Exploring an Unknown Graph Efficiently*

Rudolf Fleischer[1] and Gerhard Trippen[2]

[1] Fudan University, Shanghai Key Laboratory of Intelligent Information Processing, Department of Computer Science and Engineering, Shanghai, China
fleischer@acm.org
[2] The Hong Kong University of Science and Technology, Hong Kong, Department of Computer Science
trippen@cs.ust.hk

Abstract. We study the problem of exploring an unknown, strongly connected directed graph. Starting at some node of the graph, we must visit every edge and every node at least once. The goal is to minimize the number of edge traversals. It is known that the competitive ratio of online algorithms for this problem depends on the deficiency d of the graph, which is the minimum number of edges that must be added to make the graph Eulerian. We present the first deterministic online exploration algorithm whose competitive ratio is polynomial in d (it is $O(d^8)$).

1 Introduction

Exploring an unknown environment is a fundamental problem in online robotics. Exploration means to draw a complete map of the environment. Since all decisions where to proceed with the exploration are based on local (or partial) information, the exploration problem falls naturally into the class of *online algorithms* [7,12]. The quality of an online algorithm is measured by the *competitive ratio* which is the worst-case quotient of the length of the path traveled by the online algorithm and the length of the shortest path that can visit the entire environment.

The exploration problem has been studied for various restrictions on the online algorithm (power of local sensors, fuel consumption, etc.) and many types of environment: directed graphs with a single robot [1,10,16] or with cooperating robots [4], planar graphs when we only need to visit all nodes [15], undirected graphs when we must regularly return to the starting point [2], and polygonal environments with obstacles [9] or without obstacles [14]. Similarly, the search problem has been studied for some of these environments [3,5,6,18].

In this paper, we study the online exploration problem for strongly connected directed graphs. We denote a graph by $G = (V, E)$, where V is the set of n nodes

* The work described in this paper was partially supported by the RGC/HKUST Direct Allocation Grant DAG03/04.EG05 and by a grant from the German Academic Exchange Service and the Research Grants Council of Hong Kong Joint Research Scheme (Project No. G-HK024/02-II).

G.S. Brodal and S. Leonardi (Eds.): ESA 2005, LNCS 3669, pp. 11–22, 2005.
© Springer-Verlag Berlin Heidelberg 2005

and E is the set of m edges. The *deficiency* d of the graph is is the minimum number of edges that must be added to make the graph Eulerian. A node with more outgoing than incoming edges is a *source*, and a node with more incoming than outgoing edges is a *sink*.

Initially, we only know the starting node s. We do not know n, m, d, or any part of G except the outdegree of s. We can traverse edges only from tail to head. We say an edge (node) becomes *visited* when we traverse (discover) it for the first time. At any time, we know all the visited nodes and edges, and for any visited node we know the number of unvisited edges leaving the node, but we do not know the endpoints of the unvisited edges.

The cost of an exploration tour is defined as the number of traversed edges. The Chinese Postman Problem describes the offline version of the graph exploration problem [11]. For either undirected or directed graphs the problem can be solved in polynomial time. For mixed graphs with both undirected and directed edges the problem becomes NP-complete [17]. We always need at least m edge traversals to visit all edges (and nodes) of a graph, and there exist graphs with deficiency d that require $\Omega(dm)$ edge traversals [1].

In the online problem, we may have the choice of leaving the current node on a visited or an unvisited edge. If there is no outgoing unvisited edge, we say we are *stuck* at the current node.

For undirected graphs, the online exploration problem can easily be solved by `DFS`, which is optimally 2-competitive [10]. The problem becomes more difficult for strongly connected directed graphs. Deng and Papadimitriou [10] gave an online exploration algorithm that may need $d^{O(d)}m$ edge traversals. They also gave an optimal 4-competitive algorithm for the special case $d = 1$, and they conjectured that in general there might be a $poly(d)$-competitive online exploration algorithm. For large d, i.e., $d = \Omega(n^c)$ for some $c > 0$, this is true because `DFS` never uses more than $O(\min\{nm, m + dn^2\})$ edge traversals [16] (as actually do most natural exploration algorithms [1]). The best known lower bounds are $\Omega(d^2m)$ edge traversals for deterministic and $\Omega(d^2m/\log d)$ edge traversals for randomized online algorithms [10].

