An Agent Exploration in Unknown Undirected Graphs with Whiteboards

Yuichi Sudo
Graduate School of Information Science and Technology, Osaka University
y-sudou@ist.osaka-u.ac.jp

Daisuke Baba
Graduate School of Information Science and Technology, Osaka University
d-baba@ist.osaka-u.ac.jp

Junya Nakamura
Graduate School of Information Science and Technology, Osaka University
junya-n@ist.osaka-u.ac.jp

Fukuhito Ooshita
Graduate School of Information Science and Technology, Osaka University
f-oosita@ist.osaka-u.ac.jp

Hirotsugu Kakugawa
Graduate School of Information Science and Technology, Osaka University
kakugawa@ist.osaka-u.ac.jp

Toshimitsu Masuzawa
Graduate School of Information Science and Technology, Osaka University
masuzawa@ist.osaka-u.ac.jp

ABSTRACT

We consider the exploration problem with a single agent in undirected graphs. Starting from an arbitrary node, the agent has 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 space of the agent, which dominate the amount of communication during the exploration. In our setting, the agent is allowed to use the local memory called the whiteboard on each node (the whiteboard model), while most of existing exploration algorithms do not use the whiteboard (the no-whiteboard model). In the no-whiteboard model, the agent must memorize in its memory all information needed to explore the graph, and thus designing an exploration algorithm of small agent memory is difficult. In this paper, by allowing the agent to use whiteboards, we present four exploration algorithms such that both the number of agent moves and the agent memory space are small.

Keywords

graph exploration, mobile agent, whiteboard

1. INTRODUCTION

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

Our goal is to design an exploration algorithm with a small number of moves and a small memory size of an agent. We achieve this goal by adopting the *whiteboard model*. In the whiteboard model, every node is equipped with a local memory called the *whiteboard*, and the agent can freely read and write the content of the whiteboard while it stays at the node. On the contrary, the *no-whiteboard model* does not assume the existence of whiteboards. In the no-whiteboard model, the agent must memorize in its memory all information needed to explore the whole graph, and thus designing an exploration algorithm of small agent memory is difficult. On the other hand, in the whiteboard model, the information can be stored distributedly on nodes of the graph, and thus there is a possibility to explore the whole graph with small memory of the agent. In this paper, we present four exploration algorithms in the whiteboard model. By using whiteboards, these algorithms reduce the agent memory space of two existing algorithms which guarantee a small number of moves but require a relatively large agent memory space. Thus, our proposed algorithms realize both a small number of moves and a small memory of the agent.

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 explores all nodes and edges in the graph with $2m$ moves by the simple depth-first search (We denote the number of edges in the graph by m. In what follows, we denote the depth-first search by DFS). Panaite and Pelc [10] improve DFS and proposed faster algorithm, with which the agent explores an arbitrary

WRAS '10 Zurich, Switzerland

undirected graph within $m+3n$ moves (We denote the number of nodes in the graph by n. In what follows, we denote this algorithm by PP). When the agent moves through already traversed edge, the move is said to be a *penalty move*. Algorithm PP achieves $O(n)$ penalty moves, which is asymptotically optimal: Consider the case that the agent begins the exploration at the center node of a line graph. The exploration of labeled directed graphs is studied in [1, 5, 7].

Exploring an anonymous graph is more demanding task. Budach[3] proves that the agent cannot explore an arbitrary anonymous graph without the ability to mark nodes in some way. Therefore, anonymous graph exploration is studied with assuming that the agent can mark nodes in some way[2, 4, 8] or with restricting the topology of the graph[6, 9]. In the model where the agent can put and retrieve a finite number of *pebbles* on nodes, Bender et al.[2] analyze a necessary and sufficient number of pebbles to explore any arbitrary directed graph with a polynomial number of moves. Fraigniaud et al.[8] proposed an algorithm with a constant-size agent memory which explores any arbitrary directed graphs in the whiteboard model. Their interest is to minimize the size of agent and whiteboard memory, and they do not care the number of agent moves (although their algorithm terminates in polynomial time). Das et al.[4] consider the exploration by k agents in the whiteboard model. They propose 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 maps of the graph. Hence, the size of agent memory is not the authors' concern.

Our Contribution.

