The Power of a Pebble: Exploring and Mapping Directed Graphs*

Michael A. Bender†
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400
bender@cs.sunysb.edu

Antonio Fernández‡
ESCET
Universidad Rey Juan Carlos
28933 Móstoles, Madrid, Spain
afernandez@acm.org

Dana Ron§
Department of EE – Systems
Tel Aviv University
Ramat Aviv, Israel
danar@eng.tau.ac.il

Amit Sahai¶
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139
amits@theory.lcs.mit.edu

Salil Vadhan‖
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139
salil@theory.lcs.mit.edu

*A preliminary version of this work appeared in *STOC '98* [9].

†This work was done while the author was at the Division of Engineering and Applied Sciences, Harvard University, and was supported by NSF grants CCR-95-04436 and CCR-93-13775.

‡Supported by the Spanish Ministry of Education, Army grant DAAH04-95-1-0607, and ARPA contract N00014-95-1-1246. This work was done while the author was at the Laboratory for Computer Science, MIT.

§This work was done while the author was at the Laboratory for Computer Science, MIT, and was supported by an NSF postdoctoral grant and by an ONR Science Scholar Fellowhip at the Bunting Institute.

¶Supported by a DOD NDSEG doctoral fellowship and partially by DARPA grant DABT63-96-C-0018.

‖This work was done while the author was supported by a DOD NDSEG graduate fellowship and partially by DARPA grant DABT63-96-C-0018.

Abstract

Exploring and mapping an unknown environment is a fundamental problem that is studied in a variety of contexts. Many results have focused on finding efficient solutions to restricted versions of the problem. In this paper, we consider a model that makes very limited assumptions about the environment and solve the mapping problem in this general setting.

We model the environment by an unknown directed graph G, and consider the problem of a robot exploring and mapping G. The edges emanating from each vertex are numbered from '1' to 'd', but we do not assume that the vertices of G are labeled. Since the robot has no way of distinguishing between vertices, it has no hope of succeeding unless it is given some means of distinguishing between vertices. For this reason we provide the robot with a "pebble" — a device that it can place on a vertex and use to identify the vertex later.

In this paper we show: (1) If the robot knows an upper bound on the number of vertices then it can learn the graph efficiently with only *one* pebble. (2) If the robot does not know an upper bound on the number of vertices n, then $\Theta(\log \log n)$ pebbles are both necessary and sufficient. In both cases our algorithms are deterministic.

1 Introduction

The problem of exploring and mapping an unknown environment is a fundamental problem with applications ranging from robot navigation to searching the World Wide Web. As such, a large body of work has focused on finding efficient solutions to variants of the problem, with restrictive assumptions on the form of the environment (*cf.* [16, 15, 22, 31, 17, 35, 10, 6, 2]). In this paper, we consider a model that makes very limited assumptions about the environment, and give efficient algorithms to solve the mapping problem in this general setting.

A natural way to model the problem is by a robot exploring a graph $\mathrm{G} = (\mathrm{V}, \mathrm{E})$. The case in which the graph has both undirected edges and labeled vertices can be solved in time linear in the number of edges by depth first search. Other search techniques [30] improve on this bound by a constant factor. Unfortunately, many exploration and mapping problems do not satisfy these constraints. For instance, if the graph represents a city (having one-way streets) or the Internet, it contains directed edges. This alone does not make the problem substantially more difficult, since the problem with directed edges and labeled vertices can be solved by a greedy search algorithm in time $O(|\mathrm{V}| \cdot |\mathrm{E}|)$. More sophisticated techniques [22, 2] yield improved running times.

Regardless of whether there are directed edges, a more daunting difficulty arises if vertices are not uniquely labeled. This situation could arise in applications from the limited sensory capabilities of a robot or from the changing appearance of vertices. If no assumptions are made on the labeling of the vertices (so that all vertices may appear the same), then we need a way to mark vertices in order to have any hope of mapping the environment [23]. In this paper, we model a marking device by a pebble, which can be dropped at a vertex and later identified and retrieved. This notion of marking is basic and can be simulated in many situations. Dudek, Jenkin, Milios, and Wilkes [23] show that a robot provided with a pebble can map an *undirected* graph with unlabeled vertices in time $O(|\mathrm{V}| \cdot |\mathrm{E}|)$, by repeatedly marking nodes and backtracking.[1] However, without the assumption that either the edges are undirected or the vertices are labeled, the existence of an efficient algorithm has remained open.

The main contribution of this paper is a general mapping algorithm which efficiently solves the mapping problem without assuming unique labelings of the vertices while allowing directed edges.

The problem. Let G be a strongly-connected directed graph over n vertices, where the vertices have no labels. The outdegree of each vertex is d, where d is assumed to be known, and the outgoing edges at each vertex are numbered from '1' to 'd'. We first observe that identical outdegrees can be assumed without loss of generality, because vertices v having outdegree smaller than d can be treated as if they have $d - \deg(v)$ additional self-loops. In fact, differences in degrees can actually help our mapping algorithms, as discussed in Section 3.5. It is a minimal assumption that the edges emanating from each vertex have labels. This is a *local* (and weak) assumption, as opposed to a *global* assumption that the vertices are labeled. Such a method for distinguishing edges is essential because otherwise it is undefined how to choose or specify a path from one vertex to another, even when provided with a map of the graph. The vertices' indegrees are not assumed to be seen, since this too can only aid the robot in distinguishing between vertices.

The robot is placed at an arbitrary starting vertex in G, and at each step it traverses one of the edges emanating from its current vertex. The robot's task is to explore and map G efficiently. That is, after walking a polynomial number of steps (in the size of the graph), it should output a graph

[1] In addition to undirected edges and labeled vertices, other simplifying assumptions that can be made about the environment include geometric structure, such as planarity, having a small diameter, and more.

$\widehat{G}$ isomorphic to G. However, as noted in [23], unless the robot has a tool to help it distinguish vertices, it is condemned to failure as a cartographer. For example, a robot traveling alone cannot decide whether G consists of a single vertex or many vertices. A basic tool for the robot is a *pebble*. Now, as the robot explores G, it can *mark* a vertex by dropping the pebble, and it can *identify* the vertex if it finds the pebble later. Upon finding the pebble, the robot can pick it up. However, because the graph is directed, the robot cannot retrace its steps to retrieve the pebble.

Bender and Slonim [10] show that a robot given a pebble can explore and map any graph in *exponential* time. However, they prove that a robot cannot map graphs in *polynomial* time using a constant number of pebbles, when it does not know a bound on n. This lower bound motivates two questions: (1) How many pebbles are needed to learn graphs efficiently if n is known? (2) How many pebbles are in fact needed if n is unknown?

In this paper we demonstrate that surprisingly few pebbles are needed in both cases. We show that

- If the robot knows n (or an upper bound $\hat{n}$ on n), it can learn the graph with only *one* pebble in time polynomial in n (respectively, $\hat{n}$).
- If the robot does not know n (or $\hat{n}$), then $\Theta(\log\log n)$ pebbles are both necessary and sufficient. Here we think of there being a *source* of pebbles that the robot has access to, and the bound is on the total number of pebbles it takes from this source in the process of exploring and mapping the graph.

In both cases our algorithms are deterministic. The lower bound of $\Omega(\log\log n)$ for the case of unknown n holds even for probabilistic algorithms.

Intuition. To understand the difficulties facing the exploring robot, consider the problem of *traversing* a graph (i.e., visiting all vertices and edges). Certainly, in order to map a graph, the robot must traverse it. One standard technique that comes to mind is *random walks*. Unfortunately, for *directed* graphs, the expected time until a random walk visits all vertices may be exponential in n and random walks are therefore ineffective for traversing. (For undirected graphs the expected time is polynomial in n.)

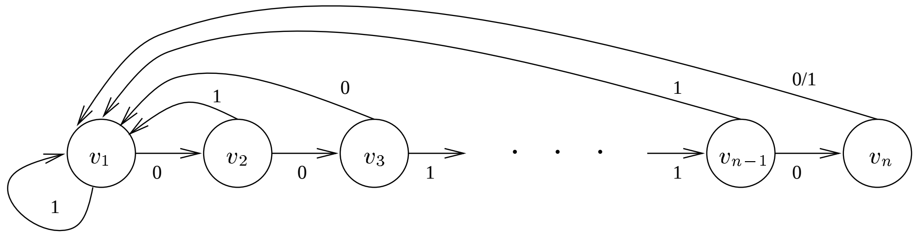

Figure 1: A combination lock graph.

Consider, for example, the graph in Figure 1. This graph is called a *combination lock graph*, because in order to reach the rightmost node v_n starting from the leftmost node v_1, the robot must discover the unique sequence of edge labels (the combination) extending from v_1 to v_n. Notice that, with very high probability, a polynomial-time random walk only visits a logarithmic number of vertices in the combination lock. More generally, for any polynomial-time (randomized) algorithm that does not mark vertices, there exists a combination lock graph that (with high probability) the algorithm will not fully explore.

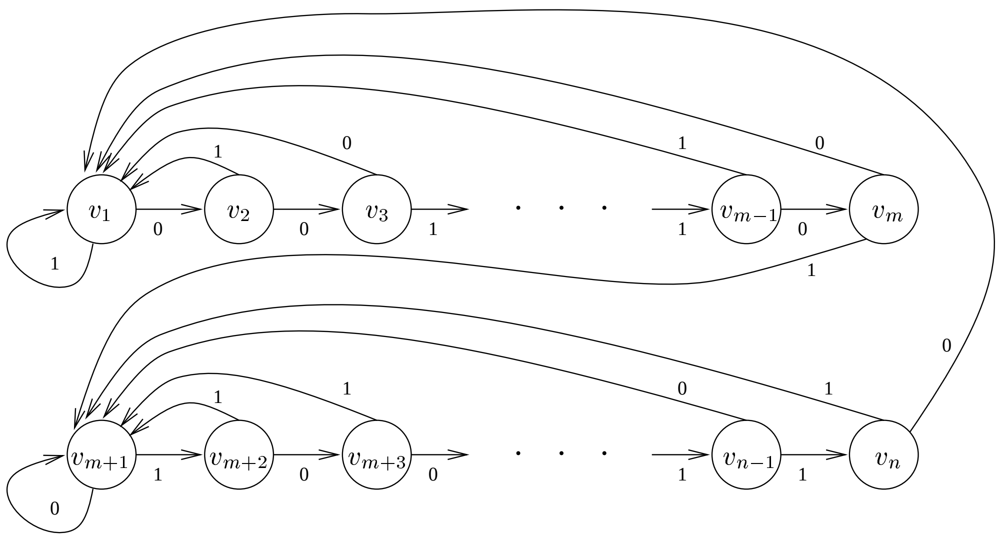

Figure 2: A graph consisting of two combination locks.

We now return to the problem of learning with a pebble. Although one (pebbleless) robot cannot traverse combination locks efficiently, a robot with a pebble can learn them using random walks [10].[2] However, consider the graph shown in Figure 2. This graph consists of two combination locks, where the end of one combination lock leads into the beginning of the other. If the robot ever drops its pebble in the top lock and travels into the bottom lock, then it is doomed. The robot will be stuck in the bottom combination lock without its pebble, and cannot even traverse this lock, much less learn it.

This example illustrates the dilemma facing the robot as it explores the unknown graph G. The robot *must* drop the pebble in order to learn new terrain, but when the robot drops the pebble, it runs the risk of *losing* it.