Albers and Henzinger [1] gave a first improvement to the Deng and Papadimitriou algorithm. They presented the `Balance` algorithm which can explore a graph of deficiency d with $d^{O(\log d)}m$ edge traversals. They also showed that this bound is tight for their algorithm. To show the difficulty of the problem they also gave lower bounds of $2^{\Omega(d)}m$ edge traversals for several natural exploration algorithms as `Greedy`, `Depth-First`, and `Breadth-First`. For `Generalized Greedy` they gave a lower bound of $d^{\Omega(d)}m$ edge traversals.

No randomized online graph exploration algorithm has ever been analyzed, but in an earlier paper we presented an experimental study of all known deterministic and randomized algorithms [13]. The experiments basically show that on random graphs the simple greedy strategies work very well, in particular much better than the deterministic algorithms with better worst-case bounds.

In this paper we give the first $poly(d)$-competitive online graph exploration algorithm. The main idea of our algorithm is to finish chains (i.e., newly discovered

paths) in a breadth-first-search manner, and to recursively split off subproblems that can be dealt with independently because they contain new cycles which we can use to relocate without using other edges. We prove that our algorithm needs at most $O(d^8 m)$ edge traversals.

This paper is organized as follows. In Section 2, we review some of the old results which we use for our new algorithm in Section 3. We analyze the competitive ratio of the algorithm in Section 4, and we close with some remarks in Section 5.

2 Basics

If a node has no unvisited outgoing edges, we call it *finished*. Similarly, a path is *finished* if it contains no unfinished nodes. Note that we must traverse any path at least twice. The first time when we *discover* it, and then a second time to finish all nodes on the path. But some paths must be traversed more often. If we get stuck while exploring some new path or after just finishing a path, we must *relocate* to another unfinished path. Bounding these relocation costs is the main difficulty in the design of an efficient online exploration algorithm.

A graph is *Eulerian* if there exists a round trip visiting each edge exactly once. The simple greedy algorithm used for testing whether a graph is Eulerian [8, Problem 22-3] will explore any unknown Eulerian graph with at most $2m$ edge traversals. In fact, this algorithm is optimal [10]. We simply take an unvisited edge whenever possible. If we get stuck (i.e., reach a node with no outgoing unvisited edges), we consider the closed walk (it must be a cycle) just visited and retrace it, stopping at nodes that have unvisited edges, applying this algorithm recursively from each such node. This algorithm has no relocation costs (because we never get stuck). In fact, it is easy to see that no edge will be traversed more than twice. The reason for this is that the recursive explorations always take place in completely new parts of the graph. Our new algorithm for arbitrary graphs will use a similar feature.

For non-Eulerian graphs, when exploring a new path the start node will become a new source of the currently known subgraph and the node where we get stuck is either a newly discovered sink or a source of the currently known subgraph (which therefore is not a source of G). Therefore, at any time the explored subgraph has an equal number of sources and sinks (counting multiple sources and sinks with their respective multiplicity). We maintain a matching of the sources and sinks in the form of *tokens* [1]. Initially, each sink of G has a token uniquely describing the sink (for example, we could use numbers $1, \ldots, d$, because the number of tokens must equal the deficiency of G [1]). We can only get stuck when closing a cycle (as in the Eulerian case), or at a node with a token. In the former case, we proceed as in the Eulerian greedy algorithm. In the latter case, we say we *trigger* the token and we move it to the start node of the path we just followed before getting stuck (which is a new sink of the explored subgraph).

The concatenation of all paths triggering a token τ, which is a path from the current token node to its corresponding sink, is denoted by P_τ. Initially, P_τ is

an empty path. Similarly, τ_P denotes the token at the start node of a path P (if there is a token). We note that tokens are not invariably attached to a certain path. Instead, we will often rearrange tokens and their paths.

