the set of identifiers $\{id(v_j) \mid 0 \leq j \leq i\}$ is stored in the whiteboard variable $hist(v_i)$ of node v_i (Fig. 4). This variable is used to identify the nodes constituting the cycle, which the agent should delete from RP_1.

The pseudo code of Saturate(r) in WPP1 is given in Algorithm 5. When Saturate(r) is invoked, the agent initializes both variables RP_1 and $hist(r)$ to $\{id(r)\}$ (Lines 1-2). When GetForward(q) is invoked, the agent moves through port q, and then checks whether this move creates a cycle on the return path. This check is easily done: a cycle is created if and only if the detecting condition $id(v_{\mathrm{cur}}) \in RP_1$ holds (Line 13). If the condition does not hold, the agent extends the return path by updating RP_1 and $p_{return}(v_{\mathrm{cur}})$ (Line 16 and 17). At the same time, the agent stores the copy of RP_1 in $hist(v_{\mathrm{cur}})$ (Line 18). If the detecting condition holds, the agent substitute $hist(v_{\mathrm{cur}})$ to RP_1 (Line 14). This substitution realizes deletion of the cycle. (Consider the case that the agent moves from node 3 to node 5 in Fig. 4. This move changes RP_1 from $\{1, 3, 4, 5\}$ to $\{1, 5\}$, which realizes deletion of the cycle $(5, 4, 3, 5)$.) When GoBack() is invoked, the agent removes node v_{cur} $(= v_k)$ from RP_1, and then moves through port $p_{return}(v_{\mathrm{cur}})$ $(= p_{v_k}(e_k))$.

Let $RP = (v_0, e_1, v_1, \ldots, e_k, v_k)$. We say that RP_1 is *correct* when $RP_1 = \{id(v_j) \mid 0 \leq j \leq k\}$ holds. We say that *whiteboard information is correct about histories* when $hist(v_i) = \{id(v_j) \mid 0 \leq j \leq i\}$ holds for all i $(0 \leq i \leq k)$. We say *whiteboard information is correct about local directions* when $p_{return}(v_i) = p_{v_i}(e_i)$ holds for all i $(1 \leq i \leq k)$.

Lemma 2: Let α be the total number of invocations of GetForward() and GoBack() in Saturate(r) and let i be any positive integer such that $1 \leq i \leq \alpha$. Then, RP_1 is correct at the start of the i-th invocation of the sub-routines in Saturate(r). Whiteboard information is also correct both about histories and about local directions at the start of the i-th invocation

Algorithm 5 Saturate(r) of WPP1

Variable in Agent
$RP_1 \in 2^{\{1,2,\ldots,n\}}$

Variables in v's Whiteboard
$p_{return}(v) \in \{1, 2, \ldots, \delta(v)\}$
$hist(v) \in 2^{\{1,2,\ldots,n\}}$

Saturate(r):
{$v_{\text{cur}} = r$ must hold when Saturate(r) starts.}
1: $RP_1 := \{id(r)\}$ // initialize return path RP
2: $hist(v_{\text{cur}}) = RP_1$ // Store the current RP_1 on the whiteboard
3: **while not** ($id(v_{\text{cur}}) = id(r)$ **and** all ports of v_{cur} are explored) **do**
4: **if** an unexplored port q of v_{cur} exists **then**
5: GetForward(q)
6: **else**
7: GoBack()
8: **end if**
9: **end while**

GetForward(q)
10: BeforeMove(q)
11: Move through port q
12: AfterMove()
13: **if** $id(v_{\text{cur}}) \in RP_1$ **then**
14: $RP_1 := hist(v_{\text{cur}})$ // Delete a detected cycle
15: **else**
16: $RP_1 := RP_1 \cup \{id(v_{\text{cur}})\}$ // Extend the return path
17: $p_{return}(v_{\text{cur}}) := p_{\text{in}}$
18: $hist(v_{\text{cur}}) := RP_1$ // Store the current RP_1 on the whiteboard
19: **end if**

GoBack()
18: $RP_1 := RP_1 \setminus \{v_{\text{cur}}\}$
19: Move through $p_{return}(v_{\text{cur}})$
20: AfterMove()

of the sub-routines in Saturate(r).

