Vol.2018-AL-168 No.15
2018/5/26

Graph Exploration Using Constant-Size Memory and Storage*

Naoki Kitamura[1] Kazuki Kakizawa[1] Yuya Kawabata[1] Taisuke Izumi[1,a)]

Abstract: We consider the exploration problem in undirected graphs without node labels, which requires a mobile agent initially placed at an arbitrary node to visit all nodes and terminates. We assume that both of both of the agent and nodes are equipped with little memory, and the algorithm cannot use any initial knowledge on the topology of the graph. In this paper, we propose a new deterministic polynomial-time exploration (more precisely, depth-first search) algorithm which can be implemented using only $O(1)$-bit memory of the agent and $O(1)$-bit storage on each node. To the best of our knowledge, this is the first polynomial-time exploration algorithm achieving both sublogarithmic memory and storage. The technical ingredient of our algorithm consists of the idea from the recent progress on small-space DFS algorithms by Asano et al. [ISAAC2014], and Elmasry et al. [STACS2015] and a new distributed backtrack algorithm for DFS paths. The algorithm also includes a new compact (i.e., using $O(1)$-bit storage) *s-t* path maintenance mechanism, which may be of independent interest. As an application, we also show a biconnected component decomposition algorithm which runs in the asymptotically same time and space complexity as our DFS algorithm.

1. Introduction

1.1 Background and Our Result

The graph exploration problem is one of the fundamental problems in both distributed and centralized contexts, which requires an agent to visit all the nodes in the graph. It does not have only several practical applications such as mobile robots and web crawlers, but also receives much attention in relation to the complexity theory of small-space computability: The problem of deciding *s-t* connectivity is closely related to graph exploration, which is known as a key problem of capturing the hardness of space-bounded computation. In this paper, we focus on the space-complexity matter of graph exploration in distributed settings. A standard setting is the exploration of *unlabelled* graphs, where each node has no unique IDs, and its neighborhoods are identified by *local port numbers* assigned to the edges incident to each node. The agent has a limited amount of persistent memory, and does not have any knowledge on the network topology. It starts the exploration from an arbitrary node, and has to go back the initial node after visiting all other nodes. The model we consider in this paper equips each node also with persistent memory (to distinguish the agent memory, we call it the *storage* or *whiteboard*), which can be read and written by the agent visiting there. Obviously, this model has two measurements on space complexity, that is, memory size and storage size (per one node). That fact derives several natural questions: For some specific problem, can we trade one of those costs to the other one? Or is it possible to solve some problem achieving the low-space complexity in both senses? In the graph exploration problem, the $O(\log n)$-bit space restriction on memory or storage is recognized as the state-of-the-art borderline. Solving the graph exploration problem using $O(\log n)$-bit storage is rather easy. The classical DFS algorithm can be implemented on the agent-based system even if the agent is oblivious (i.e., no persistent memory). The case of $O(\log n)$-bit memory space is more complicated, but it is surprisingly possible. The seminal Reingold's undirected *s-t* connectivity algorithm achieves graph exploration using only $O(\log n)$-bit memory and no storage [1]. The optimality of this algorithm in no-storage cases is also shown by Fraigniaud et al. [2]. However, in fully-sublogarithmic cases (i.e., the memory and the storage are both sublogarithmic), the feasibility of polynomial-time graph exploration is still unclear. One of the closest results on this question is the pebble-based algorithm proposed by Diesser et al. [3], which is a graph exploration algorithm using $O(\log \log n)$ distinguished pebbles. The k-pebble model is one of the restricted versions of the memory-storage model, where each agent has k distinguished pebbles, and can put and pick-up each pebble on

[1] Nagoya Institute of Technology, Gokiso-cho, Showa-ku, Nagoya, Aichi, 466-8555, Japan
[a)] t-izumi@nitech.ac.jp

any node for leaving some information there. Since $\Omega(k)$-bit memory and storage is sufficient to implement k-pebble systems on the top of the memory-storage model, the algorithm by Diesser et al. achieves $O(\log \log n)$-bit memory and storage. Unfortunately, the running time of this algorithm can become super-polynomial of n ($O(n^{\log \log n})$ steps), and thus the existence of the algorithm achieving all of three properties — polynomial-running time, sublogarithmic memory, and sublogarithmic storage — is still an open problem.

