The Power of Team Exploration: Two Robots Can Learn Unlabeled Directed Graphs

Michael A. Bender*
Aiken Computation Laboratory
Harvard University
Cambridge, MA 02138
bender@das.harvard.edu

Donna K. Slonim†
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139
slonim@theory.lcs.mit.edu

September 6, 1995

Abstract

We show that two cooperating robots can learn exactly any strongly-connected directed graph with n indistinguishable nodes in expected time polynomial in n. We introduce a new type of homing sequence for two robots, which helps the robots recognize certain previously-seen nodes. We then present an algorithm in which the robots learn the graph and the homing sequence simultaneously by actively wandering through the graph. Unlike most previous learning results using homing sequences, our algorithm does not require a teacher to provide counterexamples. Furthermore, the algorithm can use efficiently any additional information available that distinguishes nodes. We also present an algorithm in which the robots learn by taking random walks. The rate at which a random walk on a graph converges to the stationary distribution is characterized by the conductance of the graph. Our random-walk algorithm learns in expected time polynomial in n and in the inverse of the conductance and is more efficient than the homing-sequence algorithm for high-conductance graphs.

*Supported by NSF Grant CCR-93-13775.
†Supported by NSF Grant CCR-93-10888 and NIH Grant 1 T32 HG00039-01.

1 Introduction

Consider a robot trying to construct a street map of an unfamiliar city by driving along the city's roads. Since many streets are one-way, the robot may be unable to retrace its steps. However, it can learn by using street signs to distinguish intersections. Now suppose that it is nighttime and that there are no street signs. The task becomes significantly more challenging.

In this paper we present a probabilistic polynomial-time algorithm to solve an abstraction of the above problem by using two cooperating learners. Instead of learning a city, we learn a strongly-connected directed graph G with n nodes. Every node has d outgoing edges labeled from 0 to $d-1$. Nodes in the graph are indistinguishable, so a robot cannot recognize if it is placed on a node that it has previously seen. Moreover, since the graph is directed, a robot is unable to retrace its steps while exploring.

For this model, one might imagine a straightforward learning algorithm with a running time polynomial in a specific property of the graph's structure such as cover time or mixing time. Any such algorithm could require an exponential number of steps, however, since the cover time and mixing time of directed graphs can be exponential in the number of nodes. In this paper, we present a probabilistic algorithm for two robots to learn any strongly-connected directed graph in $O(d^2n^5)$ steps with high probability.

The two robots in our model can recognize when they are at the same node and can communicate freely by radio. Radio communication is used only to synchronize actions. In fact, if we assume that the two robots move synchronously and share a polynomial-length random string, then no communication is necessary. Thus with only minor modifications, our algorithms may be used in a distributed setting.

Our main algorithm runs without prior knowledge of the number of nodes in the graph, n, in time polynomial in n. We show that no probabilistic polynomial-time algorithm for a single robot with a constant number of pebbles can learn all unlabeled directed graphs when n is unknown. Thus, our algorithms demonstrate that two robots are strictly more powerful than one.

1.1 Related Work

Previous results showing the power of team learning are plentiful, particularly in the field of inductive inference (see Smith [Smi94] for an excellent survey). Several team learning papers explore the problems of combining the abilities of a number of different learners. Cesa-Bianchi *et al.* [CBF+93] consider the task of learning a probabilistic binary sequence given the predictions of a set of experts on the same sequence. They show how to combine the prediction strategies of several experts to predict nearly as well as the best of the experts. In a related paper, Kearns and Seung [KS93] explore the statistical problems of combining several independent hypotheses to learn a target concept from a known, restricted concept class. In their model, each hypothesis is learned from a different, independently-drawn set of random examples, so the learner can combine the results to perform significantly better than any of the hypotheses alone.

There are also many results on learning unknown graphs, but most previous work has concentrated on learning undirected graphs or graphs with distinguishable nodes. For example, Deng and

Papadimitriou consider the problem of learning strongly-connected, directed graphs with *labeled* nodes, so that the learner can recognize previously-seen nodes. They provide a learning algorithm whose competitive ratio (versus the optimal time to traverse all edges in the graph) is exponential in the deficiency of the graph [DP90, Bet92]. Betke, Rivest, and Singh introduce the notion of *piecemeal* learning of undirected graphs with labeled nodes. In piecemeal learning, the learner must return to a fixed starting point from time to time during the learning process. Betke, Rivest, and Singh provide linear algorithms for learning grid graphs with rectangular obstacles [BRS93], and with Awerbuch [ABRS95] extend this work to show nearly-linear algorithms for general graphs.