In this paper, we present four algorithms WDFS1, WDFS2, WPP1 and WPP2 in whiteboard model. Algorithms WDFS1 and WDFS2 are designed based on DFS, and algorithms WPP1 and WPP2 are designed based on PP [10].

Generally, it is not an easy task to simulate an algorithm designed in no-whiteboard model with an algorithm of smaller agent memory in whiteboard model. Let A be an algorithm that is designed in no-whiteboard model and requires large agent memory. In the execution of A, the agent can remember global information about the graph in its sufficiently large memory. Hence, at any node, the agent can determine the next move depending on the global information. On the other hand, in the execution of the simulating algorithm, the agent cannot remember such global information in its memory due to the lack of space. Hence, at node v, the agent has to determine the next move depending on only local information stored on v's whiteboard.

The performances of these algorithms are summarized in Table 1. The two existing algorithms guarantee a small number of moves without using whiteboards but require relatively large spaces of the agent memory. In algorithm DFS, the agent use $\Theta(m + n \log n)$ bits of its memory to remember all the already visited nodes and all the already traversed edges. In algorithm PP, the agent use $\Theta(\mathrm{MAP})$ bits of its memory to remember the map of the explored parts of the graph, where $\mathrm{MAP} = \min(m \log n, n^2)$. Our proposed algorithms reduce these relatively large spaces of the agent memory by utilizing whiteboards. Algorithms WDFS1 and WDFS2 simulate DFS with no agent memory, and algorithms WPP1 and WPP2 simulate PP with $\Theta(n)$ and $\Theta(n \log n)$ bits of agent memory respectively. The algorithms other than WDFS2 keep the same number of moves as the original existing algorithms while the number of moves of WDFS2 is at most twice as those of DFS. Algorithms WDFS1 and WDFS2 have a trade-off relation, and so do 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 WDFS1.

Table 1: Performances of algorithms. ($\delta(v)$ is the degree of node v. $\mathrm{MAP} = \min(m \log n, n^2)$.)

	#moves	agent memory	memory of node v
DFS	$2m$	$\Theta(m + n \log n)$	-
PP[10]	$m + 3n$	$\Theta(\mathrm{MAP})$	-
WDFS1	$2m$	0	$\Theta(\delta(v))$
WDFS2	$4m$	0	$\Theta(\log \delta(v))$
WPP1	$m + 3n$	$\Theta(n)$	$\Theta(n)$
WPP2	$m + 3n$	$\Theta(n \log n)$	$\Theta(\delta(v) + \log n)$

All our algorithms do not require labels of nodes while the two existing algorithms require them. In DFS1 and DFS2, the agent do not use the labels, and in PP1 and PP2, the agent can easily assigns unique labels to all nodes using $O(\log n)$ space of both the agent and the whiteboard of each node. However, for simplicity, the existence of label on each node is assumed as the model. (we shall see in the next section.)

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 and the set $I(v)$ of edges incident to v is defined by $\{u \in V \mid \{v, u\} \in E\}$ and $\{\{v, u\} \in E \mid u \in V\}$ respectively. We denote $|I(v)|$, the degree of v, by $\delta(v)$. A *port labeling* p is a collection of functions $(p_v)_{v \in V}$ where each $p_v : I(v) \to \{1, 2, \ldots, \delta(v)\}$ uniquely assigns *port number* to every edge incident to node v. The agent needs these port numbers to distinguish edges in $I(v)$ when located at v. The port labeling p is locally independent: two port numbers $p_u(e)$ and $p_v(e)$ may differ for edge $e = \{u, v\} \in E$. The node u neighboring to node v such that $p_v\{v, u\} = q$ is called the *qth neighbor* of v and is denoted by $N(v)[q]$. All nodes $v \in V$ have the unique identifiers $id(v) \in \mathbb{N}$. The size of the identifier space is polynomial, 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 $\mathcal{P}$ has complete control over the move of the agent. The *current node* that the agent currently exists on is denoted by v_{cur}. The previous node that the agent existed before moving to v_{cur} is denoted by v_{pre}. Port p_{in} is defined as $p_{v_{\mathrm{cur}}}(\{v_{\mathrm{pre}}, v_{\mathrm{cur}}\})$, via which the agent comes to v_{cur}. For simplicity, we suppose $v_{\mathrm{pre}} = \mathit{null}$ and $p_{\mathrm{in}} = 0$ 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. The content of $\mathcal{M}$ and $w(v)$ are bit sequences. Initially, $\mathcal{M} = \varepsilon$ and $w(v) = \varepsilon$ hold for any $v \in V$ where ε represents the null string. Program $\mathcal{P}$ is invoked every time the agent finishes its move or when the exploration begins. It takes 5-tuple $(\delta(v_{\rm cur}), p_{\rm in}, \mathcal{M}, w(v_{\rm cur}), id(v_{\rm cur}))$ as the input, and returns 3-tuple $(p_{\rm out}, \mathcal{M}', w')$ as the output. Here $p_{\rm out}$ is a port number of v, and both $\mathcal{M}'$ and w' are arbitrary bit sequences. Obtaining the output from $\mathcal{P}$, the agent performs two substitutions $\mathcal{M} := \mathcal{M}'$ and $w(v_{\rm cur}) := w'$, and then, moves to the next node through port $p_{\rm out}$. The agent terminates when $p_{\rm out} = 0$.