Proof: We prove the lemma by induction on $1 \leq i \leq \alpha$ (or ordinal number of invocation of GetForward() and GoBack() in Saturate(r)). The predicate of the lemma trivially hold for $i = 1$ since $RP = (r)$ and $RP_1 = hist(r) = \{id(r)\}$ holds at the start of the first invocation. Then, we prove the predicate for any $i > 1$ assuming that RP_1 and whiteboard information about histories and local directions are correct at the start of the t-th invocation for any t ($0 \leq t < i$). Let $RP = (v_0, e_1, v_1, \ldots, e_k, v_k)$ at the start of the $(i-1)$-th invocation. The following three cases of the $(i-1)$-th invocation exist: (i) The $(i-1)$-th invocation is GetForward() and this invocation creates a cycle in the return path, (ii) The $(i-1)$-th move is GetForward() and this invocation does not create a cycle in the return path, and (iii) The $(i-1)$-th invocation is GoBack(). It trivially holds for cases (ii) and (iii) that RP_1 and whiteboard information about histories and local directions are correct at the start of the i-th invocation. Hence, it suffices to prove the correctness for case (i). Let us consider that the agent moves from v_k to v at this invocation of GetForward() without loss of generality. In this case, $v_j = v$ holds for some j ($0 \leq j \leq k$). Thus, RP becomes $(v_0, e_1, v_1, \ldots, e_j, v_j)$ after the $(i-1)$-th move by the cycle deletion. Because of the correctness of RP_1, the agent detects the cycle at Line 13. Then, the agent substitutes $hist(v)$ to RP_1 by Line 14, the value of which is $\{id(v_0), id(v_1), \ldots, id(v_j)\}$ since $hist(v)$ is correct at the start of the $(i-1)$-th invocation. Thus, RP_1 is correct at the start of the i-th invocation. Whiteboard information about histories and local directions is also correct at the start of i-th invocation because the agent does not modify $hist()$ or $p_{return}()$ of any node at the $(i-1)$-th invocation. □

Lemma 3: The subroutine Saturate(r) of WPP1 simulates Saturate(r) of PP. It uses $O(n)$ space of the agent memory and $O(n)$ space of $w(v)$.

Proof: Since whiteboard information is always correct about local directions at the start of invocation of GoBack() by Lemma 2, the agent in WPP1 always selects the same edge to go back as in PP. Thus, the subroutine Saturate(r) of WPP1 simulates Saturate(r) of PP. Memory spaces of the agent and each node are $O(n)$ since the variables RP_1 and $hist(v)$ can be implemented as n-bit arrays. □

4.2.2 Main Routine with Whiteboards

Both WPP1 and WPP2 perform the main routine in the same way. In this section, we call them WPP collectively.

The goal of the main routine is to visit all the nodes with at most $2n$ moves. The original PP achieves this goal in a simple way: if unmarked node u exists in $N(v_{\text{cur}})$ then the agent moves to and marks u; Otherwise, the agent backtracks to the parent node of v_{cur} (in the depth first tree). In PP, the agent can determine whether an unmarked node exists in $N(v_{\text{cur}})$ or not by referring a map of the graph. However, in WPP, the agent does not have any map of the graph, thus it cannot detect existence of an unmarked node in $N(v_{\text{cur}})$ by the same way as PP. Hence, another mechanism is needed to visit all the nodes with at most $2n$ moves.

Roughly speaking, our solution is as follows: (i) During executions of Saturate(), we construct a spanning tree on the graph and store its edges on whiteboards, and (ii) in the main routine, the agent visits all the nodes with $2(n-1)$ moves by exploring the spanning tree. Before presenting the solution in detail, we introduce directed tree $D = (V_D, E_D)$, which expands dynamically as the agent moves: (i) Initially, $V_D = \{v_{\text{st}}\}$ and $E_D = \emptyset$, and (ii) every time the agent visits an unexplored node, node v_{cur} and directed edge $(v_{\text{pre}}, v_{\text{cur}})$ is added to V_D and E_D respectively. We say that node u is a child of node v if edge (v, u) exists in E_D, and define *children port set* of v as $C_D(v) = \{p_v(\{v, u\}) \mid (v, u) \in E_D\}$.