Although G is strongly connected, it is unavoidable that in the beginning our explored subgraph is not strongly connected. Albers and Henzinger described in [1, Section 3.3] a general technique that allows any exploration algorithm to assume w.l.o.g. that at any time the explored subgraph of G is strongly connected. In particular in the beginning we know an initial cycle C_0 containing s. For this assumption, we have to pay a penalty of a factor of d^2 in the competitive ratio. Basically, the algorithm must be restarted d times at a different start node, each time with a penalty factor of d for not knowing a strongly connected subgraph, before we can guarantee that our known subgraph is strongly connected. We will also make this assumption. To be more precise, we assume that in the beginning we know an initial cycle C_0 containing the start node s, and we can reach s from any token that we may discover.

3 The Algorithm

3.1 High-Level Description

We divide the graph into *clusters*. Intuitively, a cluster is a strongly connected subgraph of G that can be explored independently of the rest of G (although clusters created at different stages of the algorithm may overlap). If a cluster L was created while we were working on cluster K, then we say L is a *subcluster* of K, and L is a child of K in the *cluster tree* S. Each cluster K has an *initial cycle* C_K. The root cluster K_0 of S corresponds to the initial cycle C_0 containing the start node s. Fig. 1 shows a generic partition of a graph into recursive subclusters.

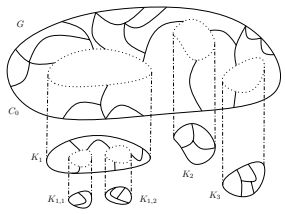

Fig. 1. A generic partition of a graph into recursive subclusters

From [1] we borrow the concept of a *token tree*. Actually, we need a token forest. For each cluster K there is a token tree T_K, with the initial cycle C_K of the cluster as root (this node does not correspond to a token). We denote the top-level token tree corresponding to K_0 by T_0. For each token there will be one node in one of the token trees of the token forest. Also, for each cluster there will be one node, called a *cluster node*, in one token tree.

When we are currently in the process of finishing a path P_τ and trigger another token π in the same token tree, then we say τ becomes the *owner* of π. The token trees represent this ownership relation: a parent token is the owner of all its child tokens. The children are always ordered from left to right in the order in which they appear on the parent token path. If π belongs to a different token tree than τ, then π will become the child of some cluster node containing τ, but the token will as usually move to v_τ, the node where we started exploring the new path that ended in π.

If π is a predecessor of τ in the token tree we cannot simply change the ownership. Instead, we create a new subcluster L whose initial cycle C_L intuitively spans the path from π to τ in the token tree. If C_L uses the initial part of a token path P_γ (i.e., γ is lying on C_L) at the time L is created as a subcluster of K (there will always be at least one such token), γ becomes an *active member* of L and its corresponding token tree node will move from T_K to T_L. In K, γ will become an *inactive member*. Also, T_K will get a new cluster node representing L containing all the new active member tokens of L. Thus, a token can be member of several clusters, but only the most recently created of these clusters will contain a node for the token in its token tree. Initially, all tokens are active members of the initial cluster K_0.

The number of tokens in a cluster node is the *weight* of this node. A cluster node is always connected to its parent by an edge of length equal to its weight.

We will maintain the invariant that any token path P_τ always consists of an unfinished subpath followed by a finished subpath, where either subpath could be empty. Since the finished subpath is not relevant for our algorithm or the analysis, we will from now on use P_τ to denote only the unfinished subpath which we call the *token path* of τ. We say a token is *finished* if its token path is empty. When we start finishing a path, called the *current chain*, we always finish it completely before doing something else, i.e., if we explore a new path and get stuck we immediately return to the current chain and continue finishing it. We say a cluster is *finished* if all nodes of its token tree are finished, otherwise it is *active*.

The clusters will be processed recursively, i.e., if a new cluster L is created while we are working in cluster K we will immediately turn to L after finishing the current chain (in K) and only resume the work in K after L has been finished. Thus, in S exactly the clusters on the rightmost path down from the root are active, all other clusters are finished.