Exploration Problem.

The *starting node*, denoted by $v_{\rm st}$, is the node on which the agent exists at the beginning of exploration. The goal of the agent is to traverse all edges[1] 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 regardless of graph G and starting node $v_{\rm st}$: (i) agent $\mathcal{A} = (\mathcal{P}, \mathcal{M})$ eventually terminates, (ii) the agent traverses every edge at least once until it terminates, and (iii) $v_{\rm cur} = v_{\rm st}$ holds when the agent terminates.

We measure the 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 the moves that the agent made during the exploration. The memory space of the agent and the whiteboard on node v are defined as the maximum numbers of bits, during the exploration, of $\mathcal{M}$ and $w(v)$ respectively. All the above metrics is evaluated in the worst-case manner with respect to G and v.

In what follows, a node (or edge) is called *explored* when the node (edge) is already visited (traversed) at the time. Otherwise, the node (edge) is called *unexplored*. A node v is *saturated* if all edges in $I(v)$ are explored.

3. ALGORITHMS BASED ON *DFS*

In this section, we present algorithms WDFS1 and WDFS2, both of which are based on algorithm DFS.

The original DFS, illustrated in Algorithm 1, is simple. Here, we call a move invoked by Line 7 a *forward move* and call a move invoked by another line a *backward move* (or *backtracking*). The agent keeps on making a forward move to a new node as long as an unexplored edge exists in $I(v_{\rm cur})$ (Line 7). However, if the new node is already visited, the agent backtracks to the last visited node (Line 4), and resumes moving forward. If the new node is not visited before, the agent remembers port $p_{\rm in}$ in a variable $port_{return}(v)$ of agent memory (Line 2), and continues to move forward. The agent backtracks through port $port_{return}(v_{\rm cur})$ when the agent cannot find any unexplored edge in $I(v_{\rm cur})$ (Line 9). Eventually, $v_{\rm st}$ is saturated and then the agent terminates the exploration. The number of agent moves is exactly $2m$ since the agent makes exactly one forward move and exactly one backward move over every edge in the graph.

Algorithm 1 DFS

Variable in Agent

$port_{return}(v) \in \{1, 2, \ldots, \delta(v)\}$

Program

```
1: if v_cur is not visited before then
2:    port_return(v_cur) := p_in        // p_in = 0 when v_cur = v_st
3: else if the last move is performed in Line 7 then
4:    Move through p_in                 // Backtrack to v_pre
5: end if
6: if unexplored edge e ∈ I(v_cur) exists then
7:    Move through edge e
8: else
9:    Move through port port_return(v_cur)
                  // When v_cur = v_st, the agent stops
                     since port_return(v_cur) must be 0
10: end if
```

WDFS1.