Closed paths. To avoid losing its pebble, the robot must know how to return to it. Thus, before dropping the pebble at a vertex, the robot should know a closed path containing this vertex. However, such a path may be difficult to obtain. When n is unknown, the robot can only identify a closed path by dropping the pebble and finding it again. Thus, we encounter a chicken-and-egg situation. In order to safely drop the pebble, the robot must find a closed path. But in order to find a closed path, the robot must drop its pebble.

Now we recognize the tangible benefit of knowing n. By repeating the same pattern of edges n times, the robot can enter a closed path *without* dropping its pebble. For example, if the robot repeatedly follows edges labeled '1', it enters a cycle after at most n moves. We refer to this as the *cycling technique*. Once the robot knows a closed path, it can map the subgraph visited by the path using the pebble. However, it is not clear how to harness this additional power. By repeating one pattern of edges, the robot enters a closed path and can map one subgraph. Later, the robot may repeat a different pattern of edges, enter another closed path, and map a second subgraph. Thus, the robot can map many subgraphs, but it is not obvious how to piece these maps together. This is because the robot has little information about how the subgraphs overlap and interconnect. As a result, finding closed paths permits the robot to drop the pebble, map a (small) portion of

[2]More generally, graphs having high *conductance* can be learned efficiently [10].

the graph and retrieve the pebble, but does not solve the mapping problem.

In order to solve the mapping problem, we use an algorithmic tool that we call an *orienting procedure*. An orienting procedure allows our algorithms to construct a limited number of maps. Instead of trying to piece these maps together, the algorithm expands them separately until one maps all of G. This expansion is possible because by executing the orienting procedure, the robot can recognize particular vertices in the graph that are associated with the maps.

Orienting procedures. Intuitively, an orienting procedure for a graph G leads the robot around the graph and ultimately leaves the robot at a vertex it "recognizes". The robot recognizes this vertex by observing the output produced by the procedure. More precisely, if the robot sees the same output when executing the procedure from two different initial vertices, then both times it ends up at the same vertex.[3] The notion of orienting procedures is analogous to the notion of (*adaptive*) *homing sequences* in automata theory [28], and it is closely related to the notion of *two-robot homing sequences* introduced by Bender and Slonim [10]. In the context of learning, homing sequences were first applied by Rivest and Schapire [35, 34]; they were used for learning environments modeled by finite automata.

We argue that every graph has a polynomial-time 1-pebble orienting procedure. (Later, we deduce from our mapping algorithm that there is actually a polynomial-time *universal* 1-pebble orienting procedure that works for all graphs of a given size.) We show that given an orienting procedure, the robot can build maps of subgraphs containing each of the possible ending vertices of the procedure. Since the robot is not provided with an orienting procedure, it builds maps using a partially-constructed orienting procedure, which it gradually improves. Each map is associated with a different output of the procedure. There is a difficulty, however, in using a partial orienting procedure. Namely, the underlying graph may look different from what the map associated with the procedure's output suggests. As a result, the robot could become disoriented and lose the pebble.

A central idea in our algorithms is how to avoid losing the pebble while using misleading information about the graph. The algorithms employ a two-tiered structure of the cycling technique mentioned above. At the lower level, the robot uses the cycling technique to verify safely whether the underlying graph is consistent with its map. If verification fails the robot is able to improve the partial orienting procedure. At the higher level, the robot uses a generalization of the cycling technique to arbitrary deterministic procedures (instead of edge-label patterns). This generalized cycling technique allows the robot to find closed paths that visit increasingly large portions of G, until all of G is visited and mapped.

Related work. The model we consider is essentially the directed-graph analogue of the one introduced by Dudek, Jenkins, Milios and Wilkes [23]. Their problem involves a robot with a single pebble mapping an *undirected* graph with unlabeled vertices. Their modeling of edge labels differs slightly from ours, in that the labeling of edges leaving a vertex can depend on the previous vertex visited (whereas our edge labelings are absolute). However, they impose an additional condition on the edge labelings which permits backtracking. Hence they are able to solve the mapping problem by repeatedly marking vertices and backtracking. Furthermore, we present an extension of our algorithm (in Subsection 3.6) that works in directed graphs when the labels of edges emanating from a vertex may depend on the previous vertex visited. Thus, we solve a problem that is strictly more general than the one treated by Dudek et al.

[3] Actually, the robot may be at vertices equivalent under automorphism, but we avoid this issue in the introduction.

Subsequent work in the model of Dudek et al. includes mapping algorithms that perform well from the perspective of competitive analysis [21], and efficient solutions to related problems such as "self-location" [24] and "map verification" [20].

Our work is very closely related to the work of Bender and Slonim [10]. Bender and Slonim show that two cooperating robots can explore and map unknown directed graphs with unlabeled vertices in polynomial time. The robots do not require any prior knowledge of the size of the graph. Bender and Slonim demonstrate that two robots are strictly more powerful than one robot with $O(1)$ pebbles; they prove that one robot with a constant number of pebbles cannot (efficiently) learn arbitrary directed graphs without knowing an upper bound on the number n of vertices. They conjecture that the same holds when n is known; our results disprove this conjecture. Our $O(\log \log n)$-pebble algorithm (for unknown n) can be simulated by two robots. This yields a deterministic alternative to Bender and Slonim's randomized two-robot algorithm.[4]

Most early work on graph exploration assumed that the robot is a finite automaton. Rabin [32] first proposed the idea of providing the automaton with pebbles to help it explore. This led to a body of work examining the number of pebbles needed to explore various environments [38, 16, 15, 5, 33]. For a survey on automata exploring labyrinths, see [29]. Deng and Papadimitriou [22] propose and study the problem of exploring an unknown directed graph having labeled vertices. Albers and Henzinger [2] give improved algorithms for this problem. These works study exploration from the perspective of competitive analysis. The results are stated in terms of the *deficiency* of the graph (i.e., the minimum number of edges to be added to make the graph Eulerian). Betke, Rivest, and Singh [12] and together with Awerbuch [6] study the problem of *piecemeal* learning undirected labeled graphs. In the piecemeal learning problem the robot is required to return to its starting position periodically.

Rivest and Schapire [35, 34] study the problem of learning environments modeled by finite automata. Here, an environment is represented by a directed graph, in which each vertex has one of two (or any constant number of) possible labelings. The robot has learned the environment (automaton) when it can predict the label of any vertex (state) reached on an arbitrary walk. Hence, if the automaton is irreducible, then the robot actually learns the topology of the underlying graph. Their algorithms (with the exception of one, for permutation automata) rely on a teacher, and build on the work of Angluin [3]. The teacher supplies counterexamples to the robot's hypotheses. Variants of this problem that do not rely on a teacher are studied in [17, 26, 36, 25]. We note that Dean et al. [17] apply a cycling technique related to ours, but for different purposes. For a survey covering some of the results mentioned above among others, see [18].

Exploring and navigating in geometric environments is studied extensively. A sample of papers includes [7, 31, 19, 14, 8, 13, 11, 27, 4].

Applications. As mentioned earlier, algorithms for exploring and mapping unknown environments have a variety of applications. Examples are obtaining maps of existing networks (e.g., computer networks, sewage systems, unexplored caves) for which there are no maps or the existing maps are outdated (e.g., after some links have gone down on a computer network). Another type of application is obtaining maps of changing environments, like the Internet or the World Wide Web. Due to the dynamic and distributed nature of these systems, it is often infeasible to maintain a completely updated map of them. However, obtaining accurate maps of small parts of the network is still useful. Another example of a changing environment comes from *ad hoc mobile wireless networks* [37]. These are networks in which the routers are mobile devices, and the topology depends

[4] In light of our results and those of Bender and Slonim, we see that a friend is only worth $\log \log n$ pebbles.

on which devices are within range of each other. If the network does not change too rapidly, a fast exploring algorithm could be used to obtain occasional snapshots of the network. We emphasize that no exact implementation of our algorithms will satisfy these applications. Even for a modest number of nodes, our algorithms are too time consuming to be immediately practical. However, the underlying ideas of our algorithms could prove useful in these applications when the nodes are not perfectly distinguishable and some of the links are unidirectional.

We also note that the problem solved in this paper is a generalization of the "twisty little passageways, all alike" problem made famous in the 1970's computer game "Colossal Cave Adventure" by Crowthers and Woods (cf., [1]).

2 Preliminaries

Let $\mathrm{G} = (\mathrm{V}, \mathrm{E})$ be the unknown directed graph the robot has to explore and map. Suppose that the graph is *strongly connected* and that all the vertices of G are unlabeled and have (the same) outdegree d. Let the edges emanating from each vertex be labeled by distinct indices in $\{1, \ldots, d\}$ and denote an edge from u to v with label σ by $\langle u, \sigma, v\rangle$. (In Section 3.6, we treat a more general model in which the edge labeling can depend on the previous vertex visited.) Let $n = |\mathrm{V}|$ and let $\hat{n}$ be an upper bound on n.

The exploring robot starts at an arbitrary vertex of G. The robot has a single pebble.[5] At each time step, the robot may traverse any outgoing edge from the vertex it is at. In addition, the robot may *drop the pebble* at the vertex or *pick up the pebble* that it has previously dropped at the vertex.

We often use the term *map* to refer to a graph $\mathrm{M} = (\mathrm{V_M}, \mathrm{E_M})$ in which each vertex has outdegree at most d and the edges leaving each vertex are labeled by distinct indices $i_1, \ldots, i_{\deg(v)} \in \{1, \ldots, d\}$. We say a map $\mathrm{M} = (\mathrm{V_M}, \mathrm{E_M})$ is *isomorphic* to G (denoted, $\mathrm{M} \cong \mathrm{G}$) if there exists an isomorphism between the two graphs that preserves edge labels. Namely, there exists a one-to-one and onto mapping $f : \mathrm{V_M} \to \mathrm{V}$, such that the following holds: For every two vertices w and z in $\mathrm{V_M}$, there is an edge labeled σ from w to z in M, if and only if there is an edge labeled σ from $f(w)$ to $f(z)$ in G. Let w_0 and v_0 be distinguished vertices in M and G, respectively. We use the notation $(\mathrm{M}, w_0) \cong (\mathrm{G}, v_0)$ to say that there exists an isomorphism f between M and G such that $f(w_0) = v_0$. We say that map (M, w_0) is *consistent* with (G, v_0) if there exists a subgraph G' of G containing v_0, such that $(\mathrm{M}, w_0) \cong (\mathrm{G}', v_0)$.

We say that the robot at vertex v in G has *learned* the graph G when it outputs a graph $\widehat{\mathrm{G}}$ together with a vertex $\hat{v}$ in $\widehat{\mathrm{G}}$ such that $(\widehat{\mathrm{G}}, \hat{v}) \cong (\mathrm{G}, v)$. Since in each time step the robot traverses a single edge, the *running time* of the algorithm is the number of moves the robot makes. Though computation time is ignored in this definition, we note that the total computation time of our algorithms is polynomial in the upper bound $\hat{n}$ on the size of the graph.