The main contribution of this paper is to answer positively this question in very strong sense. More precisely, we present an extremely-simple deterministic polynomial-time graph exploration algorithm only using $O(1)$-bit memory and storage. While many of known algorithms are based on much complicated techniques such as *universal exploration sequences*, the approach of our algorithm is just a simulation of the depth-first search. The key ingredient of our algorithm consists of the recent progress on small-space (centralized) DFS algorithms by Asano et al. [4], and Elmasry et al. [5], as well as a new distributed backtrack algorithm for DFS paths. It includes a novel compact (i.e., using $O(1)$-bit storage) *s-t* path maintenance mechanism called *R-path*, which may be of independent interest. The total running time of our algorithm is $O(mn)$ steps. Since DFS is an important building block in many graph algorithms, the authors believe that the proposed algorithm has a vast number of applications. As an example, we show an agent-based biconnected component decomposition algorithm which runs in the asymptotically same time and space complexity as our DFS algorithm.

1.2 Related Work

The graph exploration problem has a long history, whose explicit origin goes back to the Shannon's experiment on maze-solving mouse [6]. In the field of theoretical computer science, it became in the spotlight after the seminal paper by Aleliunas et al. [7], which provides a framework based on random walks and gives an explicit polynomial-time upper bound (in expectation) on exploring all nodes in the graphs (so-called *cover time*). Combining a single $O(\log n)$-bit counter yields a simple randomized graph exploration algorithm with termination. It should be noted that achieving sublogarithmic agent memory is not trivial even using randomization, as long as we consider the task with termination. This result triggers the interest of log-space *deterministic* graph exploration algorithms, in relation to the computational complexity issue of log-space computability. The primary tool on this line is *universal exploration (or traversal) sequences*. Roughly, it is a sequence of numbers common to all possible instances in a target graph class, which guides the agent to a specific path (more precisely, a sequence of port numbers) covering all the nodes in the instance. In this approach, the small-space graph exploration is closely related to the small-space generation of universal exploration sequences, and a number of techniques along that way are developed [8, 9]. The Reingold's seminal logspace undirected connectivity algorithm is one of the milestones in this approach [1]. Yet another approach receiving much attention is the *rotor-router* model [10], which is also known as a technique of derandomizing random walks. The rotor-router model guides the agent following "local" sequences managed be each node. While our model may have the possibility of implementing that mechanism, it is not known if there exists any rotor-router mechanism applicable to any graphs and implementable using only $O(1)$-bit storage or not.

A variety of the attempts breaking the logarithmic barrier by Fraigniaud et al. [2], not relying on node storage, are presented so far: Consider a subclass of graphs such as trees [11, 12], designing port numbers [13], or adding some precomputed information (so-called *advice*) to each node in advance [14], and so on. In much strong context of distributed computing, the map construction [15–17], and collaborative exploration [12, 18] are also well studied.

1.3 Outline of the paper

The paper is organized as follows: In Section 2, we introduce the formal model and the problem definition. The algorithm is presented in an incremental manner. Section 3 introduces the R-path data structure, and then in Section 4 the main algorithm using R-path is presented. Following the explanation on the application to biconnected component decomposition in Section 5, we conclude the paper in Section 6.

2. Preliminaries

2.1 Model

Throughout this paper, we denote by $[a, b]$ the set of natural numbers at least a and at most b. The graph exploration problem is considered on a simple undirected connected graph with port numbering $G = (V, E, P)$, where V is the set of nodes, $E \subseteq V \times V$ is the set of edges, and P is the set of port-numbering functions. Let n and m be the numbers of nodes and edges respectively. In the following argument, we refer each node as a value in $[1, n]$. The reference values (IDs) are used only for introducing and analyzing our algorithms, and thus the agent cannot be aware of them. Let $E(v)$ be the set of edges incident to node $v \in V$, and $\Delta(v)$ be the degree of node $v \in V$. The port numbering function $p_v \colon E(v) \to [1, \Delta(v)]$ assigns each edge incident to v with a port number in $[1, \Delta(v)]$. The set P consists of the port-numbering functions for all nodes $v \in V$. Let $N(v)$ be the set of nodes that are adjacent to v. Since $E(v)$ and $N(v)$ has one-to-one correspondence, we often treat p_v as a function on $N(v)$. That is, for any $u \in N(v)$, $p_v(u)$ represents the v's port number of edge (v, u). Furthermore, we also denote the inverse function of $p_v(u)$ (on $N(v)$) by p_v^{-1} (i.e., $p_v^{-1} \colon [1, \Delta(v)] \to N(v)$). The port numbering is not necessarily consistent with the end-

points of one edge. That is, $p_v(u) \neq p_u(v)$ may hold for some $(u, v) \in E$. Each node $v \in V$ has an $O(1)$-bit storage called *whiteboard* $b(v)$, which is a persistent memory keeping the contents even after the agent leaves (that is, the agent can refer that information when it comes back). Note that while we incur the constant-size restriction on both the memory and whiteboards, we do not restrict the temporal memory usable in local computation at each step. That assumption is crucial because the agent must handle port numbers, which cannot be stored only with $O(1)$ bits.