In WDFS1, the agent completely simulates the move of DFS with no memory space of agent by using $\Theta(\delta(v))$ space of $w(v)$. To this end, it is sufficient for the agent to get the following information locally at node v: (i) whether v is visited or not, (ii) whether every $e \in I(v)$ is visited or not, (iii) the value of $port_{return}(v)$ and (iv) whether the last move is forward or not. The agent can easily records on $w(v)$ the information of (i), (ii) and (iii) with $O(\delta(v))$ space. Furthermore, the agent can evaluate the condition of (iv) with only local information: the last move is forward if and only if edge $p_v^{-1}(p_{\rm in})$ $(= \{v_{\rm pre}, v_{\rm cur}\})$ was unexplored just before the last move.

WDFS2.

In WDFS2, only $O(\log \delta(v))$ space is available on whiteboard $w(v)$. With $O(\log \delta(v))$ space, the agent can still store information (i) and (iii) on $w(v)$ but cannot record information (ii). To explore all edges in $I(v)$ without knowing which edges in $I(v_{\rm cur})$ are explored, the agent maintains a whiteboard variable $port_{recent}(v)$, which memorizes the most recently used port to move forward from v. By using variable $port_{recent}(v)$, the agent moves forward through all edges $e \in I(v)$ other than $\{p_v^{-1}(port_{return}(v))\}$ in ascending order of port numbers $p_v(e)$. Variable $port_{recent}(v)$ are also used to evaluate condition (iv): the last move is forward if and only if $p_{\rm in} \neq port_{recent}(v_{\rm cur})$ holds or v is not visited before.

In DFS (and WDFS1), the agent performs a forward move only through an unexplored edge. On the other hand, in WDFS2, the agent may move forward through an explored edge. Letting $p_w(\{v, w\}) = 5$, consider the situation that the agent makes a forward move from node v to w when $port_{recent}(w) = 3$. At this time, the agent cannot record that edge $\{v, w\}$ (port 5) is already explored, and consequently, the agent will eventually make a forward move from w to v. Thus, the agent makes an additional move in WDFS2. However, the number of agent moves is bounded above by $4m$ (The proof is omitted due to the lack of space).

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[10]. First of all, we introduce the original algorithm PP.

4.1 Algorithm *PP [10]*

The algorithm PP solves the exploration problem in any undirected graph with no whiteboard and $O(\text{MAP})$ agent memory $(\text{MAP} = \min(m \log n, n^2))$. During the execution

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

Algorithm 2 PP

Main Routine:

```
1: Saturate(v_cur)                    // Saturate the start node v_st
2: while true do
3:    if ∃u ∈ N(v_cur), u ∉ V_s then
4:       Move through edge {v_cur, u}          // Move to u
5:       port_parent(v_cur) := p_{v_cur}({v_pre, v_cur})
6:       Saturate(v_cur)        // Incorporate v_cur = u into S
7:    else if v_cur ≠ v_st then
8:       Move through port port_parent(v)
9:    else
10:      STOP()        // At this time, S is a spanning tree
11:   end if
12: end while
```

Saturate(r):

```
13: while not (v_cur = r and v_cur is saturated) do
14:    if non-visited edge e ∈ I(v_cur) exists then
15:       GetForward(e)
16:    else
17:       GoBack()
18:    end if
19: end while
```

{The conditions in Lines 3, 13 and 14 are evaluated with map H.}

of PP, the agent memorizes the map $H = (V_H, E_H)$ of graph G. The map H consists of explored nodes and edges, that is, $V_H = \{v \in V \mid v \text{ is explored}\}$ and $E_H = \{e \in E \mid e \text{ is explored}\}$. The agent can easily construct the map H thanks to identifiers of nodes. We omit the map construction part of algorithm PP from the pseudo code (Algorithm 2).

The algorithm PP, illustrated in Algorithm 2, tries to saturate all the nodes in V. Obviously, all edges are explored if all the nodes are saturated. To saturate a node r, PP uses subroutine Saturate(r). This subroutine guarantees that, when the execution finishes, node r is saturated and the agent exists on r again. During exploration, the agent constructs and keeps the *saturated tree* $S = (V_S, E_S)$, consists of all the nodes v where the agent finishes Saturate(v). The saturated tree S is maintained by variable $port_{parent}(v)$ of the agent, which stores a port number that points to v's parent in S. That is, saturated tree S is defined as follows: $V_S = \{v \in V \mid \text{Saturate}(v) \text{ finished already}\}$ and $E_S = \{p_v^{-1}(port_{parent}(v)) \mid v \in V_S\}$. When all the nodes are included in S, the exploration finishes.