3 Learning with a Single Pebble

In this section we present our algorithm for efficiently learning any graph using a single pebble and knowledge of $\hat{n}$. We start (in Section 3.1) by describing an important subroutine of our algorithm, which we call *path compression*. The robot executes this subroutine (using the pebble) to map subgraphs of G that are visited by closed paths known to the robot. In Section 3.2 we show that

[5] In Section 4 we consider a robot having a source of pebbles.

the robot can learn G if we assume the robot has access to a *return-path oracle* for G. The robot can query this oracle from any vertex in the graph and receive a sequence of edges that leads it back to its start vertex. In the following sections we progressively weaken this assumption. In Section 3.3 we formally define an *orienting procedure* and describe how to devise such a procedure based on procedures for *distinguishing* between vertices. In Section 3.4 we replace the assumption that the robot has access to a return-path oracle with the assumption that it knows an orienting procedure for G. Finally, in Section 3.5 we show how the robot can use knowledge of $\hat{n}$ to explore and learn the graph while building an orienting procedure on its own. Our algorithm and the subroutines it uses are described in pseudocode in Figures 4, 5, 6 and 7 at the end of this section.

3.1 Compressing Closed Paths

Here we present an essential building block of our algorithms. Let the robot be at vertex v in G, and assume the robot knows a closed path in G that starts (and ends) at v. The path visits a subgraph $\mathrm{G}_{\texttt{path}}$ of G. Namely, $\mathrm{G}_{\texttt{path}}$ consists of all vertices and edges traversed along the path. Since the path may visit the same vertices several times, $\mathrm{G}_{\texttt{path}}$ is not necessarily a simple cycle. In the path compression procedure the robot uses the pebble to identify repeated vertices on the path and construct a graph M isomorphic to $\mathrm{G}_{\texttt{path}}$.

More precisely, let $\texttt{path} = \sigma_1, \ldots, \sigma_k$ be a sequence of edge labels corresponding to a closed path starting (and ending) at v. Let $u_0, u_1, \ldots, u_k$ be the vertices in G visited along the path, where $u_0 = u_k = v$. The robot maintains a list of length $k+1$ where ultimately the i-th entry in the list will identify the i-th vertex occurring on the path in G (where i ranges from 0 to k). Initially, the list is $(w_0, \Lambda, \ldots, \Lambda, w_0)$, where Λ means "unidentified." The goal of the robot is to replace all "unidentified" entries with vertex names.

The algorithm proceeds in at most n stages, each starting and ending with the robot and the pebble at v. In the 0-th stage, the robot drops the pebble at vertex v and follows the entire closed path; for each i such that the robot observes the pebble after i steps (i.e., at the vertex reached by traversing $\sigma_1, \ldots, \sigma_i$), the robot replaces the i-th entry in the list with w_0. In the j-th stage (for $j = 0, 1, \ldots$), let t be the smallest index such that the t-th entry in the list is Λ. The robot traverses $\sigma_1, \ldots, \sigma_t$, and after the t-th step drops the pebble at the vertex reached. Then it replaces the t-th entry with w_j (i.e., a new vertex name). As in the first stage, it traverses the rest of the closed path (and returns to v). For each i such that the robot observes the pebble after i steps (counting steps from when it left v), the robot replaces the i-th entry in the list (which must be a Λ) with w_j. After returning to v, the robot follows $\texttt{path}$ once more to pick up the pebble.

The algorithm maintains the property that the same label w_j appears in places k and k' in the list if and only if the k-th and k'-th vertices on the closed path in G are the same. When the list is completed, the robot constructs a map M in accordance with the list and the edge labels in $\texttt{path}$. Namely, the vertices of M are the vertices $\{w_j\}$ in the list, and if w_j and $w_{j'}$ appear in places i and $i+1$ in the list, then there is an edge $\langle w_j, \sigma_{i+1}, w_{j'} \rangle$ in M. Pseudocode for this path compression procedure is given in Figure 4.

Lemma 1 *Let v be a vertex in* G *and* $\texttt{path}$ *be a sequence of edge labels that corresponds to a closed path in* G *starting and ending at v. Let $\mathrm{G}_{\texttt{path}}$ be the subgraph of* G *visited by* $\texttt{path}$. *The path compression procedure runs in time $\mathrm{O}(n \cdot |\texttt{path}|)$ and outputs a graph* M *such that $(\mathrm{M}, w_0) \cong (\mathrm{G}_{\texttt{path}}, v)$.*

3.2 Learning with a Return-Path Oracle

In this section, we assume that the robot is given access to a *return-path oracle*. Namely, at any time step it can query the oracle and receive a sequence of edge labels that returns the robot to a particular vertex v_0.

We show how the robot can learn G by querying the oracle and using repeated applications of the path compression procedure. The return-path algorithm proceeds in at most $n \cdot d = |\mathrm{E}|$ stages. In each stage the robot learns at least one new edge in G. In the i-th stage, the robot constructs a strongly connected map M_i having a designated vertex w_0. The initial map, M_0, consists only of the vertex w_0 (and no edges). The final map is the output, $\widehat{\mathrm{G}}$, of the algorithm. The algorithm maintains the invariant that (M_i, w_0) is consistent with (G, v_0) (where consistency is defined in Section 2). The algorithm associates a closed path $\mathtt{path}(\mathrm{M}_i)$ with each map M_i. This path starts and ends at w_0 and passes through all vertices and edges in M_i. Since M_i is strongly connected, the robot can easily compute such a path of length $\mathrm{O}(n^2 d)$.

We say that a vertex w in a map M_i is *finished* if it has d outgoing edges in M_i. Otherwise it is *unfinished*. In the $(i+1)$-th stage the algorithm augments the map M_i with a new edge emanating from an unfinished vertex in M_i and perhaps other vertices and edges. This is done as follows. Let w be an unfinished vertex in M_i and let σ be the label of a missing edge from w. Let $\mathbf{explore}(\mathrm{M}_i)$ be a sequence of edge labels connecting w_0 to w, concatenated with σ. The robot performs the walk corresponding to $\mathbf{explore}(\mathrm{M}_i)$ in G starting from v_0. It then queries the return-path oracle. Let the return path that the oracle provides be called $\mathtt{ret}_i$. The robot returns to v_0 using the path $\mathtt{ret}_i$. Then it compresses the closed path $\mathtt{path}_{i+1} = \mathtt{path}(\mathrm{M}_i) \circ \mathbf{explore}(\mathrm{M}_i) \circ \mathtt{ret}_i$. The algorithm lets M_{i+1} be the resulting map. By Lemma 1, we know that $(\mathrm{M}_{i+1}, w_0) \cong (\mathrm{G}_{\mathtt{path}_{i+1}}, v_0)$. Since $\mathtt{path}_{i+1}$ contains $\mathtt{path}(\mathrm{M}_i)$, M_{i+1} contains M_i as a subgraph; by the choice of w and σ, M_{i+1} also contains at least one new edge (the edge labeled σ going out of w).

Note that the time complexity of this algorithm can be improved. However, the above formulation serves as a basis for subsequent algorithms (that do not rely on a return-path oracle). From all the above, we obtain the following lemma.

Lemma 2 *Let ℓ be the length of the longest return path provided by the oracle. The return-path algorithm runs in time $\mathrm{O}(n^2 d \cdot (n^2 d + \ell))$ and outputs a map $\widehat{\mathrm{G}}$ isomorphic to* G.

3.3 Orienting Procedures

Intuitively, an *orienting procedure* for a graph G guides the robot around the graph and ultimately leaves the robot at a vertex it "recognizes." An orienting procedure need not lead the robot back to a *particular* vertex, so assuming an orienting procedure is weaker than assuming a return-path oracle. Before we define an orienting procedure formally, we explain the notion of equivalence between vertices. We say that two vertices u and v in G are *equivalent*, denoted $u \equiv v$, if $(\mathrm{G}, u) \cong (\mathrm{G}, v)$, i.e., there exists an automorphism of G mapping u to v.

Definition 1 *An* orienting procedure **op** *for a graph* G *has the following properties.*

1. *It determines the robot's actions (i.e., what edge labels it traverses and when it drops and picks up the pebble).*
2. *The robot starts and ends with the pebble, regardless of the starting vertex.*
3. *The procedure is deterministic.*

4. *The procedure returns an* output. *The output is determined by the steps at which the robot sees the pebble.*

   (Notice that because the procedure is deterministic, every time the robot executes the orienting procedure starting from any fixed vertex v in G*, it returns the same output and finishes at the same final vertex. Thus, an orienting procedure has at most n outputs.)*

5. *Let* $\mathtt{output}(\mathbf{op}, v)$ *be the output of the procedure* **op** *when started at vertex v, and let* $\mathtt{final}(\mathbf{op}, v)$ *be the final vertex reached. An orienting procedure guarantees that for every u and v in* G $\mathtt{output}(\mathbf{op}, u) = \mathtt{output}(\mathbf{op}, v) \implies \mathtt{final}(\mathbf{op}, u) \equiv \mathtt{final}(\mathbf{op}, v)$.

   (Note that the converse is not guaranteed. Namely, the procedure may end at the same vertex with two different outputs.)

We show how to build an orienting procedure using *distinguishing procedures* for inequivalent vertices in G.

Definition 2 *Let u and v be two inequivalent vertices in* G*. A* distinguishing procedure $\mathbf{dp}_{u,v}$ *for u and v has the following properties.*

1–4. As in Definition 1.

5. $\mathtt{output}(\mathbf{dp}_{u,v}, u) \neq \mathtt{output}(\mathbf{dp}_{u,v}, v)$.

Notice that a distinguishing procedure differentiates between *starting* vertices whereas an orienting procedure differentiates between *ending* vertices. In addition, a distinguishing procedure differentiates between a single pair of starting vertices whereas an orienting procedure differentiates among all possible ending vertices.

Every orienting procedure **op** that we consider can be viewed as a tree $\mathrm{T}_{\mathbf{op}}$ in the following sense: Each leaf in $\mathrm{T}_{\mathbf{op}}$ corresponds to a different output of **op**. The internal nodes of $\mathrm{T}_{\mathbf{op}}$ are distinguishing procedures. The branches emitting from a node are labeled by the possible outputs of the distinguishing procedure. Leaves are labeled by the sequence of outputs on the branches leading from the root to the leaf. For an illustration, see Figure 3. Consider all vertices in G that the robot may end at when **op** terminates with output A at a leaf ζ_{A}; denote this set of vertices by $\mathtt{reach}(\mathrm{A})$. Property 5 dictates that all vertices in $\mathtt{reach}(\mathrm{A})$ are equivalent.

We can build an orienting procedure of the above type in stages, extending the tree in each stage. Initially we let our *candidate* orienting procedure **cop** be the empty procedure, i.e. the robot makes no actions, and the tree $\mathrm{T}_{\mathbf{cop}}$ has a single leaf. Assume inductively that **cop** preserves properties *1–4* and has k possible outputs (so that $\mathrm{T}_{\mathbf{cop}}$ has k leaves). If **cop** is not yet a complete orienting procedure, then for some output A corresponding to leaf ζ_{A} there exist inequivalent vertices u and v in $\mathtt{reach}(\mathrm{A})$. Let $\mathbf{dp}_{u,v}$ be a distinguishing procedure for u and v. We replace the leaf ζ_{A} with $\mathbf{dp}_{u,v}$. Since $\mathtt{output}(\mathbf{dp}_{u,v}, u) \neq \mathtt{output}(\mathbf{dp}_{u,v}, v)$, the new tree has at least $k+1$ leaves. Therefore, the modified **cop** has at least $k+1$ outputs. Since an orienting procedure has at most n different outputs, we obtain an orienting procedure after at most $n-1$ stages.[6] It can be shown that for every pair of inequivalent vertices there exists a distinguishing procedure with running time $\mathrm{O}(n^3 d)$. Hence, every graph has an orienting procedure with running time $\mathrm{O}(n^4 d)$. In Section 3.5, we exhibit an algorithm in which the robot devises distinguishing procedures and builds an orienting procedure while exploring the graph.[7]

[6]For the purposes of this construction, it actually suffices to relax the definition of a distinguishing procedure to allow *either* $\mathsf{output}(\mathbf{dp}_{u,v}, u) \neq \mathsf{output}(\mathbf{dp}_{u,v}, v)$ *or* $\mathsf{final}(\mathbf{dp}_{u,v}, u) \equiv \mathsf{final}(\mathbf{dp}_{u,v}, v)$.