The agent is a deterministic state machine moving on the nodes in G along edges. Each agent has an $O(1)$-bit memory (i.e., a constant number of states). The behavior of the agent is formally specified by a deterministic function $A\colon Q\times\mathcal{B}\times\mathbb{N} \to Q\times\mathcal{B}\times\mathbb{N}$, where Q is the set of agent states, $\mathcal{B}$ is the set of possible values written to each whiteboard. The input triple of A means the current agent state, the contents of the whiteboard where the agent stays, and the port number of the edges from which the agent comes to the current node. The triple of A's output represents the state of the agent, the value written to the whiteboard, and the port number to which the agent goes out at the next movement. If the agent does not move, A returns zero as the port number. In addition, the port number zero is also inputted to A initially. The execution of the agent follows discrete time steps. At each time step, the agent performs a local computation following A, and moves to the computed destination. It is guaranteed that the movement at time step x finishes by the beginning of time $x + 1$. The node where the agent currently stays is referred as v_{cur}, and the port number from which the agent comes to v_{cur} is referred as p_{in}. Note that the "current time" implied by notations v_{cur} and p_{in} depends on the context.

2.2 Graph Exploration Problem

In the graph exploration problem, an agent runs the algorithm from an arbitrary node $r \in V$ (say *root*). The goal of the agent is to visit all the nodes and go back to r. Without loss of generality, we assume that the agent can detect if the current location is the root or not. This assumption can be implemented by putting some special mark to the whiteboard of the root at the beginning of algorithms. Since this paper considers only deterministic algorithms, the behavior of the agent is uniquely determined by the input graph $G = (V, E, P)$, the root $r \in V$, and algorithm A. Let $S_A(G, r)$ be the sequence of the nodes that the agent visits in the execution of A for input instance G and r. An algorithm A solves the graph exploration problem if for any $G = (V, E, P)$ and $r \in V$, $S_A(G, r)$ has a finite length, has the tail node same as the head, and contains all nodes in V.

2.3 Lexicographically-Ordered DFS

The problem our algorithm solves is much stronger than graph exploration. It actually solves the lexicographically-ordered depth-first search (Lex-DFS) problem. The Lex-DFS is a special case of the standard depth-first search, which requires some specific search order following the port numbers: When the agent chooses an unvisited node in $N(v_{\mathrm{cur}})$, it must decide the node $u \in N(v_{\mathrm{cur}})$ whose port number $p_{v_{\mathrm{cur}}}(u)$ is the minimum of all unvisited neighbors. The Lex-DFS for graph $G = (V, E, P)$ and root r uniquely defines a depth-first search tree $T_{G,r}$ rooted by r. We denote the parent of v in $T_{G,r}$ by $par(v)$.

2.4 Subroutines

For ease of presentation, we introduce two fundamental subroutines used in our algorithm.

2.4.1 Procedure FindFirst

In the design of our algorithm, the agent often has to discover the neighbor with a specific state. The procedure FindFirst(O, P) probes all the neighbors $N(v_{\mathrm{cur}})$ in the order specified by O. The procedure terminates when the agent finds the neighbor $v \in N(v_{\mathrm{cur}})$ satisfying the predicate $P\colon \mathcal{B} \to$ {True,False} for the first time. Then the port p_{in} indicates the node v. Note that the value of p_{in} is modified by the run of the procedure. The value of v_{cur} does not change (i.e., the procedure necessarily finishes at the node where it is invoked). To simplify the description of the algorithm , if there are no neighbor u satisfying $P(u) =$ True, p_{in} stores -1 values [1]. We can choose the value of O from the following four options: HeadAscend, TailDescend, MiddleAscend, and MiddleDesend. The choice of Head, Tail, and Middle determines the port number where the procedure starts the search, which respectively means 1, $\Delta(v_{\mathrm{cur}})$, and p_{in}. The choice of Ascend/Descend is the order of the search, each of which corresponds to the ascending and descending orders of port numbers.

Note that iterative probing of neighbors does not need any persistent counter: In the invocation at node v, the agent repeats the go-and-back for each neighbor in $N(v)$, which is implemented by the mechanism of moving the agent coming back from v to the neighbor through port $p_{\mathrm{in}} + 1$ or $p_{\mathrm{in}} - 1$.