All instructions of the main routine is well defined: In line 3, since node v_{cur} has been already saturated, the agent easily evaluates the condition $\exists u \in N(v_{\text{cur}}),\ u \notin V_s$ using the map H. The agent does not backtrack (Line 8) unless all neighboring nodes belong to S. Consequently, S is a spanning tree when the agent stops at node v_{st}. This means that all nodes are saturated. Hence, PP solves the exploration problem correctly.

Let us observe how Saturate(r) saturates node r. During the execution of Saturate(r), the agent keeps the *return path* $RP = (v_0, e_1, v_1, e_2, \ldots, e_k, v_k),\ v_0 = r,\ v_k = v_{\text{cur}}$ in its memory $\mathcal{M}$. The initial value of RP is (r). All the moves are performed by invoking two subroutines, GetForward(e) and GoBack(). When GetForward(e) is invoked, the agent moves through edge e, and then adds (e, v_{cur}) to the tail of RP. If this addition makes a cycle in RP, the agent deletes the cycle. More precisely, if $v_{\text{cur}} = v_i$ holds for some $i,\ 0 \le i \le k$, then the agent assigns $(v_0, e_1, v_1, e_2, \ldots, e_i, v_i)$ to RP. When GoBack() is invoked, the agent retrieves the last two elements (e_k, v_k) from RP and moves through edge e_k. The agent keeps on traversing an unexplored edge e by invoking GetForward(e) as long as unexplored edges exist in $I(v_{\text{cur}})$ (Line 15). The agent backtracks along RP by invoking GoBack() when no unexplored edge exists in $I(v_{\text{cur}})$ (Line 17). By definition, it is guaranteed that r is saturated when Saturate(r) finishes. Note that, during the execution of Saturate(r), another node v may be saturated. However, v is not included in saturated tree S at that time. Such v is included in S when Saturate(v) is executed.

We consider the number of moves required for exploration by PP. Note that the agent invokes GoBack() only when v_{cur} is saturated. Consider the case that the agent has just performed GoBack() and backtracked from a node v. Then, v is saturated, and RP does not include v. Hence, after that, the agent never visits v during the execution of any Saturate(u) for any node $u \in V$. This means that the number of invoking GoBack() is at most n in all the executions of Saturate(). Clearly, the number of invoking GetForward() is at most m. (Every invocation consumes one unexplored edge). Hence, the number of moves performed during the execution of Saturate() is at most $m + n$. And, the number of moves during the execution of main-routine is exactly $2(n-1)$(This routine makes depth-first search on the saturated tree S). Summing up these upper bounds, we see that the number of moves of PP is at most $m + 3n$.

THEOREM 1 (PANAITE AND PELC[10]).
Without whiteboards, an agent $\mathcal{A} = (PP, \mathcal{M})$ 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 whiteboard, these two algorithms simulate the move of PP with less (agent) memory space. Algorithm WPP1 uses $O(n)$ memory space on the agent and every whiteboard while algorithm WPP2 uses $O(n \log n)$ memory space on the agent and $O(\delta(v) + \log n)$ space on every whiteboard $w(v)$.

In both the two algorithms, we assume that all $id(v)$ satisfies $1 \le id(v) \le n$. This assumption does not matter since the agent can easily reassigns the labels $1, \ldots, n$ to all nodes by using $O(\log n)$ space of both the agent and each whiteboard $w(v)$. In addition, $\Omega(\delta(v))$ space of $w(v)$ also makes it possible that the agent records on $w(v)$ whether v has already been visited and whether e has already been explored or not for every $e \in I(v)$. Hence, we also assume that, at any time, the agent knows whether it already visited v_{cur} before or not and which edges in $I(v_{\text{cur}})$ are explored.

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

4.2.1 Saturate(v) with Whiteboards

The Difficulty of simulating Saturate(r) exists only in maintaining the return path RP in the subroutines GetForward() and GoBack(). Remind that the return path RP is the path from r to v_{cur}, where r is the node that has invoked Saturate(r). Other instructions can be easily performed: The agent can easily evaluate the condition $v_{\text{cur}} = r$ (Line

13) by marking the node r initially, and the other conditions are also easy to evaluate. Clearly, the simulation succeeds if the agent maintains the return path correctly.