[7]However, our algorithm may terminate (correctly) before the orienting procedure is complete.

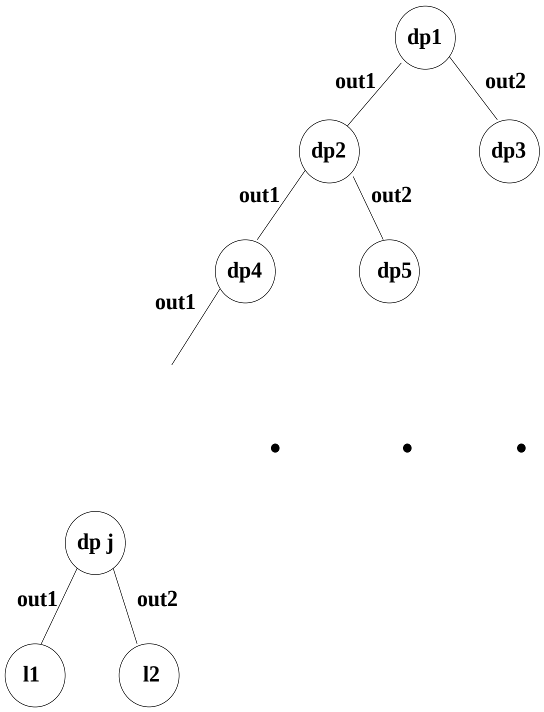

Figure 3: An illustration of $T_{\mathbf{op}}$ assuming distinguishing procedures have two possible outputs (which is not necessarily true but is the case in our usage). Each **dp** denotes a distinguishing procedure, and **out1** and **out2** are the two possible outputs. The orienting procedure begins with an execution of **dp1**. Depending on the output (**out1** or **out2**) either **dp2** or **dp3** is next executed. Each leaf corresponds to the sequence of outputs labeling the edges on the path from the root to the leaf. The leaf **l1** for example corresponds to the output **out1** ... **out1**. Since **op** is an orienting procedure, no matter where it is started, if the sequence of distinguishing procedures on the path from the root to **l1** is executed and the outputs **out1** ... **out1** are observed, then the vertices reached are equivalent.

3.4 Learning with an Orienting Procedure

In this section we assume that the robot is provided with an *orienting procedure* **op** for the graph G. For ease of presentation, we assume throughout this section that the graph has no nontrivial automorphisms (and hence no vertices are equivalent). This assumption can easily be removed here and is not used in later sections.

By the above assumption, for each possible output A, the set `reach`(A) (defined in Section 3.3) contains a single vertex, which we denote v_{A}. With each output A, the algorithm associates a map M(A), which is constructed as the algorithm proceeds. The map M(A) contains a designated vertex $w_0(\mathrm{A})$. The algorithm ensures that each M(A) is strongly connected and maintains the following invariant:

Invariant 1 (orienting procedure): *For every output* A *of* **op**, $(\mathrm{M}(\mathrm{A}), w_0(\mathrm{A}))$ *is consistent with* $(\mathrm{G}, v_{\mathrm{A}})$.

Learning proceeds in at most n^2d phases. In each phase, some map M(A) is augmented with at least one new edge. We say that a map is *finished* if all its vertices are finished (as defined in Section 3.2). The algorithm terminates when some map M(A) is finished, in which case it outputs

M(A). We use the shorthand $\texttt{path}(A)$ to represent $\texttt{path}(M(A))$ and $\textbf{explore}(A)$ to represent $\textbf{explore}(M(A))$, where $\texttt{path}(\cdot)$ and $\textbf{explore}(\cdot)$ were defined in Section 3.2. Let $G_{\texttt{path}(A)}$ be the subgraph of G visited by $\texttt{path}(A)$ when starting from v_A. In each phase the algorithm uses the orienting procedure to find a closed path satisfying the following:

1. For some output A, the path starts and ends at v_A.
2. The path visits all of $G_{\texttt{path}(A)}$ and at least one additional edge.

The robot compresses this closed path and replaces M(A) with the resulting map.

To find a closed path satisfying the above properties the robot does the following. Starting from its current vertex, it executes the orienting procedure, observes its output A_1, and follows $\texttt{path}(A_1) \circ \textbf{explore}(A_1)$. It then executes the orienting procedure again, observes its output A_2, and follows $\texttt{path}(A_2) \circ \textbf{explore}(A_2)$. The robot repeats the above until it observes an output A_j that it has previously seen (i.e., $A_j = A_i$ for some $i < j$). Note that some output must reappear after at most $n + 1$ repetitions (though the robot need not know n). At this point the robot has discovered a closed path that starts and ends at v_{A_j}. Furthermore, this closed path starts with $\texttt{path}(A_i) \circ \textbf{explore}(A_i)$, and hence visits all of $G_{\texttt{path}(A_i)}$ and at least one additional edge. Informally, since the robot does not know to which vertex it will return, it "prepares" all vertices v_{A_i} for the possibility. It does so by following $\texttt{path}(A_i) \circ \textbf{explore}(A_i)$ from each v_{A_i}.

Let $T(\textbf{op})$ be the running time of $\textbf{op}$. Since for every map M(A), $|\texttt{path}(A)| = O(n^2 d)$, and $|\textbf{explore}(A)| \le n$, the length of the closed path found is $O(n \cdot (T(\textbf{op}) + n^2 d))$. By Lemma 1, the closed path can be compressed in time $O(n^2 \cdot (T(\textbf{op}) + n^2 d))$. We obtain the following lemma.

Lemma 3 *A robot with a single pebble can learn any strongly connected graph* G *using an orienting procedure* $\textbf{op}$ *for* G *in time* $O(n^4 d \cdot (T(\textbf{op}) + n^2 d))$.

3.5 Learning the Graph while Building an Orienting Procedure

In this section we show that a robot having a single pebble can efficiently explore and map any strongly-connected directed graph if it knows an upper bound $\hat{n}$ on the size of the graph. Recall that if the robot does not know $\hat{n}$ then this task is impossible. The structure of the algorithm presented here is similar to the structure of the algorithm described in Section 3.4. Since the robot does not have a *real* orienting procedure it uses a *candidate* orienting procedure $\textbf{cop}$. In each phase, for some output A of $\textbf{cop}$ the algorithm either (1) replaces M(A) with a new, larger M(A) or (2) discovers a distinguishing procedure $\textbf{dp}_{u,v}$ for some inequivalent vertices u and v in $\texttt{reach}(A)$. In the latter case it improves $\textbf{cop}$ using $\textbf{dp}_{u,v}$ (as described in Section 3.3). Since the improved $\textbf{cop}$ will never again output A, the algorithm discards M(A). The algorithm terminates when some M(A) is finished, in which case it outputs M(A). We show that the algorithm maintains the following invariant, which is a relaxation of Invariant 1.

INVARIANT 2 (candidate orienting procedure): *For every output* A *of* $\textbf{cop}$ *there exists a vertex* $u \in \texttt{reach}(A)$ *such that* $(M(A), w_0(A))$ *is consistent with* (G, u).

In particular this invariant ensures that the finished map is isomorphic to G.

In Section 3.4 we had the property that $\texttt{reach}(A)$ consisted of a single vertex v_A. This provided a method for the robot to identify closed paths that start and end at some v_A. Here, this method does not work since $\texttt{reach}(A)$ may contain several vertices (equivalent or inequivalent). Therefore, the robot could observe output A twice *without* being on a closed path. The robot's knowledge of $\hat{n}$ combined with the following observation suggests a remedy for this problem — that is, how to find a closed path that starts and ends at a vertex u in some $\texttt{reach}(\mathrm{A})$.

Observation 1 *Let $f : \mathrm{V} \to \mathrm{V}$ be any deterministic function. Then for every vertex $v \in \mathrm{V}$, the sequence v, $f(v)$, $f(f(v))$, $\ldots$ becomes cyclic within the first n applications of f.*

Suppose the robot repeats the following: it executes **cop**, observes its output A, and follows $\texttt{path}(\mathrm{A}) \circ \textbf{explore}(\mathrm{A})$. Then after at most $\hat{n}$ repetitions it has entered a cycle. We later show how after another $2\hat{n}$ repetitions it can find a closed path that starts and ends at a vertex u in $\texttt{reach}(\mathrm{A})$, for some output A.

Suppose the robot runs the algorithm from the previous section with the enhancement above. The robot can now find closed paths, but the algorithm still has a serious flaw. Consider a map $\mathrm{M(A)}$ that results from compressing a closed path that starts and ends at $u \in \texttt{reach}(\mathrm{A})$. Assume that in a subsequent stage in the algorithm, the robot obtains a new $\mathrm{M(A)}$ by compressing a closed path that starts and ends at $u' \in \texttt{reach}(\mathrm{A})$. If $u' \equiv u$ then the argument that the new $\mathrm{M(A)}$ is larger than the old $\mathrm{M(A)}$ holds as before. However, if $u' \not\equiv u$ then we can claim nothing about the size or structure of the new $\mathrm{M(A)}$. This is because (old $\mathrm{M(A)}, w_0(\mathrm{A})$) may not be consistent with (G, u'). Hence, the argument that the new $\mathrm{M(A)}$ is bigger than the old $\mathrm{M(A)}$ is no longer valid. This motivates the need for a *map verification procedure*.

Map Verification. Suppose the robot is at a vertex v in some $\texttt{reach}(\mathrm{A})$. We would like a procedure to verify that $(\mathrm{M(A)}, w_0(\mathrm{A}))$ is consistent with (G, v). This is not difficult if we allow the robot to lose its pebble. In particular the robot hypothesizes that $\texttt{path}(\mathrm{A})$ corresponds to a closed path in G starting at v. Then the robot attempts to compress $\texttt{path}(\mathrm{A})$. If $\texttt{path}(\mathrm{A})$ is not a closed path starting from v and the robot loses the pebble, then clearly $(\mathrm{M(A)}, w_0(\mathrm{A}))$ is not consistent with (G, v). Otherwise, the robot compares $\mathrm{M(A)}$ to the map resulting from compressing the closed path.

Since we cannot allow the robot to lose the pebble (or else it will not be able to learn the graph), we must modify the above procedure. The new procedure, described below, performs a weaker form of verification. We later show that it nonetheless meets the needs of the algorithm.

1. The robot starts from v and follows $\texttt{path}(\mathrm{A})$ $\hat{n}$ times.

   Clearly, if $(\mathrm{M(A)}, w_0(\mathrm{A}))$ is consistent with (G, v), then the robot ends at v. However, even if $(\mathrm{M(A)}, w_0(\mathrm{A}))$ is not consistent with (G, v) then by Observation 1 we know that the robot has entered a cycle.

2. Next the robot drops the pebble at its current vertex v' and follows $\texttt{path}(\mathrm{A})$ once.

   - If the pebble is not at the vertex reached, then verification fails. To retrieve the pebble, the robot continues repeating $\texttt{path}(\mathrm{A})$ until it finds the pebble.
   - Otherwise, the robot compresses $\texttt{path}(\mathrm{A})$, which it has now identified as a closed path, starting from v'. If the resulting map differs from $\mathrm{M(A)}$ then verification fails. Otherwise verification passes.