When we start working on a new cluster K, we process its token tree T_K BFS-like. Usually, the initial cycle (i.e., the root of T_K) will be partially unfinished when we enter K, so this is usually the first chain to finish. When the next token in T_K (according to BFS) is τ and P_τ is unfinished, P_τ becomes the current chain that we are going to finish next. Then we continue with the next token to the right, or the leftmost token on the next lower level if τ was the rightmost token on its level. T_K will dynamically grow while we are working on K, but only below the level of the current chain.

We classify the edges of a cluster as native, adopted, or inherited. If a cluster K is created, it *inherits* the edges on the initial cycle C_K from its parent cluster. An edge is *native* to cluster K if it was first visited while traversing a new path that triggered a token whose token tree node was then moved to T_K (note that we can trigger tokens in a cluster K while working in some other cluster). Only active clusters can acquire new native edges, and edges are always finished while we are working in the cluster to which they are native. If a cluster is finished, all its native edges are *adopted* by its parent cluster.

A token π is a *forefather* of a token τ if there exists a cluster L such that π and τ are members of L, and either π or a cluster node containing π lies on the path from the root to τ or a cluster node containing τ. Note that an ordinary ancestor of a node in a token tree is also a forefather.

3.2 Finishing a Chain

Assume we are currently working in cluster K. To finish the current chain $C = P_\tau$ we relocate to the node currently holding the token τ and then move along C. Whenever we reach an unfinished node v we explore a new path P by repeatedly choosing arbitrary unvisited outgoing edges until we get stuck at some node w. We distinguish three cases.

(1) If $v = w$, then we cut C at v, add P to C (see Fig. 2), and continue finishing C at v, first finishing the subpath P. This is like the greedy Eulerian algorithm.

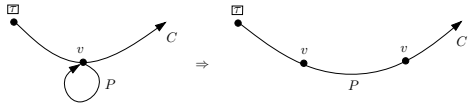

Fig. 2. Case (1): we found a loop P in C, and we extend C by P

(2) If $v \neq w$ and w holds a token π which is not a forefather of τ (if w holds several such tokens, we choose an arbitrary one), we extend the token path P_π by prepending the new path P and moving π to v (see Fig. 3).

For the update of the token trees we have to consider a few cases. Let L be the lowest ancestor (in S) of the current cluster K in which both π and τ are members. If τ is an active member of L, let z_τ be the node τ in L; otherwise, let z_τ be the cluster node in L containing τ. If π is an active member of L, then we move in T_L the node π together with its subtree as a new child below z_τ; otherwise, let z_π be the cluster node in L containing π corresponding to some subcluster L_π. We *destroy* L_π by rearranging the tokens and token paths in L_π such that π becomes the root of the token tree of L_π (instead of the initial cycle). Then we can move this new token tree as a new child below z_τ. All native edges of L_π are adopted by L.

In all cases, if z_τ is a cluster node, we also keep as additional information in node π its true ownership (in case the subcluster z_τ needs to be destroyed later, see below).

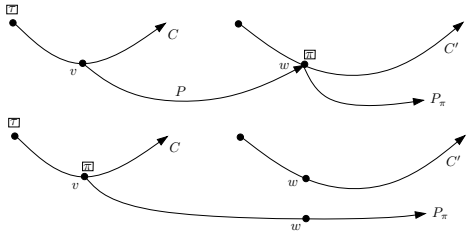

Fig. 3. Case (2): we extend the token path P_π starting in w by prepending P to P_π. The token π moves from w to v, and C becomes the new owner of π.