2.4.2 Procedure Mark

The procedure Mark(I) updates the whiteboard $b(p^{-1}_{v_{\mathrm{cur}}}(p_{\mathrm{in}}))$ following the instruction I. The actual behavior inside the procedure is that the agent first goes to $p^{-1}_{v_{\mathrm{cur}}}(p_{\mathrm{in}})$, updates the contents of $b(p^{-1}_{v_{\mathrm{cur}}}(p_{\mathrm{in}}))$ following I, and goes back to v_{cur}. Note that the value of p_{in} and v_{cur} does not change before and after the run of the procedure.

2.4.3 Note on Procedure Calls

During the execution of FindFirst or Mark, the agent has to know the argument given to the procedure call. If many types of predicates or instructions are used in the algorithm, the agent needs much memory to store the type. Yet another matter on using subroutine calls is the cost for

[1] Since the system does not have port number -1, it is actually implemented by preparing one-bit flag indicating the success or failure of the exploration in the agent state.

switching the context of programs. When the agent executes a subroutine, it also needs to remember the program counter and the state of the current context. Fortunately, the number of the predicates and instructions we use in our algorithm is bounded by a constant, and a constant number of bits suffices to store the context of the run (because we use only constant-depth nesting). Consequently, the implicit cost incurred by subroutine calls does not cause any problem, and we do not have to care about it in the following argument.

3. R-path Data Structure

In our algorithm, we utilize a novel distributed date structure called *R-path*. An instance X of *R-path* maintains a traversable path connecting an arbitrary *base node* (the root node in our use) to a *target node*. The important features of *R-path* are twofold: We can dynamically update the location of the target node, and $O(1)$-bit memory and whiteboards suffice to implement it.

3.1 Specification

An R-path X provides the following two operations to the upper application layer.

- MoveTop: The agent goes back to the base node of X.
- Modify&Move: The agent changes the target node of X to $p_{v_{\mathrm{cur}}}^{-1}(p_{\mathrm{in}})$, and moves to the new target node.

Note that these operations can be executed only when the agent is on the target node [2]. In addition, we prepare one more operation, which is used only in the implementation of the two operations above.

- MoveOneHopDown: When the agent is on the path managed by X, it moves to the neighbor in that path closer to the target node.

3.2 Compact Encoding of Path

We first explain the the structure of memory and whiteboards: The agent is oblivious in procedure MoveTop and MoveOneHopDown. In Modify&Move, it has two states, called *Find* and *Delete*. The whiteboard consists of four variables target, inPath, direction, and color. The variable target is just a flag indicating the current target node. The variables inPath and direction are also binary flags for recording the information on the maintained path (the details are explained later). The variable color is a set of marks internally used in the procedures, which takes one of four colors {white, red, blue, yellow}. Initially, all nodes have color white. We call the node satisfying inPath = True a *in-path node*, and denote by P the set of in-path nodes.

The main idea for saving space is that R-path maintains the set of in-path nodes constituting a "minimal" path from the base node to the target node, where the minimality means that the subgraph induced by P forms a path graph. This feature guarantees that any node in P has at most two neighbors in P. The flag inPath records if it belongs to the maintained path or not. To traverse the maintained path correctly, we further add one-bit information by variable direction, which indicates the upward in-path neighbor (i.e., the neighbor closer to the base node). Precisely, the value "Up > Down″ (resp. "Up < Down″) of variable direction implies that the port number of the upward neighbor is greater (resp. smaller) than that of the downward one. Supported by this information, the agent can perform upward and downward movement in the path correctly. For a node z in the maintained path, the upward and downward neighbors of z are respectively denoted by pred(z) or succ(z).

3.3 Algorithmic Ideas

For implementing MoveTop and MoveOneHopDown, it suffices to show that the agent can identify pred(v_{cur}) or succ(v_{cur}) correctly using the information of variables inPath and direction. That mechanism is implemented by a single invocation of FindFirst with an appropriate search order. For example, if $b(v_{\mathrm{cur}})$.direction = "Up < Down″ and the agent wants to find pred(v_{cur}), the agent runs FindFirst with Headascend order. Then the port returned by FindFirst is the correct way to pred(v_{cur}). All other cases can be processed similarly (see Figure 1).