We refer to this procedure as $\textbf{ver}(\mathrm{A})$. Pseudocode for $\textbf{ver}(\cdot)$ can be found in Figure 6.

Note 2 *There are two situations in which* $\textbf{ver}(\mathrm{A})$ *passes:*

1. *$(\mathrm{M}(\mathrm{A}), w_0(\mathrm{A}))$ is consistent with (G, v), or*

2. *$(\mathrm{M}(\mathrm{A}), w_0(\mathrm{A}))$ is not consistent with (G, v), but $(\mathrm{M}(\mathrm{A}), w_0(\mathrm{A}))$ is consistent with (G, v').*

If verification fails, then because of Invariant 2 **ver**(A) is a distinguishing procedure. This procedure distinguishes between v and the vertex u in `reach`(A) such that $(\mathrm{M}(\mathrm{A}), w_0(\mathrm{A}))$ is consistent with (G, u). Since for every map M(A), the length of `path`(M(A)) is $\mathrm{O}(n^2 d)$, the running time of **ver**(A) is $\mathrm{O}(\hat{n} \cdot n^2 d)$.

We note that the map verification problem is also considered in [24, 20]. However, those works involve *undirected* graphs, so the problem of losing the pebble does not arise. We are now ready to describe the final mapping algorithm.

The Algorithm. The algorithm proceeds in at most $2n^2 d$ phases. Initially, its candidate orienting procedure **cop** is the empty procedure (as described in Section 3.3). Each phase consists of at most 4 stages:

1. To enter a closed path, the robot repeats the following $\hat{n}$ times.

   $(*)$ The robot executes **cop** and obtains an output A. If this is the first appearance of output A then the algorithm creates a new map M(A) consisting of a single vertex $w_0(\mathrm{A})$. Next the robot executes **ver**(A) to verify the map M(A).

   - If **ver**(A) fails, then **ver**(A) is a distinguishing procedure between a pair of vertices in `reach`(A). The robot uses this distinguishing procedure, which outputs PASS or FAIL, to improve **cop** (as described in Section 3.3). Thus, the output of **cop** is in $\{\text{PASS}, \text{FAIL}\}^*$. Because of the extension to **cop**, **cop** will never again output A, so the robot discards M(A). The robot stops repeating $(*)$, skips Stages 2–4 (described below), and goes to the next phase with the improved **cop**.

   - Otherwise (i.e., if **ver**(A) passes), the robot follows **explore**(A). Note that by definition of **ver**(A), the robot follows **explore**(A) starting from a vertex u such that $(\mathrm{M}(\mathrm{A}), w_0(\mathrm{A}))$ is consistent with (G, u).

   The subroutine $(*)$ can be viewed as a function taking the vertex at which the robot starts to the vertex at which it finishes. By Observation 1, we know that after $\hat{n}$ repetitions of $(*)$, the robot enters a closed path consisting of some number of executions of $(*)$.

2. The aim of this stage is to determine the closed path the robot has entered.[8] To determine this closed path, the robot repeats $(*)$ another $2\hat{n}$ times. For $i = 1, \ldots, 2\hat{n}$, let A_i be the output observed in the i'th repetition of $(*)$ and let L_i be the sequence of edge labels traversed. The robot finds the smallest p such that the sequence of pairs $(\mathrm{A}_1, \mathrm{L}_1), \ldots, (\mathrm{A}_{2\hat{n}}, \mathrm{L}_{2\hat{n}})$ consists entirely of periodic repetitions of its last p entries. More precisely, for all i, $(\mathrm{A}_{2\hat{n}-i}, \mathrm{L}_{2\hat{n}-i}) = (\mathrm{A}_{2\hat{n}-(i \bmod p)}, \mathrm{L}_{2\hat{n}-(i \bmod p)})$. Let `seq` $= (\mathrm{L}_{2\hat{n}-p+1}, \ldots, \mathrm{L}_{2\hat{n}})$ be the sequence of edge labels in these last p entries. By the minimality of p, the closed path consists of one or more repetitions of `seq`. To determine the closed path, the robot drops the pebble and repeatedly traverses `seq` until it finds the pebble at the end of one of its traversals of `seq`. It then retrieves the pebble for future use.

[8] Note that the robot cannot simply drop the pebble and repeat $(*)$ until it sees the pebble again because the robot needs the pebble to execute $(*)$.

3. The robot proceeds along the closed path found above until it reaches the end of any execution of **cop**, say with output A. The robot then compresses the closed path and replaces M(A) with the resulting map.
4. If the new M(A) is finished then the algorithm outputs (the new) M(A) and terminates.

Pseudocode for this algorithm and subroutines used by the robot are provided in Figures 4, 5, 6 and 7. We now proceed to analyze the algorithm. As noted above, if **ver** ever fails in Stage 1, the robot can improve **cop**. If all verifications pass, by Lemma 1 we know that in each phase (new $\mathrm{M(A)}, w_0(\mathrm{A})$) is consistent with (G, u) for some $u \in$ `reach`(A), and thus Invariant 2 is preserved. Because **ver**(A) is part of the closed path and by Note 2, the new M(A) contains the old M(A) as a subgraph. Because **explore**(A) is part of the closed path (and is followed from u) the new M(A) also contains at least one new edge.

The algorithm terminates after at most $2n^2d$ phases because in each phase the algorithm can either improve the candidate orienting procedure or enlarge a map. More precisely, since the candidate orienting procedure can be improved at most $n-1$ times, at most $n-1$ maps are discarded. At any time the algorithm maintains at most n maps, and so the algorithm builds at most $2n-1$ maps. Since each map contains at most $n{\cdot}d$ edges, the bound on the number of phases follows. Note that the algorithm may terminate before completing the orienting procedure.

The running time of each phase is the sum of (1) the time to find a closed path, and (2) the running time of the compression procedure. Item (1) is $\mathrm{O}(\hat{n})$ times the sum of (a) the running time of the candidate orienting procedure, (b) the running time of the verification procedure, and (c) the length of the exploration sequence (which is at most n). Recall that the running time of the verification procedure is $\mathrm{O}(\hat{n}n^2d)$. Also recall that verification procedures (that fail) are distinguishing procedures for improving the candidate orienting procedure. Therefore, we can bound the running time of any candidate orienting procedure by $n \cdot \mathrm{O}(\hat{n}n^2d) = \mathrm{O}(\hat{n}n^3d)$. Thus, Item (1) amounts to $\hat{n} \cdot \mathrm{O}(\hat{n}n^3d) = \mathrm{O}(\hat{n}^2n^3d)$. By Lemma 1, Item (2) is bounded by $n \cdot \mathrm{O}(\hat{n}^2n^3d) = \mathrm{O}(\hat{n}^2n^4d)$. Since there are at most $2n^2d$ phases, we obtain the following Theorem.

Theorem 1 *A robot having a single pebble can learn any strongly connected graph given an upper bound $\hat{n}$ on the size of the graph in time $\mathrm{O}(\hat{n}^2n^6d^2)$.*

Note that the fact that the running time is stated as a function of n (and not only $\hat{n}$) does not contradict the fact that the algorithm does not know n. The algorithm terminates when it has a complete map, and only the analysis ensures the time bound as a function of n (as well as $\hat{n}$ and d).

We observe that although our mapping algorithm may terminate before the orienting procedure it devises is completed, the algorithm as a whole can be viewed as an orienting procedure that outputs a completed map and a designated vertex. Thus, we have:

Corollary 4 *There is a universal one-pebble orienting procedure that works for all graphs of out-degree d with at most $\hat{n}$ vertices and runs in time* $\mathrm{poly}(\hat{n}, d)$.

Using additional knowledge. As noted in the introduction, we have tried to make as few assumptions on the graph as possible. In particular, we have not assumed that the vertices are labeled in any way, while we have assumed the outdegrees of all vertices are the same, and that the indegrees are not observed. In case any additional distinguishing information is provided, the robot can use it to its benefit. For example, suppose the outdegrees of the vertices vary, where the outdegree of each vertex can be obtained at the vertex. Then this information can be incorporated into the orienting procedure. In particular, when there is no distinguishing information, then the output of the procedure is determined only by the step(s) in the procedure in which the pebble (which was previously dropped) is observed. If some vertices have different outdegrees than others, then the output of the orienting procedure can be determined also by the degrees of the vertices observed during its execution.

compress$(\sigma_1, \ldots, \sigma_k)$
/* $\sigma_1, \ldots, \sigma_k$ corresponds to a closed path from the current vertex. This procedure outputs a map of the subgraph corresponding to the edges traversed by this path. */

1. for $i = 0, \ldots, k$ do: $List[i] \leftarrow \Lambda$.
2. $j \leftarrow 0$.
3. while $\exists i$ s.t. $List[i] = \Lambda$ do
   (a) $t \leftarrow \min\{0 \leq i \leq k : List[i] = \Lambda\}$.
   (b) traverse $\sigma_1, \ldots, \sigma_t$.
   (c) drop pebble.
   (d) $List[t] \leftarrow w_j$.
   (e) for $i = t+1, \ldots, k$ do
      i. traverse σ_i;
      ii. if pebble found then $List[i] \leftarrow w_j$.
   (f) follow $\sigma_1, \ldots, \sigma_k$ and pick up the pebble on the way.
   (g) $j \leftarrow j + 1$.
4. return map defined by $List$ and $\sigma_1, \ldots, \sigma_k$ (where w_0 is distinguished).

Figure 4: Subroutine **compress**.

explore(M,w_0)
/* M is a (strongly connected) map, w_0 a distinguished vertex in M. This procedure (deterministically) traverses an edge that is unmapped in M. */

1. traverse a sequence of edge labels that induces a path in M from w_0 to some unfinished vertex w (i.e., w has outdegree smaller than d in M). (It is easy to deterministically find such a path of length $\leq n$.)
2. traverse an edge label corresponding to an unmapped edge from w in M.

Figure 5: Subroutine **explore**.

3.6 An Extension to Relative Edge Labels

The graph model treated in the previous sections captures a mapping problem for a very general class of environments. However, it does assume that the labels on the edges incident to a vertex are fixed. Although mapping would be impossible without some level of consistency in the labeling of

ver(M,w_0)
/* M is a (strongly connected) map, w_0 a distinguished vertex in M. This procedure verifies if the robot eventually reaches (or is currently at) a subgraph isomorphic to (M, w_0). */

1. let `path` be a sequence of edge labels that induces a closed path starting and ending at w_0 traversing all edges in M. (This can be found using the straightforward deterministic $O(n^2 d)$ algorithm that simply concatenates paths to and from all edges in M.)
2. follow `path` $\hat{n}$ times.
3. drop pebble.
4. follow `path` once.
5. if pebble found at vertex reached then
   (a) pick up pebble.
   (b) $(\mathrm{M}', w_0') \leftarrow$ **compress**(`path`).
   (c) if (M', w_0') is isomorphic to (M, w_0) then return pass.
   (d) else return fail.
6. else
   (a) repeatedly follow `path` until pebble is found, and pick up pebble.
   (b) return fail.

Figure 6: Subroutine **ver**.

edges, we can consider a relaxed model in which the local labeling of edges leading out of a vertex can be a function of the previous vertex in the robot's path. In this section, we sketch how our algorithm can be adapted to this setting as well.