Procedure in WPP2.

As for WPP2, the solution is easy. In WPP2, the agent can memorize the entire return path in its memory $\mathcal{M}$. The agent remembers the return path $RP = (v_0, e_1, \ldots, e_k, v_k)$ as a sequence of pairs of a node label and a port number $RP_2 = (id(v_0), q_1, \ldots, q_k, id(v_k))$. Here, $q_i = p_{v_i}(e_i)$ holds for every i, $1 \leq i \leq k$. Since the length of the return path is at most $n-1$, available agent memory space $O(n \log n)$ is sufficient to keep RP_2. Since the agent have RP_2 in its memory, the agent can maintain RP_2 in exactly the same way as the agent obeying PP maintains RP.

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

Procedure in WPP1.

In the rest of this section, we describe how WPP1 maintains the return path. Due to the lack of memory space, the agent cannot memorize the entire return path $RP = (v_0, e_1, v_1, \ldots, e_k, v_k)$ in its memory $\mathcal{M}$. However, WPP1 maintains RP by storing it separately on the agent memory and the whiteboards of all the nodes in RP. The agent only remembers the *set* of identifiers $\{id(v_j) \mid 0 \leq j \leq k\}$ on the variable RP_1 instead of the *sequence* of the identifiers. Each whiteboard $w(v_i)$ contains the port number $p_{v_i}(e_i)$ in the variables $port_{return}(v_i)$. The set RP_1 is used to detect a cycle in RP in GetForward(e), and $port_{return}(v)$ 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. This variable is used to delete a cycle when the agent detects the cycle in RP. The pseudo codes of GetForward(e) and GoBack() are given in Algorithm 3. When GetForward(e) is invoked, the agent moves through edge e, and then checks whether this move makes a cycle on the return path. This check is easily done: a cycle is created if and only if the detecting condition $id(v_{\text{cur}}) \in RP_1$ holds. If the condition does not hold, the agent extends the return path by updating RP_1 and $port_{return}(v_{\text{cur}})$ (Lines 5 and 6). At the same time, the agent stores the copy of RP_1 in $hist(v_{\text{cur}})$. If the detecting condition holds, the agent assigns $hist(v_{\text{cur}})$ to RP_1 (Line 7). Then, the return path RP equals to the path that RP_1 and $port_{return}(v_i)$ together represent. When GoBack() is invoked, the agent removes node v_{cur} ($= v_k$) from RP_1, and then moves through port $port_{return}(v_{\text{cur}})$ ($= p_{v_k}(e_k)$).

We have the following lemma. Note that the variables RP_1 and $hist(v)$ can be implemented as n-bit array.

LEMMA 2. *The subroutine* Saturate(r) *of WPP1 simulates* Saturate(r) *of PP. It uses $O(n)$ memory space of the agent and $O(n)$ memory space of $w(v)$.*

4.2.2 Main Routine with Whiteboards

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

Algorithm 3 GetForward(e) and GoBack() of WPP1

Variable in Agent

$RP_1 \in 2^{\{1,2,\ldots,n\}}$: Initially $RP_1 = \{id(r)\}$

Variables in v's Whiteboard

$port_{return}(v) \in \{1, 2, \ldots, \delta(v)\}$
$hist(v) \in 2^{\{1,2,\ldots,n\}}$: Initially $hist(r) = \{id(r)\}$

GetForward(e)

1: Move through edge e
2: **if** $id(v_{\text{cur}}) \in RP_1$ **then**
3: $RP_1 := hist(v_{\text{cur}})$ // Delete a detected cycle
4: **else**
5: $RP_1 := RP_1 \cup \{id(v_{\text{cur}})\}$ // Extend the return path
6: $port_{return}(v_{\text{cur}}) := p_{\text{in}}$
7: $hist(v_{\text{cur}}) := RP_1$ // Store the current RP_1 on $w(v)$
8: **end if**

GoBack()

9: $RP_1 := RP_1 \setminus \{v_{\text{cur}}\}$
10: Move through $port_{return}(v_{\text{cur}})$

The goal of the main routine is to incorporate all the nodes into the saturated tree S with $2n$ agent moves. The original PP achieves this goal in a simple way: If node $u \in N(v_{\text{cur}}) \setminus V_S$ exists then node u and edge $\{v_{\text{cur}}, u\}$ is selected to be incorporated into S (Lines 3-6); Otherwise, the agent backtracks to the parent node of v_{cur} in S (Line 8). However, in the execution of WPP, the agent cannot evaluate the condition $u \notin V_S$ for any $u \in N(v_{\text{cur}})$ because the agent cannot have any map of G. Therefore, another mechanism is needed to incorporate all the nodes into S with $2n$ agent moves.