In WPP, the agent stores $C_D(v)$ in variable $P_{child}(v)$ of v's whiteboard. To maintain $P_{child}(v)$, the agent executes sub-routines BeforeMove() and AfterMove() in Saturate(r) (Lines 9, 11 and 18 in Algorithm 4 and Lines 10, 12 and 20 in Algorithm 5). The pseudo codes of BeforeMove() and AfterMove() are shown in Algorithm 6. The idea of these sub-routines is simple. Consider the case that the agent has just moved from node u to node v, and v is unexplored before the move. Then, by raising a flag on u (agent variable $flag(id(u))$), the agent remembers that u has a new child (Lines 2 and 4). When the agent visits u again after that,

the agent knows from the flag that u has a new child, and then adds $p_u(\{u, v\})$ to $P_{child}(u)$ (Lines 6-7). This addition is realized by storing port number $p_u(\{u, v\})$ on $w(u)$ every time it moves from u to v (Line 1). Note that we invoke BeforeMove() only before the move in GetForward() and not before the move in GoBack(), because the agent never finds an unexplored node at moves in GoBack().

Lemma 4: When the agent exists on node v, $P_{child}(v)$ equals to $C_D(v)$.

Proof: We prove the lemma by induction on the number of the agent's visits on v. At the time of the first visit on v, we have $P_{child}(v) = C_D(v)$ since both are empty sets. In the following, we prove that, under the assumption that $P_{child}(v) = C_D(v)$ holds at the time of the i-th visit on v, $P_{child}(v) = C_D(v)$ also holds at the time of the $(i + 1)$-th visit on v. Let $u \in N(v)$ be the node to which the agent moves just after the i-th visit on v.

Case 1: u is explored at the time of the i-th visit on v.
In this case, neither $P_{child}(v)$ nor $C_D(v)$ changes, and $P_{child}(v) = C_D(v)$ holds at the $(i + 1)$-th visit on v.
Case 2: u is unexplored at the time of the i-th visit on v.
In this case, $p_v(\{v, u\})$ is added to $C_D(v)$ when the agent arrives at u just after the i-th visit on v. The agent raises $flag(id(v))$ at this time, and the flag never goes down until the $(i + 1)$-th visit on v. Hence, at the $(i + 1)$-th visit on v, the agent recognizes the raised flag and adds $p_v(\{v, u\})$ to $P_{child}(v)$. Meanwhile, no other update can happen on $P_{child}(v)$ or $C_D(v)$. Thus, $P_{child}(v) = C_D(v)$ holds at the $(i+1)$-th visit on v. □

We denote D at the end of the exploration by $D_{\text{final}} = (V_{D_{\text{final}}}, E_{D_{\text{final}}})$. By the definition of D, $C_D(v) = C_{D_{\text{final}}}(v)$ always holds after node v is saturated. Therefore, by Lemma 4, $P_{child}(v) = C_{D_{\text{final}}}(v)$ holds after v is saturated.

The goal of the main routine, which is described in Algorithm 7, is to explore D_{final}. At Lines 3-9, we can assume v_{cur} is saturated thanks to Line 2, and hence, $P_{child}(v_{\text{cur}}) = C_{D_{\text{final}}}(v_{\text{cur}})$ holds. With whiteboard variables $P_{child}()$, the agent can explore D_{final} in the depth-first fashion: it moves through an unexplored port q as long as such a port q exists among $P_{child}(v_{\text{cur}})$ and it backtracks to the parent node of v_{cur} when such a port does not exist (Lines 3-9). All the nodes in the graph are explored when the agent goes back to v_{st} and all ports in $P_{child}(v_{\text{st}})$ are already explored. Then, the agent terminates the exploration.

Thus, the agent eventually visits all the nodes of D_{final} in the main routine of WPP. Hence, proving $V_{D_{\text{final}}} = V$ suffices to prove the correctness of WPP.

Lemma 5: D_{final} is a spanning tree on G. (i.e. $V_{D_{\text{final}}} = V$).

Proof: By definition of D, if node $v \in V_D$ is saturated, all nodes neighboring v are included in D. Since the agent visits and saturates all the nodes in D_{final}, all the nodes connected to v_{st} are included in D_{final}. □

The number of moves performed in the main routine is exactly $2(n - 1)$ since the agent moves every edge in D_{final} twice (once in each direction). And, the number of moves during all executions of Saturate() is at most $m + n$ (Lemmas 1 and 3). Consequently, we obtain the following two theorems.