The main technical challenge of realizing R-path is how to implement Modify&Move. The pseudocode of its implementation is presented in Algorithm 1. To explain its details, we consider the situation where the agent is on the target node t and wants to update the target node with a neighbor $t' \in N(t)$. First, the agent colors t' with yellow (Line 1). If t' is on the s-t path (s is the base node), the edge (t', t) is the tail edge of the current s-t path in X (Figure 2(a)). Then the update completes by simply removing it (Line 4-6). The case that t' is not on the s-t path is more complicated. Then, the agent moves to s by invoking MoveTop, and starts the *find phase* by changing its state to Find. In the find phase, the agent descends the current s-t path by using MoveOneHopDown (Figure 2(b1)). During the phase it also checks if the current in-path node has t' as its neighbor, which is done by searching the yellow neighbor with FindFirst (Line 11). Since the agent eventually reaches t, which has t' as a neighbor, the find phase always terminates with successfully finding the yellow neighbor. If the yellow neighbor is found at a node u (say *branching node*) for the first time, the concatenation of the path from s to u and the edge (u, t') creates a new path connecting s and t', which satisfies the minimality requirement of R-path. After finding the branching node u, the agent must continue to descend the s-t path to t for deleting the expired path from u to t (Figure 2(b2)). To obtain the correct downward way at node u, we do not immediately modify the variables inPath and direction at u at the end of

[2] More precisely, MoveTop can be executed when the agent is on the node contained in the path managed by the R-path structure. However we call this procedure only at the target node.

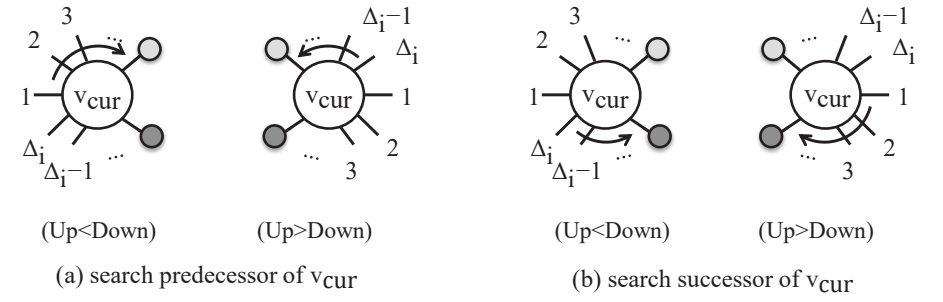

Fig. 1: Finding pred(z) or succ(z)

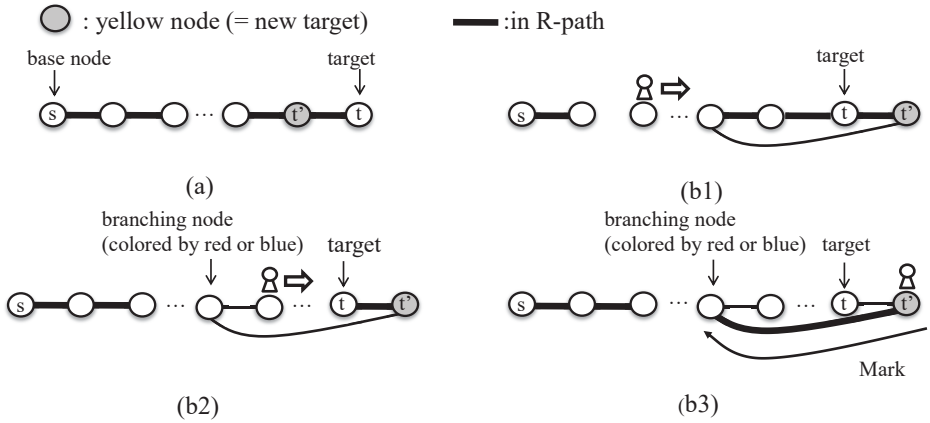

Fig. 2: The implementation of Modify&Move

the find phase. Instead, when u is found, the agent colors node u by red or blue. The color red (resp. blue) implies that the direction variable at node u must be updated by the value "Up>Down" (resp. "Up<Down"). Note that the decision of the color assigned to u can be computed by comparing the port numbers of the yellow neighbor and pred(u) via FindFirst (Line 14-21). The actual updates of $b(u)$.direction is processed at the final step of the delete phase (Line 29-33). In the remaining issue is to delete the expired u-t (sub)path. To do it, the agent starts the *delete phase* after finding a branching node, with changing its state to Delete. In the delete phase, the agent continues to descend the path with resetting inPath by False(Line 36-37). Arriving at t, the agent finds the yellow neighbor t', and thus moves to there. Then it checks the color of u (note that u is always a neighbor of t'), and updates the variables $b(u)$.inPath and $b(u)$.direction according to u's color (Line 29-33) via operation Mark(Figure 2(b3)). Since no node in the s-u path has t' as its neighbor, the established path is also minimal.