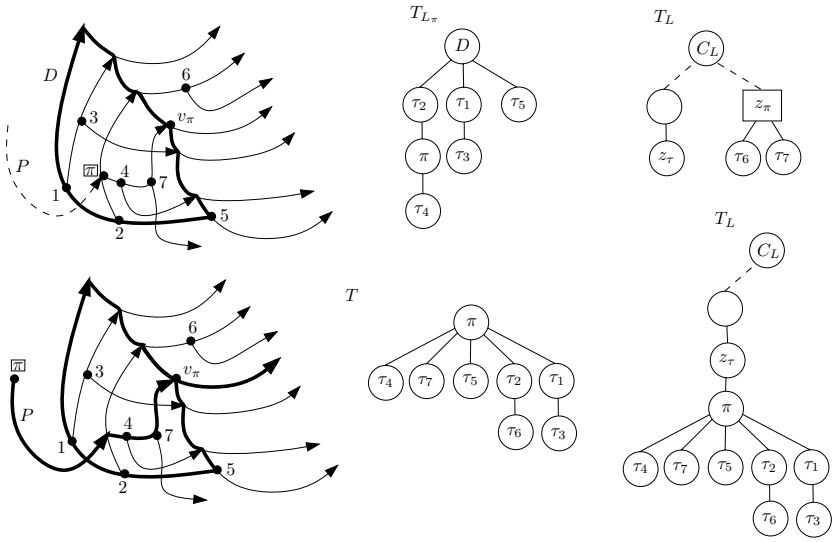

Fig. 4. How to destroy a cluster. **Top:** The numbers $1, \ldots, 7$ denote the tokens $\tau_1, \ldots, \tau_7$. Since τ_6 and τ_7 correspond to paths leading outside L_π, they are not members of L_π; instead, they are members of L and children of the cluster node z_π (which contains π) in some ancestor cluster L. A new path P from somewhere outside L_π triggers π in L_π. v_π is the position of π at the time L_π was created with initial cycle D (in bold). **Bottom:** π has moved to the start point of P. L_π was rearranged to yield a new tree T with root π. Note that this tree now contains the tokens τ_6 and τ_7. Finally, T becomes a new child of z_τ in T_L.

The crucial observation when destroying L_π is that the token path P_π was crossing the initial cycle D of L_π in some node v_π at the time L_π was created. But then we can include the cycle D into P_π and create a new token tree T for L_π with root π by cutting off the subtree rooted at π, moving all children of D as children below π, and deleting the node D. We can do this because there exists a path only using edges of L_π (and its recursive subclusters) from D to every member of L_π. If z_π happens to have children, we must also

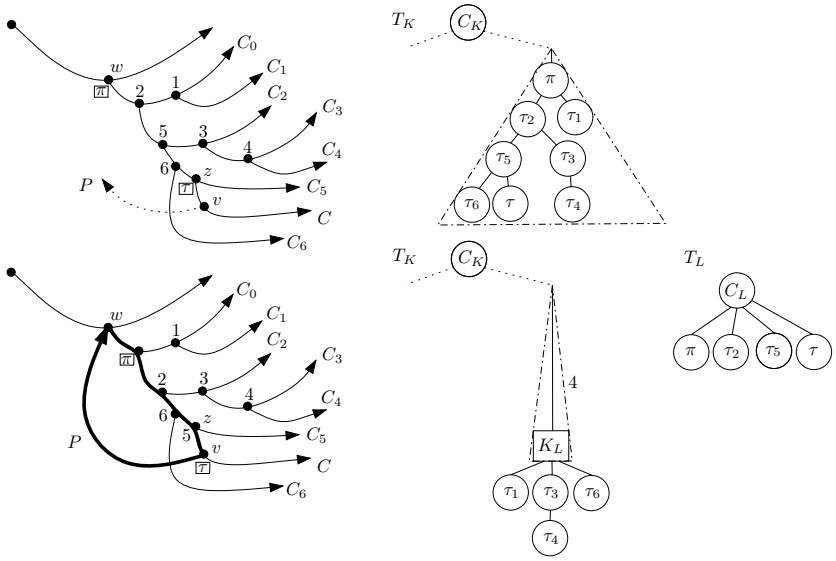

Fig. 5. Creating a cluster. **Top:** The new path P starting at v triggers π, which is an ancestor of τ in T_K. **Bottom:** We delete π and τ from T_K and create a subcluster L with initial cycle $D = (w, \tau_2, \tau_5, z, v, w)$ (in bold). The cluster node K_L in T_K has weight 4.

include these children in T by adding them as children of their owners. See Fig. 4 for an example.