Algorithm 6 BeforeMove(q) and AfterMove()

Variables in Agent

$flag : \{1, \ldots, n\} \to \{0, 1\}$: Initially, $\forall i \in [1, n]$, $flag(i) = 0$
$label_{pre} \in \{1, \ldots, n\}$

Variables in v's Whiteboard

$p_{recent}(v) \in \{1, 2, \ldots, \delta(v)\}$
$P_{child}(v) \in 2^{\{1,2,\ldots,\delta(v)\}}$: Initially, $\forall v \in V$, $P_{child}(v) = \emptyset$

BeforeMove(q):
{the agent is going to move through port q.}
1: $p_{recent}(v_{\text{cur}}) := q$
2: $label_{pre} := id(v_{\text{cur}})$

AfterMove()
3: **if** v_{cur} is not visited before **then**
4: $flag(label_{pre}) := 1$
5: **end if**
6: **if** $flag(id(v_{\text{cur}})) = 1$ **then**
7: $P_{child}(v_{\text{cur}}) := P_{child}(v_{\text{cur}}) \cup \{p_{recent}(v_{\text{cur}})\}$
8: $flag(id(v_{\text{cur}})) := 0$
9: **end if**

Algorithm 7 The main routine of WPP

Variables in v's Whiteboard

$P_{child}(v) \in 2^{\{1,2,\ldots,\delta(v)\}}$: read only
$P_{exp}(v) \in 2^{\{1,2,\ldots,\delta(v)\}}$: Initially, $\forall v \in V$, $P_{exp}(v) = \emptyset$
$p_{parent}(v) \in \{1, 2, \ldots, \delta(v)\}$

Main Routine:
1: **repeat**
2: **if** v_{cur} is not saturated **then** Saturate(v_{cur})
3: **if** port $q \in P_{child}(v_{\text{cur}}) \setminus P_{exp}(v_{\text{cur}})$ exists **then**
4: $P_{exp}(v_{\text{cur}}) := P_{exp}(v_{\text{cur}}) \cup \{q\}$
5: Move through port q
6: $p_{parent}(v_{\text{cur}}) := p_{v_{\text{cur}}}(\{v_{\text{pre}}, v_{\text{cur}}\})$
7: **else**
8: Move through port $p_{parent}(v_{\text{cur}})$
9: **end if**
10: **until** $v_{\text{cur}} = v_{\text{st}}$ **and** $P_{child}(v_{\text{cur}}) = P_{exp}(v_{\text{cur}})$

Theorem 3: WPP1 solves the exploration problem for any undirected graph. The number of moves, the agent memory space, and the whiteboard memory space of node v are at most $m + 3n$, $O(n)$, and $O(n)$ respectively.

Theorem 4: WPP2 solves the exploration problem for any undirected graph. The number of moves, the agent memory space, and the whiteboard memory space of node v are at most $m + 3n$, $O(n \log n)$, and $O(\delta(v) + \log n)$ respectively.

5. Conclusion

In this paper, we proposed four exploration algorithms in the whiteboard model. By using whiteboards, they solve the exploration problem for any undirected graph with a small number of moves and a small agent memory. In addition, our proposed algorithms give us efficient implemen-

tations of DFS and PP with smaller agent memory in the no-whiteboard model.

Acknowledgments

This work is supported in part by JSPS KAKENHI Grant Numbers 24650012, 26280022, and 26330084.