The running time of Modify&Move and MoveTop depends on the number of edges incident to in-path nodes. The computation cost at a node in the path is incurred by FindFirst. Since the number of invocations of FindFirst at each node is bounded by $O(1)$, the total number of movements are bounded by $O(m)$ (i.e., the sum of the degrees of in-path nodes).

4. Agent-Based Lex-DFS Algorithm Based on R-path

4.1 Overview

Utilizing the R-path structure, we develop a small-space Lex-DFS algorithm, called DLDFS. The fundamental structure of DLDFS follows the standard (centralized) Lex-DFS algorithm using a stack for remembering the parent node $par(v)$ in backtracking at node v. That is, when the agent visits a new node v, it marks the sign of "already visited" to the node, pushes the port number to $par(v)$ on the stack, and finds the next unvisited neighbor. If no unvisited neighbor is found, it performs the backtracking by popping the port number from the stack. Except for the part of the stack, this algorithm does not need any super-constant size memory or whiteboards. In other words, our technical challenge lies only on how we implement the stack mechanism using only $O(1)$-bit memory and whiteboards.

4.2 Implementing Abstract Stack using R-path

The implementation of the small-space stack uses the following agent/node states: In the execution of Push operation, the agent has no explicit state. For Pop operation, the agent has two states called *Find* and *Reconstruct*. Each whiteboard consists of two variables, inStack $\in$

Algorithm 1 Modify&Move(x)

```
1: Mark(color← yellow)                                   ▷ Agent marks new target node.
2: p_in ← FindFirst(HeadAscend, inPath = True∧ color = yellow)
3: if p_in ≠ −1 then                                     ▷ Agent finds yellow node on the Rpath.
4:     b(v_cur).(target, inPath) ← (False, False); Mark(target ← True; color ← white)
5:     Move to p_in
6:     Halt
7: else
8:     q ← Find; MoveTop()
9: while True do
10:    if q = Find then
11:        p_in ←FindFirst(HeadAscend, color = yellow)
12:        if p_in ≠ −1 then
13:            q ← Delete; b(v_cur).color ← red
14:            if b(v_cur).direction = "Up > Down" then
15:                p_in ←FindFirst(MiddleAscend, inPath = True)
16:                if p_in = −1 then
17:                    b(v_cur).color ← blue
18:            else
19:                p_in ←FindFirst(MiddleDescend, inPath = True)
20:                if p_in ≠ −1 then
21:                    b(v_cur).color ← blue
22:        MoveOneHopDown()
23:    if q = Delete then
24:        if b(v_cur).target = True then
25:            b(v_cur).(target, inPath) ← (False, False)
26:            p_in ← FindFirst(HeadAscend, color = yellow); Move to p_in
27:            b(v_cur).(color, inPath, target) ← (white, True, True)
28:            p_in← FindFirst(HeadAscend, color = red)
29:            if p_in = −1 then                          ▷ Branching node is blue
30:                p_in ← FindFirst(HeadAscend, color = blue)
31:                Mark(direction ← "Up < Down"; color ← white; inPath← True)
32:            else                                       ▷ Branching node is red
33:                Mark(direction ← "Up > Down"; color← white; inPath ← True)
34:            Halt
35:        else
36:            MoveOneHopDown()
37:            Mark(inPath ← False)
```

{True, False} and color ∈ {white, black, red}. Initially, all nodes are white.

Basically, the algorithm manages only the set of nodes in the stack, which is recorded by the variable inStack. The implementation of Push is very easy, that is, we only have to update $b(v_{\mathrm{cur}})$.inStack with True. The more challenging part is the implementation of Pop. Obviously, the set of nodes satisfying inStack = True (say *in-stack node*) forms a path from r to v_{cur} in $T_{G,r}$, but it does not contain the information on the order of nodes in the path. Thus we must recover the order for completing the pop operation. The recovery is obviously impossible for general stacks, but fortunately in the application to Lex-DFS, retraversing the path induced by in-stack nodes correctly recovers the path from r to v_{cur} in $T_{G,r}$. The key fact comes from the technique implicitly used in the design of small-space Lex-DFS algorithms in centralized settings, which is summarized by the following lemma.