The new model. A map M consists of a set of vertices V, and for each vertex v, a set of at most dn triples (u, σ, w). Such a triple indicates the existence of an edge leading from v to w, whose label is σ when v is entered using an edge from u. (So w is determined by v, u, and σ.) For ease of presentation, we assume that for every v, there are either 0 or d triples of the form $(u, \cdot, \cdot)$ for each possible u, but, as in the original model, allowing the outdegree to be a function of u and v only makes the problem easier. This model is now a strict generalization of the model of Dudek et al. [23], who impose an additional condition on the graph and edge labelings that enables backtracking.[9]

For example, in an environment modeling a city, the vertices might correspond to intersections and the edge labels might be "turn left", "turn right", and "continue straight." Clearly, the vertex to which one of these labels leads depends on the direction from which the current vertex was entered.

[9]Dudek et al. describe their model as allowing the labeling of edges leaving a vertex to depend on the *edge* from which the vertex is entered. However, they allow at most one edge between every two vertices, and hence the dependence on the edge entered translates to a dependence on the previous vertex visited. We allow multiple edges and hence make the dependence on the previous vertex.

Algorithm Explore-and-Map

/* Map graph given one pebble and an upper bound $\hat{n}$ on number of nodes. */

1. **cop** $\leftarrow$ empty procedure.
2. set of maps $\leftarrow$ empty.
3. while no map is completed do
   (a) `update-cop` $\leftarrow$ false.
   (b) repeat $\hat{n}$ times or until `update-cop` = true:
      i. execute **cop** and let A be the output observed.
      ii. if no map corresponds to output A then create new map M(A) with single vertex w_0(A).
      iii. if **ver**(M(A), w_0(A)) = pass then **explore**(M(A), w_0(A)).
      iv. else
         A. use **ver**(M(A), w_0(A)) to improve **cop** by replacing leaf of $\mathrm{T}_{\mathbf{cop}}$ that corresponds to A with internal node corresponding to **ver**(M(A), w_0(A)).
         B. remove M(A) from set of maps.
         C. `update-cop` $\leftarrow$ true.
   (c) if `update-cop` = false
      i. for $j = 1, \ldots, 2\hat{n}$ do /* since entered cycle in Step 3b, will not need to create new maps and the verifications below always pass */
         A. execute **cop** and let A_j be the output observed.
         B. **ver**(M(A_j), w_0(A_j)).
         C. **explore**(M(A_j), w_0(A_j)).
         D. Let L_j be the sequence of edge labels traversed in the above steps A–C.
      ii. find smallest p such that for all i, $(\mathrm{A}_{2\hat{n}-i}, \mathrm{L}_{2\hat{n}-i}) = (\mathrm{A}_{2\hat{n}-(i \bmod p)}, \mathrm{L}_{2\hat{n}-(i \bmod p)})$.
      iii. let `seq` = $(\mathrm{L}_{2\hat{n}-p+1}, \ldots, \mathrm{L}_{2\hat{n}})$.
      iv. drop pebble and repeat traversing (all of) `seq` until pebble found and retrieved. let `path` = $\sigma_1, \ldots, \sigma_k$ be the closed path found.
      v. proceed along `path` until reach end of subsequence of edges corresponding to an execution of **cop**. let the output corresponding to this execution be A, and let the last edge taken be σ_i.
      vi. replace (M(A), w_0(A)) with **compress**$(\sigma_{i+1} \ldots \sigma_k, \sigma_1 \ldots \sigma_i)$.
4. output completed map.

Figure 7: The algorithm

The new algorithm. We define a function f taking maps M in our new model to maps $f(\mathrm{M})$ in our previous model, where edge labels are unique. There is a vertex in $f(\mathrm{M})$ corresponding to each pair of vertices (u, v) connected by some edge in M. Then, for each triple of the form (u, σ, w) associated with vertex v in M, there is an edge labeled σ from (u, v) to (v, w) in $f(\mathrm{M})$. Clearly, f is efficiently computable and injective. Let G denote the complete map of the unknown graph; then $f(\mathrm{G})$ has exactly dn nodes. Our objective now will be to use the algorithm presented in the previous section to learn $f(\mathrm{G})$, since $f(\mathrm{G})$ is in our previous model. However, a direct application our mapping algorithm would require dropping the pebble on vertices of $f(\mathrm{G})$, whereas the robot is only allowed to drop the pebble on vertices of G. Below, we sketch how, with slight modifications, our mapping algorithm can be implemented even with this restriction.

We first observe that the **compress** procedure, if given a sequence of edge labels that induces a closed path in $f(\mathrm{G})$, can be implemented precisely as before. Referring to Figure 4, we see that *List* and the sequence of edge labels $\sigma_1, \ldots, \sigma_k$ completely determine a map M such that $f(\mathrm{M})$ is strongly connected. We modify the procedure only slightly, so that instead of returning a single vertex w_0, it returns the pair $(List[k-1], w_0)$ as the distinguished vertex of $f(\mathrm{M})$.

Now, every path the robot takes in G induces a path in $f(\mathrm{G})$. Since $f(\mathrm{G})$ has at most dn nodes, we obtain the following adaptation of Observation 1 to this setting:

Observation 3 *Let* **p** *be any deterministic procedure for the robot. Let $p(u, v)$ be the pair of vertices (u', v') such that if the robot begins at node v having entered from node u, then applying* **p** *leads it to vertex v', entering from u'. Then for every vertex $(u, v) \in f(\mathrm{G})$, the sequence (u, v), $p(u, v)$, $p(p(u, v))$, $\ldots$ becomes cyclic within the first dn applications of* **p**.

Thus, we redefine $\hat{n}$ to be d multiplied by our upper bound on the number of vertices. Now, by Observation 3, we can be sure that after $\hat{n}$ applications of any deterministic procedure, the robot will enter a cycle not only in G, but in $f(\mathrm{G})$, as well.

The only difficulty that remains in using our original algorithm to map $f(\mathrm{G})$ is that if the robot drops its pebble, follows some path, and finds the pebble, we *cannot conclude the robot has found a cycle in* $f(\mathrm{G})$ (even though it has found a cycle in G). In order to do this, it must check that some *pair* (u, v) occurs again after following the path. There are two places in the original algorithm where this might be a problem: once in the **ver** procedure, and once in the main algorithm. We discuss the remedy for each case now.

In the **ver** procedure, given in Figure 6, on input a map M and distinguished vertex (a, b) in $f(\mathrm{M})$, the robot follows a particular sequence of edge labels called `path` $\hat{n}$ times. (With our new definition of $\hat{n}$, we know the robot is in a cycle in $f(\mathrm{G})$ after this.) Now, the robot must first check to see if (a single execution of) `path` indeed specifies a cycle in $f(\mathrm{G})$ from its current location. We now describe a procedure to do this. The procedure assumes that there exists some $m \leq \hat{n}$ such that $\mathtt{path}^m$ is a cycle in $f(\mathrm{G})$ from the current location of the robot (where $\mathtt{path}^m$ denotes `path` concatenated with itself m times); this is indeed the because the robot has just executed `path` $\hat{n}$ times.

check(`path`): The robot drops its pebble, and does the following: For $i = 1$ to $\hat{n}$, the robot traverses `path` once, and checks to see if the pebble is found. If so, it continues the for-loop. If not, then `path` certainly does not define a cycle in $f(\mathrm{G})$, and so the robot traverses `path` repeatedly until the pebble is found (which is guaranteed since $\mathtt{path}^m$ was a cycle from the robot's starting point in $f(\mathrm{G})$). It picks up the pebble, and returns FAIL. If this for-loop ends with the robot always finding the pebble after each traversal of `path`, then by Observation 3, we know that repeated traversals of `path` induce a cycle (u_1, v_1), $(u_2, v_2), \ldots,$ (u_k, v_k) in $f(\mathrm{G})$. However, since the robot always sees the pebble after each traversal of `path`, this implies $v_1 = v_2 = \cdots = v_k = v$ for some vertex v. To confirm that `path` itself induces a cycle in $f(\mathrm{G})$, we need only test that $u_i = u_{i+1}$ for some i. Note that if `path` takes (u_i, v) back to $(u_i, v) = (u_{i+1}, v)$ for some i, by our definition of $(u_1, v_1), \ldots, (u_k, v_k)$, this implies that $u_i = u_{i+1} = \cdots = u_k = u_1 = u_2 = \cdots = u_i$, and hence `path` by itself induces a cycle in $f(\mathrm{G})$. In order to test that $u_i = u_{i+1}$, the robot picks up the pebble, and takes all but one step of `path`, and drops the pebble. The robot must now be at vertex u_i for some i. It then takes the last step of `path`, and again traverses all but the last step of `path`. The robot must now be at vertex u_{i+1}. If the pebble is not there, then `path` does not define a cycle in $f(\mathrm{G})$, so the robot takes the last step of `path`, and repeatedly traverses `path` until the pebble is found along the way. It picks up the pebble and completes the traversal of `path`, and then returns FAIL. If the pebble is found, then the robot has confirmed that following `path` takes it from some vertex (u, v) back to (u, v) in $f(\mathrm{G})$, and hence defines a closed path in $f(\mathrm{G})$. The robot retrieves the pebble, takes the last step of `path`, and returns PASS. Note that during this **check** procedure, the robot's path is always $\mathtt{path}^j$ for some integer j.

We replace Steps 3–6 of **ver** with the following: The robot executes **check**(`path`). If the check fails, the verification fails. If the check passes, then the robot calls **compress** using `path`, which returns M' and (a', b'). It then checks to see if $(f(\mathrm{M}), (a, b))$ is isomorphic to $(f(\mathrm{M}'), (a', b'))$. If so, the verification procedure returns PASS, otherwise FAIL. With these changes, the new verification procedure satisfies the conditions of Note 2 (with M(A) replaced by $f(\mathrm{M(A)})$ and G replaced by $f(\mathrm{G})$); these are precisely the properties the mapping algorithm requires from the verification procedure.

In the main procedure, given in Figure 7, the situation is a little more complicated. Here, if `update-cop` is false, we find a sequence `seq` of edge labels such that we know some number of repetitions of `seq` induces a cycle in $f(\mathrm{G})$, but we must figure out how many in order to have a valid input to supply to **compress** later. Similar to above, we must modify Step 3.c.iv in order to determine a closed path. Now, we know that at this point, the robot is in a cycle in $f(\mathrm{G})$ defined by some number of repetitions of `seq` between 1 and $\hat{n}$. We simply check each of these possibilities one by one. For $i = 1$ to $\hat{n}$, the robot executes **check**($\mathtt{seq}^i$). Whenever the check first succeeds, the robot knows that $\mathtt{seq}^i$ is a closed path in $f(\mathrm{G})$ starting at its current vertex. Thus, we let $\mathtt{path} = \mathtt{seq}^i$, exit the for-loop, and continue with the rest of the algorithm as before.

We can see by inspection that these are the only times in the algorithm where the pebble is employed, and that the above changes satisfy the requirements of the algorithm. Hence, this algorithm allows the robot to learn a map of $f(\mathrm{G})$ in polynomial time. This map of $f(\mathrm{G})$ can be easily transformed into a map of G (in the new model).

4 Learning without an Upper Bound on n

In this section we prove our results concerning the number of pebbles needed to learn graphs efficiently if the graph size is unknown. We use the algorithm of Section 3.5 as a subroutine to show that for any $c > 0$, $\lceil c \log\log n \rceil$ pebbles are sufficient. The resulting algorithm is deterministic. In addition, we prove a matching lower bound demonstrating that $\Omega(\log\log n)$ pebbles are necessary. The lower bound applies to any randomized algorithm that uses an expected polynomial number of moves. We note that in our upper bound the total computation time to decide on moves is polynomial, whereas the lower bound applies even when the robot is computationally unbounded.