Our solution is as follows. During the execution of Saturate(), the agent constructs a directed spanning tree D and stores its edges on whiteboards. In the execution of the main routine, the agent incorporates all the nodes into S with $2(n-1)$ moves by performing depth-first traversal over spanning tree D. Directed tree $D = (V_D, E_D)$ is constructed in the following way: (i) Initially, $V_D = \{v_{\text{st}}\}$ and $E_D = \emptyset$, and (ii) Every time the agent visits unexplored node v, node v_{cur} ($= v$) and directed edge $(v_{\text{cur}}, v_{\text{pre}})$ is added to V_D and E_D respectively. We say that node u is a child of node v if edge (u, v) exists in E_D, and define the *children port set* of v as $C_D(v) = \{p \mid (N(v)[p], v) \in E_D\}$. In WPP, the agent keeps $C_D(v)$ in a variable $chi(v)$ of whiteboard $w(v)$, and selects from $chi(v_{\text{cur}})$ a port to move through in Lines 3 and 4. More precisely, WPP alters Lines 3 and 4 in Algorithm 2 as follows.

3' : **if** $\exists q \in (chi(v_{\text{cur}}) \setminus port_{used}(v_{\text{cur}}))$ **then**
4-1: $port_{used}(v_{\text{cur}}) := port_{used}(v_{\text{cur}}) \cup \{q\}$
4-2: Move through port q

Here, $port_{used}(v)$ is a variable on $w(v)$, which represents the set of ports that are already used in the depth-first traversal over D. Initially $port_{used}(v) = \emptyset$ for all $v \in V$. All the other instructions (Lines 1-2 and 5-12) remain the same as the main routine of original PP. Note that the variable $port_{parent}(v)$ used in Lines 5 and 8 occupies only $O(\log \delta(v))$ space on each whiteboard $w(v)$. The agent does not remember D itself, but performs depth-first traversal over D with whiteboard variables $chi(v)$ and $port_{used}(v)$.

Algorithm 4 ConstructTree

Variables in Agent

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

Variables in v's Whiteboard

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

Before leaving via $p_{\rm out}$

1: $port_{recent}(v_{\rm cur}) := p_{\rm out}$
2: $label_{pre} := id(v_{\rm cur})$

After Arriving at $v_{\rm cur}$

3: **if** $v_{\rm cur}$ is not already visited before **then**
4: $\quad flag(label_{pre}) := 1$
5: **end if**
6: **if** $flag(id(v_{\rm cur})) = 1$ **then**
7: $\quad chi(v_{\rm cur}) := chi(v_{\rm cur}) \cup \{port_{recent}(v_{\rm cur})\}$
8: $\quad flag(id(v_{\rm cur})) := 0$
9: **end if**

To maintain the variable $chi(v)$, algorithm WPP executes a subroutine ConstructTree(), in parallel with the main routine and Saturate(). The pseudo code of ConstructTree is illustrated in Algorithm 4. This subroutine is invoked just before and just after every move of the agent. And, this subroutine does not trigger any move of the agent directly.

The idea of ConstructTree is simple. Consider the case that the agent has just moved to node v from node u, and v is unexplored before the move. Then, by raising a flag on u (agent variable $flag(u)$), the agent remembers that u has a new child (Line 4). When the agent visits u again after that, the agent knows from the flag that the current node has a new child, and then adds $p_u(\{u, v\})$ to $chi(u)$ (Line 7). To realize this addition, the agent stores port number $p_u(\{u, v\})$ on whiteboard $w(u)$ every time before it moves from u to v (Line 1). By definition, the following lemma trivially holds.