Lemma 1 (Asano et al. [4], Elmasry et al. [5])**.** *Let $G = (V, E, P)$ and $r \in V$ be any instance of Lex-DFS, and $P_u = v_0, v_1, \cdots, v_k$ be the path from r to u in $T_{G,r}$ ($r = v_0$ and $u = v_k$). Then, for any $i \in [0, k-1]$, $p_{v_i}(v_{i+1}) = \min_{j\in[i+1,k], v_j\in N(v_i)} p_{v_i}(v_j)$. holds.*

The lemma above implies that the following process correctly traverses the path P_u: Assume that the agent is at node $v_i \in P_u$, all the nodes $v_0, v_1, \cdots, v_i$ already traversed are marked by black color, and the nodes $v_{i+1}, v_{i+2}, \cdots v_k$ are marked by white color. Then the agent can identify v_{i+1} because it is the non-black in-stack neighbor with the minimum port number. After changing the color of v_i to black, the agent moves to v_{i+1}. Iterating this step the agent completely traverses the sequence P_u. When it reaches u, p_{in} points the parent of u. To run this order-recovery mechanism, the agent must go back to the root node r for staring the re-traversal, which is realized by managing r-u path using R-path structure and calling MoveTop procedure, and adding the call of Modify&Move to the implementation of Push.

The running time of the whole algorithm is dominated by the running time of Push and Pop operations (except for the cost incurred by those operations, $O(m)$ steps suffices to finish the algorithm). Since we have shown that Modify&Move and MoveTop are both requires $O(m)$ steps, the total running time is bounded by $O(mn)$ steps. Thus

the following theorem is obtained.

Theorem 1. *There exists an agent-based Lex-DFS algorithm using a constant-space memory and whiteboards, which runs in $O(mn)$ steps.*

5. Application: Biconnected Component Decomposition

5.1 Problem Definition

We define the biconnected component decomposition (BCD) problem as a node-coloring problem. An edge $e \in E$ is called a *bridge* if removing it from G partitions G into two connected components. A subgraph $G(S)$ induced by a set of nodes $S \subset V$ is called a *biconnected component* if it is a maximal subgraph containing no bridge. By the definition, any biconnected component in a graph is mutually disjoint, and thus the vertex set of G can be partitioned into a number of biconnected components. The BCD problem is defined as the one assigning two colors (black/white) to each node such that (1) any node in a biconnected component has the same color, and (2) two endpoints of any bridge are colored differently. Since the contraction of each biconnected component into a single node always induces a tree, the coloring satisfying (1) and (2) always exists.

5.2 Algorithmic Ideas

The fundamental property holding for any bridge $e \in E$ is that the graph G does not contain a cycle containing e. It follows the observation that an edge $e \in E$ is a bridge if and only if e satisfies that (1) $e \in T_{G,r}$ holds and (2) $\forall e' = (u, v) \notin T$, the simple path connecting u and v in $T_{G,r}$ does not include e. This is the key observation of our algorithm.

In our algorithm, each node has an one-bit flag called isbridge, which is initially True at all nodes. This flag represents if the edge to the parent is a bridge or not. In the algorithm, the agent performs Lex-DFS twice, where the first time is for detecting all bridges, and the second time is for coloring. In the first run of Lex-DFS, the agent checks if each traversed edge is contained in a cycle or not. Since it is well-known that any non-tree edge appears as a forward or back edge in any DFS tree, the task of cycle detection can be processed by finding a in-stack neighbor. It is processed immediately before the pop operation. Assume that now the agent backtracks at node v. Before executing the pop operation, the agent first goes back to the root node using MoveTop and moves down to v along the path induced by in-stack nodes. Then, it also checks if each node in the path, except for $par(v)$, has v as its neighbor or not. If it is found that a node u is a neighbor of v, the in-stack (sub)path between u and v and edge (u, v) forms a cycle. Thus the agent sets isbridge to False for all the nodes in the u-v path.

At the end of the first-time Lex-DFS, it is guaranteed that the nodes with isbridge = True has a bridge as the edge to its parent, which is a sufficient information to obtain the BCD coloring: In the second-time Lex-DFS, the agent colors each node along the Lex-DFS ordering of nodes. The color given to each node is initially white, and flipped when it reaches the node with isbridge = True. It is obvious that the running time of this algorithm is asymptotically the same as our Lex-DFS algorithm. Consequently, we obtain the theorem below:

Theorem 2. *There exists an agent-based algorithm solving the BCD problem using a constant-space memory and whiteboards, which runs in $O(mn)$ steps.*

6. Conclusion and Future Directions

In this paper, we proposed a small-space Lex-DFS algorithm which consumes only a constant-size memory and storages. This is the first polynomial-time graph exploration algorithm with fully-sublogarithmic space complexity. As an application of Lex-DFS, we also presented a biconnected component decomposition algorithm running in the same time complexity as the Lex-DFS algorithm.