Then we relocate to v and continue finishing the current chain C.

(3) If $v \neq w$ and w holds a token π which is a forefather of τ (if w holds several such tokens, we choose an arbitrary one), we cannot move the subtree rooted at π below τ. Instead, we either create a new cluster or extend the current cluster. Note that it can happen that $\pi = \tau$.

We distinguish a few cases. If π is an ancestor of τ in T_K, then we just closed a cycle D containing v and w. The part of D from w to v consists of edges that are initial parts of the token paths of all predecessors of τ up to π, see Fig. 5. We shorten all these token paths by moving the respective tokens along D to the node that originally held the next token. Thus, all these token paths now start on D. We create a new subcluster L with initial cycle $C_L = D$. All the member tokens of L become children of D. Note that these tokens lie on a path from π to τ in T_K. We cut off the subtree rooted at π and replace it by a new cluster node K_L containing the member tokens of L. The edge connecting this new node to T_K has length equal to its weight (i.e., the number of member tokens of L). Thus, it is on the same level as previously τ. If any of the member tokens of L had children in T_K that did not become member of L, we move these subtrees as children below K_L. The reason for this is that these subtrees correspond to token paths that do not end in L, so we should finish these paths within K and not within L (otherwise, relocation might become too expensive).

The description above assumed that none of the nodes in T_K between π and τ are cluster nodes. If there are cluster nodes on this path, we must first destroy these clusters (similarly as in case (2)) before we can create the new subcluster. Fig. 6 shows an example.

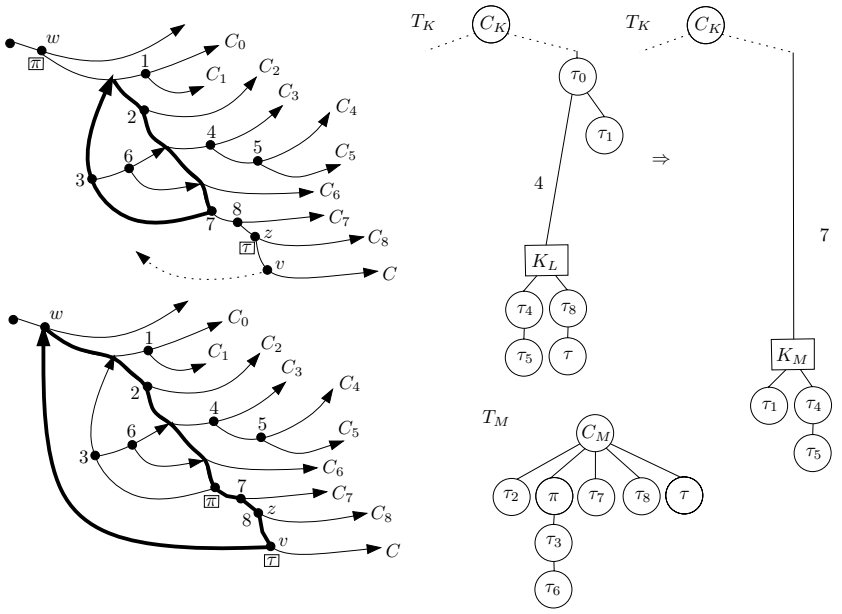

Fig. 6. Creating a cluster where the initial cycle contains a cluster node K_L of weight 4. After destroying L, we can create a new cluster M of weight 7.

Even more complicated is the case if π is contained in a cluster node in T_K, or if π is an active member of a cluster higher up in the recursion than the current cluster. In the latter case we actually extend the current cluster by including the forefather tokens up to π. Details of these constructions are left to the full version of this paper.