References

[1] Y. Sudo, D. Baba, J. Nakamura, F. Ooshita, H. Kakugawa, and T. Masuzawa, "An agent exploration in unknown undirected graphs with whiteboards," Proc. Third International Workshop on Reliability, Availability, and Security — WRAS'10, pp.1–6, 2010.
[2] P. Panaite and A. Pelc, "Exploring unknown undirected graphs," J. Algorithms, vol.33, no.2, pp.281–295, 1999.
[3] S. Albers and M.R. Henzinger, "Exploring unknown environments," Proc. Twenty-Ninth Annual ACM Symposium on Theory of Computing — STOC'97, pp.416–425, 1997.
[4] X. Deng and C.H. Papadimitriou, "Exploring an unknown graph," J. Graph. Theor., vol.32, no.3, pp.265–297, 1999.
[5] R. Fleischer and G. Trippen, "Exploring an unknown graph efficiently," Algorithms — ESA 2005, Lecture Notes in Computer Science, vol.3669, pp.11–22, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.
[6] L. Budach, "Automata and labyrinths," Math. Nachr., vol.86, no.1, pp.195–282, 1978.
[7] M.A. Bender, A. Fernández, D. Ron, A. Sahai, and S. Vadhan, "The power of a pebble: Exploring and mapping directed graphs," Inform. Comput., vol.176, no.1, pp.1–21, 2002.
[8] J. Chalopin, S. Das, and A. Kosowski, "Constructing a map of an anonymous graph: Applications of universal sequences," Principles of Distributed Systems, Lecture Notes in Computer Science, vol.6490, pp.119–134, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
[9] P. Fraigniaud and D. Ilcinkas, "Digraphs exploration with little memory," STACS 2004, Lecture Notes in Computer Science, vol.2996, pp.246–257, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.
[10] S. Das, P. Flocchini, S. Kutten, A. Nayak, and N. Santoro, "Map construction of unknown graphs by multiple agents," Theor. Comput. Sci., vol.385, no.1-3, pp.34–48, 2007.
[11] K. Diks, P. Fraigniaud, E. Kranakis, and A. Pelc, "Tree exploration with little memory," J. Algorithms, vol.51, no.1, pp.38–63, 2004.
[12] L. Gasieniec, A. Pelc, T. Radzik, and X. Zhang, "Tree exploration with logarithmic memory," Proc. Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, p.594, 2007.
[13] Y. Dieudonné and A. Pelc, "Deterministic network exploration by a single agent with byzantine tokens," Inform. Process. Lett., vol.112, no.12, pp.467–470, 2012.
[14] V.B. Priezzhev, D. Dhar, A. Dhar, and S. Krishnamurthy, "Eulerian walkers as a model of self-organized criticality," Phys. Rev. Lett., vol.77, no.25, pp.5079–5082, 1996.
[15] V. Yanovski, I.A. Wagner, and A.M. Bruckstein, "A distributed ant algorithm for efficiently patrolling a network," Algorithmica, vol.37, no.3, pp.165–186, 2003.
[16] R. Cohen, P. Fraigniaud, D. Ilcinkas, A. Korman, and D. Peleg, "Label-guided graph exploration by a finite automaton," ACM Trans. Algorithms, vol.4, no.4, pp.1–18, 2008.
[17] A. Menc, D. Pająk, and P. Uznański, "On the power of one bit: How to explore a graph when you cannot backtrack?," arXiv preprint arXiv:1502.05545, 2015.
[18] O. Reingold, "Undirected connectivity in log-space," J. ACM, vol.55, no.4, pp.1–24, 2008.

Yuichi Sudo received the B.E., M.E., and D.E. degrees from Osaka University in 2009, 2011 and 2015 respectively. He works at NTT Corporation since 2011 as a researcher. His research interests include distributed algorithms, graph theory, and network security.

Daisuke Baba received the B.E. and M.E. degrees from Osaka University in 2008 and 2010 respectively. He currently works at TIS Inc. His research interests include distributed algorithms and graph theory.

Junya Nakamura received the B.E. and M.E. degrees from the Department of Knowledge-Based Information Engineering, Toyohashi University of Technology in 2006 and 2008 respectively, and D.E. degree in computer science from Osaka University in 2014. He is now a project assistant professor of Information and Media Center, Toyohashi University of Technology. His research interests include distributed algorithms and dependability of distributed systems, especially Byzantine fault tolerance.

Fukuhito Ooshita received the M.E. and D.I. degrees in computer science from Osaka University in 2002 and 2006. He had been an assistant professor in the Graduate School of Information Science and Technology at Osaka University during 2003–2015. He is now an associate professor of Graduate School of Information Science, Nara Institute of Science and Technology (NAIST). His research interests include parallel algorithms and distributed algorithms. He is a member of ACM, IEEE, and IPSJ.

Hirotsugu Kakugawa received the B.E. degree in engineering in 1990 from Yamaguchi University, and the M.E. and D.E. degrees in information engineering in 1992, 1995 respectively from Hiroshima University. He had been an assistant professor, an instructor, and an associate professor of Hiroshima University during 1993–2005. He is currently an associate professor of Osaka University. He is a member of the IEEE Computer Society and the Information Processing Society of Japan.

Toshimitsu Masuzawa received the B.E., M.E. and D.E. degrees in computer science from Osaka University in 1982, 1984 and 1987. He had worked at Osaka University during 1987–1994, and was an associate professor of Graduate School of Information Science, Nara Institute of Science and Technology (NAIST) during 1994–2000. He is now a professor of Graduate School of Information Science and Technology, Osaka University. He was also a visiting associate professor of Department of Computer Science, Cornell University between 1993–1994. His research interests include distributed algorithms, parallel algorithms and graph theory. He is a member of ACM, IEEE, IEICE and IPSJ.