Furthermore, our upper bound holds even when the pebbles used by the robot are indistinguishable from each other, while the lower bound holds for distinguishable pebbles.

We want to study how the number of pebbles needed grows with the size of the unknown graph. We denote the *expected number of pebbles* a (probabilistic) robot A uses on graphs of size n, by $p_{\mathrm{A}}(n)$. Namely,

$$p_{\mathrm{A}}(n) \stackrel{\text{def}}{=} \max_{\mathrm{G} \in \mathcal{G}_n} \mathrm{E}[\# \text{ of pebbles that A uses on G}],$$

where $\mathcal{G}_n$ is the set of all graphs on n vertices. The *expected running time* of A is defined analogously. (Recall that in each time step the robot makes a single move, and hence the running time of the algorithm is the number of moves the robot makes.)

Theorem 2 *For every constant $c > 0$, there exists a (deterministic) algorithm that learns graphs of size n in polynomial-time using at most $\lceil c \log \log n \rceil$ pebbles, without knowledge of n.*

Theorem 3 *Consider any algorithm* A *that, with probability greater than* $1/2$*, learns any graph in expected polynomial time without knowing the size of the graph. Then $p_{\mathrm{A}}(n) = \Omega(\log \log n)$.*[10]

Throughout the following proofs, all logarithms are have a base 2.

Proof (of Theorem 2): We use the algorithm of Section 3.5 combined with a variant of the standard guess-and-double technique; instead of doubling, the algorithm takes the k'th power for a suitably chosen k. To be precise, let $k = \lceil 2^{1/c} \rceil$, let **onepeb**$(\hat{n})$ be the one-pebble learning algorithm of Section 3.5 which takes a bound $\hat{n}$ on the number of vertices as input, and suppose $q(\hat{n})$ is a polynomial bound on its running time. Assume first that the pebbles used by the robot are distinguishable. The new algorithm works as follows on a graph of outdegree d: Guess that the number of vertices in the graph is $n_1 = 2^k$, and run **onepeb**(n_1) for $q(n_1)$ steps using the first pebble. If the algorithm outputs a finished map, i.e., every vertex has d edges coming out of it, then output this graph and halt. On the other hand, if the algorithm fails to produce a finished map or the robot loses the pebble during the execution of the algorithm, then the entire process is repeated using $n_2 = n_1^k = 2^{k^2}$ instead of n_1 and using pebble 2. (If pebble 1 is seen during this execution, it is ignored.) If the execution with n_2 fails, we continue with $n_3 = n_2^k = 2^{k^3}$. We repeat like this, using $n_\ell = n_{\ell-1}^k = 2^{k^\ell}$ at the ℓ'th stage until some execution is successful.

It is easy to see that if the algorithm **onepeb** ever outputs a finished graph, the output is correct, even if the number of vertices given to **onepeb** is incorrect. Alternatively, we can simply add an extra map verification procedure as in Section 3.5 to the end of **onepeb** to guarantee that the output is always either correct or FAIL. Moreover, by Theorem 1, the algorithm **onepeb** is guaranteed to give a correct output within time $q(\hat{n})$ as long as it is given a bound $\hat{n}$ larger than the number of vertices in the graph. Thus, given a graph of n vertices, the algorithm above will always succeed by stage ℓ, where ℓ is the first integer such that $2^{k^\ell} \geq n$, *i.e.* $\ell = \lceil (\log \log n)/(\log k) \rceil \leq \lceil c \log \log n \rceil$. Since $n_\ell = n_{\ell-1}^k \leq n^k$, the running time of this algorithm is at most $\ell q(n^k) \leq n q(n^k)$, which is polynomial in n. Lastly, the algorithm uses at most $\ell \leq \lceil c \log \log n \rceil$ pebbles.

To deal with indistinguishable pebbles, we add the following modification. Whenever the algorithm **onepeb** assumes the robot is in a cycle and is about to drop its pebble, we have the robot walk once around the cycle, picking up all pebbles that are there before proceeding. Consider stage ℓ of the (parent) algorithm, where ℓ is the first integer such that $2^{k^\ell} \geq n$. Then we are guaranteed

[10] It is easy to see from the proof that the success probability of $1/2$ is arbitrary and can be replaced by any constant.

(by the properties of algorithm **onepeb**), that the robot is in fact in a cycle whenever it is about to drop its pebble. Therefore, if it always picks up all pebbles left on the cycle before dropping its current pebble, then it will not mistake its pebble with previously dropped pebbles, and will consequently succeed in learning the graph. To ensure that the parent algorithm does not halt prematurely and output an incorrect graph (in a stage ℓ such that $2^{k^\ell} < n$), we do the following. Before halting and outputting a graph, we have the robot walk around its entire supposed view of the graph collecting all pebbles it sees. If the number of pebbles it finds is the same as the number of pebbles it has ever dropped (and not picked up), then it runs the map verification procedure and halts if it passes. Otherwise, it continues to the next stage. ∎

We note that the algorithm given in the above proof can be deterministically simulated by two (synchronized or communicating) robots. The second robot can play the role of the pebble; whenever the first robot does not find the second robot within the appropriate number of steps (due to an underestimate for n), the second robot can "catch up" to the first robot by following the first robot's (deterministic) steps and then they can proceed with a larger guess for n. This gives a deterministic alternative to Bender and Slonim's randomized two-robot mapping algorithm [10].

Proof (of Theorem 3): In order to prove the theorem, we analyze the behavior of any algorithm on two types of graphs of outdegree 2: *cycles* and *combination locks with tails*. Formally, the *cycle* of n nodes is the labeled, directed graph C_n on vertex set $\{w_0, \cdots, w_{n-1}\}$, where there are two directed edges labeled 0 and 1 going from w_i to $w_{(i+1) \bmod n}$. A combination lock with tail has the following structure (see Figure 8). Let $\alpha = \alpha_1\alpha_2\cdots\alpha_\ell \in \{0,1\}^\ell$ be any string and let $m \geq 0$ be an integer. The combination lock with combination α and tail m is the graph $L_{\alpha,m}$ on vertex set $\{u_1, u_2, \ldots, u_m, v_1, \ldots, v_{\ell+1}\}$ with the following edges: For each $1 \leq i \leq m-1$, there are two edges labeled 0 and 1 from u_i to u_{i+1}; there are two edges labeled 0 and 1 from u_m to v_1; for each $1 \leq i \leq \ell$, there is an edge labeled α_i from v_i to v_{i+1} and an edge labeled $\bar{\alpha}_i$ from v_i to v_1; there are two edges labeled 0 and 1 from $v_{\ell+1}$ to u_1. It is important to note that a robot starting at vertex v_1 (i.e., the start of the combination lock) does not reach vertex v_{k+1} unless it executes the consecutive sequence of moves $\alpha_1\cdots\alpha_k$ at some point. We start by giving the intuition behind the proof.

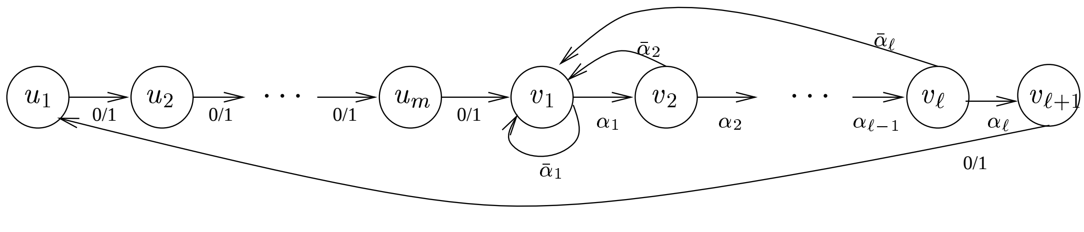

Figure 8: A combination lock with a tail.

We analyze any algorithm based on the times it drops pebbles in the case that it does not see previously-dropped pebbles. We show that there must be huge gaps in these pebble-dropping times or else the algorithm uses $\Omega(\log\log n)$ pebbles on sufficiently large cycles of length n. The quantity $\Omega(\log\log n)$ is exactly the threshold below which the gaps between pebble drops become superpolynomial. That is, for any polynomial f there are infinitely many time steps t such that no pebble is dropped between time t and time $f(t)$ with high probability. Then, for one of these big gaps, we can construct a combination lock with tail for which the following holds. With high probability, the algorithm drops no pebble within the combination lock and fails to reach the last few vertices of the lock in its allotted running time. Thus the robot fails to learn the graph. The idea of using combination locks with tails to foil a robot comes from Bender and Slonim's argument that a constant number of pebbles is insufficient [10]. The novel aspect of our proof is the analysis of pebble-dropping times to determine on which sizes of combination locks the algorithm fails.

We now turn to the details of the proof. Suppose, in contradiction to the claim in the theorem, that we have an expected polynomial-time algorithm A which succeeds in learning graphs with probability greater than 1/2, but does not use $\Omega(\log\log n)$ pebbles. Let $q(n) = \mathrm{O}(n^k)$ be a polynomial upper bound on the expected running time of the algorithm. In this proof, we use the standard technique of treating the randomized algorithm A as a distribution on deterministic algorithms A_r, *i.e.* for every infinite string $r \in \{0,1\}^{\mathbb{N}}$, A_r is the deterministic algorithm given by A using random coins r. All probabilities and expectations in this proof are taken over the choice of r.

We wish to study how the robot behaves when it doesn't see the pebbles it has dropped previously. To formalize this, we look at the infinite graph I on vertex set $\{w_1, w_2, \ldots\}$ where there are two edges labeled 0 and 1 from w_i to w_{i+1} for every $i \geq 1$. Now consider the behavior of the robot when it is placed at vertex w_1. Notice that when the robot drops a pebble at vertex w_i and moves, it never sees the pebble again. For $t \geq s \geq 1$, let $\mathrm{P}(s,t)$ be the probability that the robot drops at least one pebble between vertices w_s and w_{t-1}, inclusive, and let $\mathrm{E}(s,t)$ be the expected number of pebbles dropped by the robot between vertices w_s and w_{t-1}, so $\mathrm{E}(s,t) \geq \mathrm{P}(s,t)$. Notice that $\mathrm{E}(1,t)$ is a *lower bound* on the expected number of pebbles the robot uses on a cycle C_t of t vertices, because for every r, A_r's behavior in its first $t-1$ moves is the same in C_t as in I. We now use this to show that that there are superpolynomial gaps in the pebble-dropping times.

Claim: For every fixed $c > 0$, there are infinitely many t such that $\mathrm{P}(t, t^c) < 1/8$.

Proof of claim: Suppose not, *i.e.* there is some t_0 such that for all $t \geq t_0$, $\mathrm{P}(t,t^c) \geq 1/8$. Then for every $\ell \geq 0$,

$$\begin{aligned} \mathrm{E}(t_0, t_0^{c^\ell}) &= \sum_{j=1}^{\ell} \mathrm{E}(t_0^{c^{j-1}}, t_0^{c^j}) \\ &\geq \sum_{j=1}^{\ell} \mathrm{P}(t_0^{c^{j-1}}, t_0^{c^j}) \\ &\geq \ell/8. \end{aligned}$$

For $n \geq t_0$, let $\ell_n \stackrel{\text{def}}{=} \min\{\ell : \; n < t_0^{c^\ell}\}$. Then $\log\log n < \log\log t_0 + \ell_n \log c$, so $\ell_n = \Omega(\log\log n)$. We also have

$$\mathrm{E}(1,n) \;\geq\; \mathrm{E}(t_0,n) \;\geq\; \mathrm{E}(t_0, t_0^{c^{\ell_n - 1}}) \;\geq\; \frac{\ell_n - 1}{8} \;=\; \Omega(\log\log n).$$

But $\mathrm{E}(1,n)$ is a lower bound on the expected number of pebbles the robot uses on a cycle of length n, so we have a contradiction. $\Rightarrow\Leftarrow$

Recall that the expected running time of A is $q(n) = \mathrm{O}(n^k)$. Using the above claim with $c = k+1$, we can find a t with the following properties:

- $\mathrm{P}(t, t^{k+1}) < \frac{1}{8}$.