The authors believe that our algorithm derives several interesting open problems as follows:

- Can we obtain any fully-sublogarithmic algorithms for other problems (e.g., BFS search, computing graph properties, or any collaborative tasks by multiple agents)? Conversely, can we have any problem impossible to solve in the systems with $O(1)$-bit memory and storages?
- Is it possible to construct a faster DFS algorithm running in $o(mn)$ steps with keeping the same space usage? Or, can we have any time-complexity lower bound in this settings?
- Is there a graph exploration algorithm for oblivious agents using $O(1)$-bit storages?

Acknowledgments This research was supported by Japan Science and Technology Agency(JST) SICORP.

References

[1] Reingold, O.: Undirected connectivity in log-space, *Journal of the ACM (JACM)*, Vol. 55, p. 17 (2008).

[2] Fraigniaud, P., Ilcinkas, D., Peer, G., Pelc, A. and Peleg, D.: Graph exploration by a finite automaton, *Theoretical Computer Science*, Vol. 345, pp. 331–344 (2005).

[3] Disser, Y., Hackfeld, J. and Klimm, M.: Undirected Graph Exploration with Θ (Log Log N) Pebbles, *In Proc. of 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 25–39 (2016).

[4] Asano, T., Izumi, T., Kiyomi, M., Konagaya, M., Ono, H., Otachi, Y., Schweitzer, P., Tarui, J. and Uehara, R.: Depth-First Search Using $O(n)$ Bits., *In Proc. of 25th International Symposium on Algorithms and Computation (ISAAC)*, pp. 553–564 (2014).

[5] Elmasry, A., Hagerup, T. and Kammer, F.: Space-efficient Basic Graph Algorithms, *In Proc. of 32nd International Symposium on Theoretical Aspects of Computer Science (STACS 2015)*, Vol. 30, pp. 288–301 (2015).
[6] Shannon, C. E.: Presentation of a maze-solving machine, *8th Conference of the Josiah Macy Jr. Foundation (Cybernetics)*, pp. 173–180 (1951).
[7] Aleliunas, R., Karp, R. M., Lipton, R. J., Lovasz, L. and Rackoff, C.: Random walks, universal traversal sequences, and the complexity of maze problems, *In Proc. of SFCS '79 Proceedings of the 20th Annual Symposium on Foundations of Computer Science (FORCES)*, pp. 218–223 (1979).
[8] Koucký, M.: Universal traversal sequences with backtracking, *Journal of Computer and System Sciences*, Vol. 65, pp. 717–726 (2002).
[9] Hoory, S. and Wigderson, A.: Universal traversal sequences for expander graphs, *Information Processing Letters*, Vol. 46, pp. 67–69 (1993).
[10] Bampas, E., Gąsieniec, L., Hanusse, N., Ilcinkas, D., Klasing, R. and Kosowski, A.: Euler Tour Lock-In Problem in the Rotor-Router Model, *In Proc. of 23rd International Symposium on Distributed Computing (DISC 2009)*, pp. 423–435 (2009).
[11] Fraigniaud, P., Gasieniec, L., Kowalski, D. R. and Pelc, A.: Collective tree exploration, *Networks*, Vol. 48, pp. 166–177 (2006).
[12] Czyzowicz, J., Pelc, A. and Roy, M.: Tree Exploration by a Swarm of Mobile Agents, *In Proc. of 16th International Conference on Principles of Distributed Systems*, pp. 121–134 (2012).
[13] Dobrev, S., Jansson, J., Sadakane, K. and Sung, W.-K.: Finding Short Right-Hand-on-the-Wall Walks in Graphs, *Structural Information and Communication Complexity*, pp. 127–139 (2005).
[14] Gorain, B. and Pelc, A.: Deterministic Graph Exploration with Advice, *44th International Colloquium on Automata, Languages, and Programming (ICALP 2017)*, pp. 132:1–132:14 (2017).
[15] Panaite, P. and Pelc, A.: Exploring unknown undirected graphs, *Journal of Algorithms*, Vol. 33, pp. 281–295 (1999).
[16] Chalopin, J., Das, S. and Kosowski, A.: Constructing a Map of an Anonymous Graph: Applications of Universal Sequences, *In Proc. of 14th International Conference on Principles of Distributed Systems*, pp. 119–134 (2010).
[17] Das, S., Flocchini, P., Nayak, A., Kutten, S. and Santoro, N.: Map construction of unknown graphs by multiple agents, *Theoretical Computer Science*, Vol. 385, pp. 34–48 (2007).
[18] Das, S., Flocchini, P., Nayak, A. and Santoro, N.: Distributed Exploration of an Unknown Graph, *Structural Information and Communication Complexity*, pp. 99–114 (2005).