LEMMA 3. *When the agent exists on node v, $chi(v)$ equals to $C_D(v)$. That is, $chi(v_{\rm cur}) = C_D(v_{\rm cur})$ holds at any time.*

We denote D at the end of the exploration by $D_{\rm final} = (V_{D_{\rm final}}, E_{D_{\rm final}})$. Note that Line 3' is reached only after Saturate($v_{\rm cur}$) finished. Clearly, $C_D(v_{\rm cur}) = C_{D_{\rm final}}(v_{\rm cur})$ is guaranteed after Saturate($v_{\rm cur}$) has finished. Therefore, by Lemma 3, $chi(v_{\rm cur}) = C_{D_{\rm final}}(v_{\rm cur})$ holds when Line 3' is reached. Consequently, the agent performs the depth-first traversal over $D_{\rm final}$ during the main routine in WPP, which leads to the following lemma.

LEMMA 4. *$S = D_{\rm final}$ holds when the agent terminates.*

LEMMA 5. *$D_{\rm final}$ is a spanning tree. (i.e. $V_{D_{\rm final}} = V$).*

PROOF. Assume that $D_{\rm final}$ is not a spanning tree. Then, two nodes $u, v \in V$ must exist such that $u \notin V_{D_{\rm final}}$, $v \in V_{D_{\rm final}}$, and $u \in N(v)$ holds. However, by Lemma 4, edge $\{u, v\}$ is explored because $v \in V_{D_{\rm final}} = V_S$ holds. This means that node u is also explored, and thus $u \in V_{D_{\rm final}}$. This leads to a contradiction. □

By Lemmas 4 and 5, S is a spanning tree at the end of the exploration. This means that the agent explores all the edges in G. In the main routine, the agent performs depth-first search over the spanning tree $D_{\rm final}$. Hence, the number of moves during the execution of the main routine is exactly $2(n-1)$. And, by Lemmas 1 and 2, the number of moves during the execution of Saturate() is at most $m + n$. Consequently, the following two theorems hold.

THEOREM 2. *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 is $m+3n$, $O(n)$, and $O(n)$ respectively.*

THEOREM 3. *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 is $m+3n$, $O(n \log n)$, and $O(\delta(v) + \log n)$ respectively.*

5. CONCLUSION

In this paper, we proposed four exploration algorithms. By using whiteboards, they solve the exploration problem for any undirected graphs with a small number of moves and a small agent memory.

6. ACKNOWLEDGEMENTS

This work is supported in part by Grant-in-Aid for Scientific Research ((B)20300012, (B)22300009) of JSPS.

7. REFERENCES

[1] S. Albers and M.R. Henzinger. Exploring unknown environments. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 416–425. ACM New York, NY, USA, 1997.

[2] M.A. Bender, A. Fernández, D. Ron, A. Sahai, and S. Vadhan. The power of a pebble: Exploring and mapping directed graphs. *Information and Computation*, 176(1):1–21, 2002.

[3] L. Budach. Automata and labyrinths. *Math. Nachrichten*, 86:195–282, 1978.

[4] S. Das, P. Flocchini, S. Kutten, A. Nayak, and N. Santoro. Map construction of unknown graphs by multiple agents. *Theoretical Computer Science*, 385(1-3):34–48, 2007.

[5] X. Deng and C.H. Papadimitriou. Exploring an unknown graph. *Journal of Graph Theory*, 32(3):265–297, 1999.

[6] K. Diks, P. Fraigniaud, E. Kranakis, and A. Pelc. Tree exploration with little memory. *Journal of Algorithms*, 51(1):38–63, 2004.

[7] R. Fleischer and G. Trippen. Exploring an unknown graph efficiently. *Lecture Notes in Computer Science*, 3669:11–22, 2005.

[8] P. Fraigniaud and D. Ilcinkas. Digraphs exploration with little memory. *Lecture notes in computer science*, pages 246–257, 2004.

[9] L. Gasieniec, A. Pelc, T. Radzik, and X. Zhang. Tree exploration with logarithmic memory. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, page 594. Society for Industrial and Applied Mathematics, 2007.

[10] P. Panaite and A. Pelc. Exploring unknown undirected graphs. *Journal of Algorithms*, 33(2):281–295, 1999.