- $\frac{8q(2t+4)}{2^t} < \frac{1}{8}$.

- $t^{k+1} \geq 8q(2t+4)$.

Consider the random variable W which is a string consisting of the robot's first $8q(2t+4)$ moves in I. There are less than $|\mathrm{W}| = 8q(2t+4)$ contiguous subsequences of length t in W, so there is some string $\alpha \in \{0,1\}^t$ which occurs as a continguous subsequence of W with probability less than $8q(2t+4)/2^t < 1/8$. In other words there is a sequence of moves α of length t which the robot performs with probability less than 1/8 during its first $8q(2t+4)$ steps in I.

Let β by any binary string of length 4, and consider the behavior of the robot when placed at vertex u_1 in the combination lock $\mathrm{G}_\beta \stackrel{\mathrm{def}}{=} L_{\alpha\beta,t-1}$ with tail $t-1$ and combination $\alpha\beta$ (and vertex set $\{u_1,\ldots,u_{t-1},v_1,\ldots,v_{t+5}\}$ as above). Since A runs in expected time $q(n)$ and G_β has $2t+4$ vertices, the probability that A makes more than $8q(2t+4)$ moves in G_β is at most 1/8.

Let R_1 be the set of random coins r for which A_r would drop a pebble between vertex w_t and $w_{t^{k+1}-1}$ in I. Let R_2 be the set of random coins r for which A_r executes the sequence of moves α at some point during its first $8q(2t+4)$ moves in I. Let R_3 be the set of random coins r for which A_r makes more than $8q(2t+4)$ moves in G^β. Let $\mathrm{R} = \mathrm{R}_1 \cup \mathrm{R}_2 \cup \mathrm{R}_3$. We have shown that $\Pr[r \in \mathrm{R}] < 3/8$. Notice that for any $r \notin \mathrm{R}$, the output of A_r on G_β is the same as its output on G_γ for any string γ of length 4 because the robot never sees a pebble that it has dropped and never reaches vertex v_{t+1}. Let S_γ be the set of $r \notin \mathrm{R}$ on which A_r outputs G_γ when placed in G_γ (equivalently, G_β). Then since A has overall success probability at least 1/2, A must succeed on at least 1/8 of the $r \notin R$. So $\Pr[r \in \mathrm{S}_\gamma] > 1/8$. But there are 16 sets S_γ and they are disjoint. $\Rightarrow\Leftarrow$
■

5 Conclusions and Future Work

In this paper we studied the exploring capabilities of a robot that can drop and pick up pebbles in an unknown environment, modelled as an unknown directed graph with unlabeled and undistinguishable vertices. We showed that, if the robot knows an upper bound $\hat{n}$ on the number of vertices, n, it can deterministically learn the environment in polynomial time, while it needs $\Theta(\log\log n)$ pebbles to do the same if if does not know such a bound. The first result disproves a conjecture of Bender and Slonim [10] while the second presents a deterministic alternative to their randomized two-robot-based algorithm.

Future Research. The running time of our algorithms, though polynomial in the given parameters, leaves much to be desired. In particular, the algorithm for mapping an unknown graph given an upper bound $\hat{n}$ on the number of vertices and a single pebble, runs in time $O(\hat{n}^2 n^6 d^2)$. Thus one natural question is whether this running time can be significantly improved, either for the general case studied here or for special cases of interest.

Another question is how to adapt the algorithm to deal with uncertainty. For instance, what if the transitions taken by the robot are incorrect with some probability? (For example, upon taking an edge labeled i the robot ends at the vertex to which the edge labeled j goes.) [11] The correctness of our algorithm clearly relies on correct transitions. The question is whether any of

[11] Another standard form of uncertainty is with respect to possible observations the robot makes at vertices. Our algorithm can be viewed as dealing with this type of uncertainty by ignoring any such (possibly unreliable) information.

our techniques can be adapted to such a scenario, perhaps while making some assumptions about the graph. See [18] for further discussion on uncertainty in map learning. Even more generally, perhaps some of our ideas can be used for learning Partially Observable Markov Decision Processes (using some form of a pebble), in which for each *action* (edge label) there is a distribution on the next vertex.

References

[1] Rick Adams. Colossal cave adventure page. `http://people.delphi.com/rickadams/adventure/index.html`, April 2000.

[2] S. Albers and M. R. Henzinger. Exploring unknown environments. In *Proceedings of the Twenty Ninth Annual ACM Symposium on the Theory of Computing*, 1997.

[3] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, November 1987.

[4] D. Angluin, J. Westbrook, and W. Zhu. Robot navigation with range queries. In *Proceedings of the Twenty Eighth Annual ACM Symposium on the Theory of Computing*, pages 469–478, 1996.

[5] V. Anjan. *Doctoral Thesis*. PhD thesis, Mathematical Institute of the Academy of Sciences, Minsk, 1987.

[6] B. Awerbuch, M. Betke, R. L. Rivest, and M. Singh. Piecemeal graph exploration by a mobile robot. In *Proceedings of the Eighth Annual ACM Conference on Computational Learning Theory*, pages 321–328, 1995.

[7] R. Baeza-Yates, J. Culberson, and G. Rawlins. Searching in the plane. *Information and Computation*, pages 234–252, 1993.

[8] E. Bar-Eli, P. Berman, A. Fiat, and P. Yan. Online navigation in a room. *Journal of Algorithms*, 17(3):319–341, November 1994.

[9] M. Bender, A. Fernández, D. Ron, A. Sahai, and S. Vadhan. The power of a pebble: Exploring and mapping directed graphs. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 269–278, Dallas, TX, May 1998. ACM.

[10] M. A. Bender and D. Slonim. The power of team exploration: Two robots can learn unlabeled directed graphs. In *Proceedings of the Thirty Fifth Annual Symposium on Foundations of Computer Science*, pages 75–85, 1994.

[11] P. Berman, A. Blum, A. Fiat, H. Karloff, A. Rosen, and M. Saks. Randomized robot navigation algorithms. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 74–84, 1996.

[12] M. Betke, R. L. Rivest, and M. Singh. Piecemeal learning of an unknown environment. *Machine Learning*, 18(2/3):231–254, 1995.

[13] A. Blum and P. Chalasani. An on-line algorithm for improving performance in navigation. In *Proceedings of the Thirty Fourth Annual Symposium on Foundations of Computer Science*, pages 2–11, 1993.

[14] A. Blum, P. Raghavan, and B. Schieber. Navigating in unfamiliar geometric terrain. *SIAM Journal on Computing*, 26(1):110–137, January 1997.

[15] M. Blum and D. Kozen. On the power of the compass (or, why mazes are easier to search than graphs). In *Proceedings of the Nineteenth Annual Symposium on Foundations of Computer Science*, pages 132–142, October 1978.

[16] M. Blum and W. J. Sakoda. On the capability of finite automata in 2 and 3 dimensional space. In *Proceedings of the Eighteenth Annual Symposium on Foundations of Computer Science*, pages 147–161, 1977.

[17] T. Dean, D. Angluin, K. Basye, S. Engelson, L. Kaelbling, E. Kokkevis, and O. Maron. Inferring finite automata with stochastic output functions and an application to map learning. *Machine Learning*, 18(1):81–108, January 1995.

[18] T. Dean, K. Basye, and L. Kaelbling. Uncertainty in graph-based map learning. *Robot Learning*, 1992.

[19] X. Deng, T. Kameda, and C. Papadimitriou. How to learn an unknown environment I: The rectilinear case. *Journal of the ACM*, 45(2):215–245, March 1998.

[20] X. Deng, E. Milios, and A. Mirzaian. Robot map verification of a graph world. In *Algorithms and Data Structures (WADS '99)*, Lecture Notes in Computer Science, Vancouver, BC, Canada, August 1999. Springer-Verlag.

[21] X. Deng and A. Mirzaian. Competitive robot mapping with homogeneous markers. *IEEE Transactions on Robotics and Automation*, 12(4):532–542, August 1996.

[22] X. Deng and C. H. Papadimitriou. Exploring an unknown graph. In *Proceedings of the Thirty First Annual Symposium on Foundations of Computer Science*, pages 356–361, 1990.

[23] G. Dudek, M. Jenkin, E. Milios, and D. Wilkes. Robotic exploration as graph construction. *IEEE Transactions on Robotics and Automation*, 7(6):859–865, December 1991.

[24] G. Dudek, M. Jenkin, E. Milios, and D. Wilkes. Map validation and robot self-location in a graph-like world. *Robotics and Autonomous Systems*, 22(2):159–178, November 1997.

[25] Y. Freund, M. Kearns, Y. Mansour, D. Ron, R. Rubinfeld, and R. E. Schapire. Efficient algorithms for learning to play repeated games against computationally bounded adversaries. In *Proceedings of the Thirty Sixth Annual Symposium on Foundations of Computer Science*, pages 332–341, 1995.

[26] Y. Freund, M. Kearns, D. Ron, R. Rubinfeld, R. E. Schapire, and L. Sellie. Efficient learning of typical finite automata from random walks. *Information and Computation*, 138(1):23–48, 10 October 1997.

[27] F. Hoffman, C. Icking, R. Klein, and K. Kriegel. A competitive strategy for learning a polygon. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 166–174, 1997.

[28] Z. Kohavi. *Switching and Finite Automata Theory.* McGraw-Hill, second edition, 1978.

[29] V. B. Kudryavtsev, Sh. Ushchumlich, and G. Kilibarda. On the behavior of automata in labyrinths. *Discrete Math. and Applications*, 3:1–28, 1993.

[30] P. Panaite and A. Pelc. Exploring unknown undirected graphs. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, 1998.

[31] C.H. Papadimitriou and M. Yannakakis. Shortest paths without a map. *Theoretical Computer Science*, 84:127–150, 1991.

[32] M. O. Rabin. Maze threading automata. Seminar Talk presented at the University of California at Berkeley, October 1967.

[33] L. Reyzin. Traversal problems for certain types of deterministic and non-deterministic automata. Unpublished manuscript, 1992.

[34] R. Rivest and R. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, 1993.

[35] R. Rivest and R. Schapire. Diversity-based inference of finite automata. *Journal of the Association for Computing Machinery*, 43(3):555–589, 1994.

[36] D. Ron and R. Rubinfeld. Exactly learning automata of small cover time. *Machine Learning*, 27(1):69–96, 1997.

[37] E. M. Royer and C.-K. Toh. A review of current routing protocols for ad hoc mobile wireless networks. *IEEE Personal Communications*, 6(2):46–55, April 1999.

[38] A.N. Shah. Pebble automata on arrays. *Computer Graphics and Image Processing*, pages 236–246, 1974.

[39] L. Zhang. A survey of the problem of learning an unknown environment. Unpublished manuscript, 1994.