Before we start working recursively on the new subcluster L we must first finish the current chain C (in the current cluster K), i.e., we relocate to v. If we trigger a token that is a forefather of K_D or a token contained in K_D, we must *extend* L, see Fig. 7 for an example of the latter case. In that case, the chain that just triggered the token will be extended so that it starts on C_L, i.e., it will now contain some part of C. Note that these edges must have been recently discovered in K, i.e., originally they have been native to K, but now they become native to L instead of K. This is equivalent to what would happen if we discovered this new chain P_{τ_2} while finishing D in L.

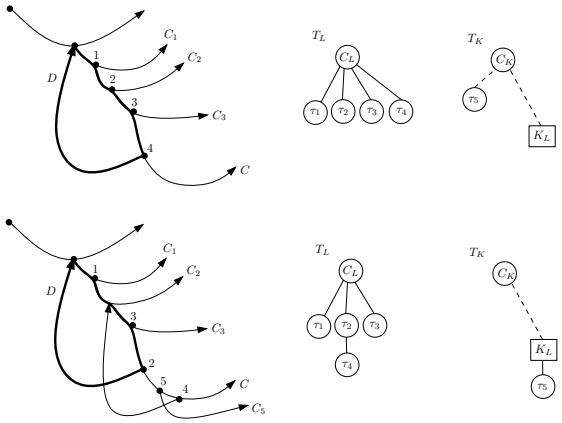

Fig. 7. Top: While finishing C (with token τ_4), we just created a subcluster L with initial chain D (in bold). **Bottom:** Before working on L we must finish C, where we trigger τ_5 and then τ_2 on D. We switch τ_2 and τ_4, making the subpath of C between the two nodes part of P_{τ_2} and its edges native to L. Then we continue finishing C which now starts at τ_4.

4 The Analysis

To analyze the total relocation cost, we count how often an edge can be used in a relocation.

Lemma 1. *At any time, any token tree T_K has at most d levels below the root, and each level can contain at most d nodes.* □

Lemma 2. *If we are currently working at a node on level h, then all nodes above level h are finished. In particular, we never move an unfinished node onto level h or above.* □

Lemma 3. *A cluster can have at most $O(d^3)$ subclusters.*

Proof. We finish at most d^2 chains by Lemmas 1 and 2. When we have finished a chain, this chain may have induced at most one subcluster. This subcluster can be extended at most $d-1$ times. □

Lemma 4. *A cluster can only inherit native or adopted edges from its parent cluster.*

Proof. Edges on the initial cycle are inherited from the parent, but never inherited by a subcluster. □

Lemma 5. *An edge can be native, inherited, or adopted in at most $O(d^3)$ clusters.*

Proof. An edge e can be native to only one cluster. e can only be adopted if its cluster is finished or destroyed. If all member tokens of a cluster K move into a subcluster L, K can never be destroyed. As soon as L is finished or destroyed, K is also finished, i.e., K adopts L's edges but it will not use them for relocations in K. Thus, each time e is adopted by an active cluster, this new cluster must have more member tokens than the previous one, i.e., e can be adopted by at most d active clusters higher up in the recursion tree. In each of these d clusters, e can be inherited by at most $O(d^2)$ subclusters because an edge can only appear in one subcluster on each of the d levels of the token tree, which then may be extended $d-1$ times. It cannot be inherited a second time by any sub-subclusters by Lemma 4. □

Lemma 6. *Not counting relocations within subclusters, at most $O(d^3)$ relocations suffice to finish a cluster.*

Proof. We finish the token tree using BFS. We have d^2 relocations after having finished a chain. While we are finishing a chain, we may trigger at most d different tokens. It may happen that the same token is triggered several times in which case it moves along the current chain. The cost of all these relocations is the same as if we had only triggered the token once, namely at the last instance. Thus, we have at most d^3 relocations due to getting stuck after exploring a new path. By Lemma 3, we must also relocate to and from $O(d^3)$ subclusters. □

Lemma 7. *Each edge is used in at most $O(d^6)$ relocations.*

Proof. This follows basically from Lemmas 5 and 6, except for those relocations where we trigger a token outside of the current cluster. In that case, we must follow some path down the cluster tree to relocate to the current chain. However, this happens only once for each token because afterwards the token is in the current cluster and we can relocate locally (and these costs are already accounted for). All the relocations then form an inorder traversal of the cluster tree, i.e., each edge in each cluster is used only a constant number of times. Since an edge can appear in $O(d^3)$ clusters by Lemma 5 and we have d tokens, each edge is used at most $O(d^4)$ times for these relocations. □

Theorem 1. *Our algorithm is $O(d^8)$-competitive.*

Proof. The assumption that we always know a strongly connected explored subgraph costs us another factor of $O(d^2)$ in the compeittive ratio [1]. □

5 Conclusions

We presented the first $poly(d)$-competitive algorithm for exploring strongly connected digraphs. The bound of $O(d^8)$ is not too good and can probably be improved. For example, the d^2-penalty for the start-up phase seems too high because some of the costs are already accounted for in other parts of our analysis. Lemma 5 seems overly pessimistic. If a cluster is extended, it does not really

incur additional costs for an edge, we could just make the cluster larger and continue as if nothing had happened. Also, destroying a cluster when a member gets triggered seems to be unnatural.

References

1. S. Albers and M. R. Henzinger. Exploring unknown environments. *SIAM Journal on Computing*, 29(4):1164–1188, 2000.
2. B. Awerbuch, M. Betke, R. Rivest, and M. Singh. Piecemeal graph exploration by a mobile robot. *Information and Computation*, 152(2):155–172, 1999.
3. E. Bar-Eli, P. Berman, A. Fiat, and P. Yan. Online navigation in a room. *Journal of Algorithms*, 17(3):319–341, 1994.
4. M. A. Bender and D. K. Slonim. The power of team exploration: Two robots can learn unlabeled directed graphs. In *Proceedings of the 35th Symposium on Foundations of Computer Science (FOCS'94)*, pages 75–85, 1994.
5. P. Berman, A. Blum, A. Fiat, H. Karloff, A. Rosén, and M. Saks. Randomized robot navigation algorithms. In *Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms (SODA'96)*, pages 75–84, 1996.
6. A. Blum, P. Raghavan, and B. Schieber. Navigating in unfamiliar geometric terrain. *SIAM Journal on Computing*, 26(1):110–137, 1997.
7. A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, Cambridge, England, 1998.
8. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, and London, England, 2. edition, 2001.
9. X. Deng, T. Kameda, and C. H. Papadimitriou. How to learn an unknown environment. *Journal of the ACM*, 45:215–245, 1998.
10. X. Deng and C. H. Papadimitriou. Exploring an unknown graph. *Journal of Graph Theory*, 32:265–297, 1999.
11. J. Edmonds and E. L. Johnson. Matching, Euler tours and the Chinese postman. *Mathematical Programming*, 5:88–124, 1973.
12. A. Fiat and G. Woeginger, editors. *Online Algorithms — The State of the Art*. Springer Lecture Notes in Computer Science 1442. Springer-Verlag, Heidelberg, 1998.
13. R. Fleischer and G. Trippen. Experimental studies of graph traversal algorithms. In *Proceedings of the 2nd International Workshop on Experimental and Efficient Algorithms (WEA'03)*. Springer Lecture Notes in Computer Science 2647, pages 120–133, 2003.
14. F. Hoffmann, C. Icking, R. Klein, and K. Kriegel. The polygon exploration problem. *SIAM Journal on Computing*, 31(2):577–600, 2001.
15. B. Kalyanasundaram and K. R. Pruhs. Constructing competitive tours from local information. *Theoretical Computer Science*, 130:125–138, 1994.
16. S. Kwek. On a simple depth-first search strategy for exploring unknown graphs. In *Proceedings of the 5th Workshop on Algorithms and Data Structures (WADS'97)*. Springer Lecture Notes in Computer Science 1272, pages 345–353, 1997.
17. C. H. Papadimitriou. On the complexity of edge traversing. *Journal of the ACM*, 23(3):544–554, 1976.
18. C. H. Papadimitriou and M. Yannakakis. Shortest paths without a map. *Theoretical Computer Science*, 84:127–150, 1991.