Rivest and Schapire [RS87, RS93] explore the problem of learning deterministic finite automata whose nodes are *not* distinguishable except by the observed output. We rely heavily on their results in this paper. Their work has been extended by Freund *et al.* [FK+93], and by Dean *et al.* [DA+92]. Freund *et al.* analyze the problem of learning finite automata with average-case labelings by the observed output on a random string, while Dean *et al.* explore the problem of learning DFAs with a robot whose observations of the environment are not always reliable. Ron and Rubinfeld [RR95] present algorithms for learning "fallible" DFAs, in which the data is subject to persistent random errors. Recently, Ron and Rubinfeld [RR95b] have shown that a teacher is unnecessary for learning finite automata with small cover time.

In our model a single robot is powerless because it is completely unable to distinguish one node from any other. However, when equipped with a number of pebbles that can be used to mark nodes, the single robot's plight improves. Rabin first proposed the idea of dropping pebbles to mark nodes [Rab67]. This suggestion led to a body of work exploring the searching capabilities of a finite automaton supplied with pebbles. Blum and Sakoda [BS77] consider the question of whether a finite set of finite automata can search a 2 or 3-dimensional obstructed grid. They prove that a single automaton with just four pebbles can completely search any 2-dimensional finite maze, and that a single automaton with seven pebbles can completely search any 2-dimensional infinite maze. They also prove, however, that no collection of finite automata can search every 3-dimensional maze. Blum and Kozen [BK78] improve this result to show that a single automaton with 2 pebbles can search a finite, 2-dimensional maze. Their results imply that mazes are strictly easier to search than planar graphs, since they also show that no single automaton with pebbles can search all planar graphs. Savitch [Sav73] introduces the notion of a maze-recognizing automaton (MRA), which is a DFA with a finite number of distinguishable pebbles. The mazes in Savitch's paper are n-node 2-regular graphs, and the MRAs have the added ability to jump to the node with the next higher or lower number in some ordering. Savitch shows that maze-recognizing automata and $\log n$ space-bounded Turing machines are equivalent for the problem of recognizing threadable mazes (i.e., mazes in which there is a path between a given pair of nodes).

Most of these papers use pebbles to model memory constraints. For example, suppose that the nodes in a graph are labeled with $\log n$-bit names and that a finite automaton with $k \log n$ bits of memory is used to search the graph. This situation is modeled by a single robot with k distinguishable pebbles. A robot dropping a pebble at a node corresponds to a finite automaton storing the name of that node. In our paper, by contrast, we investigate time rather than space constraints. Since memory is now relatively cheap but time is often critical, it makes sense to ask whether a robot with any reasonable amount of memory can use a constant number of pebbles to learn graphs in polynomial time.

Cook and Rackoff generalized the idea of pebbles to jumping automata for graphs (JAGs) [CR80]. A jumping automaton is equipped with pebbles that can be dropped to mark nodes and that can "jump" to the locations of other pebbles. Thus, this model is similar to our two-robot model in that the second robot may wait at a node for a while (to mark it) and then catch up to the other robot later. However, the JAG model is somewhat broader than the two-robot model. Cook and Rackoff show upper and lower bounds of $\log n$ and $\log n / \log\log n$ on the amount of space required to determine whether there is a directed path between two designated nodes in any n-node graph. JAGs have been used primarily to prove space efficiency for st-connectivity algorithms, and they have recently resurfaced as a tool for analyzing time and space tradeoffs for graph traversal and connectivity problems (*e.g.* [BB+90, Poo93, Edm93]).

Universal traversal sequences have been used to provide upper and lower bounds for the exploration of undirected graphs. Certainly, a universal traversal sequence for the class of directed graphs could be used to learn individual graphs. However, for arbitrary directed graphs with n nodes, a universal traversal sequence must have size exponential in n. Thus, such sequences will not provide efficient solutions to our problem.

1.2 Strategy of the Learning Algorithm

The power behind the two-robot model lies in the robots' abilities to recognize each other and to move independently. Nonetheless, it is not obvious how to harness this power. If the robots separate in unknown territory, they could search for each other for an amount of time exponential in the size of the graph. Therefore, in any successful strategy for our model the two robots must always know how to find each other. One strategy that satisfies this requirement has both robots following the same path whenever they are in unmapped territory. They may travel at different speeds, however, with one robot scouting ahead and the other lagging behind. We call this a *lead-lag strategy.* In a lead-lag strategy the lagging robot must repeatedly make a difficult choice. The robot can wait at a particular node, thus marking it, but the leading robot may not find this marked node again in polynomial time. Alternatively, the lagging robot can abandon its current node to catch up with the leader, but then it may not know how to return to that node. In spite of these difficulties, our algorithms successfully employ a lead-lag strategy.

Our work also builds on techniques of Rivest and Schapire [RS93]. They present an algorithm for a single robot to learn minimal deterministic finite automata. With the help of an equivalence oracle, their algorithm learns a homing sequence, which it uses in place of a reset function. It then runs several copies of Angluin's algorithm [Ang87] for learning DFAs given a reset. Angluin has shown that any algorithm for actively learning DFAs requires an equivalence oracle [Ang81].

In this paper, we introduce a new type of homing sequence for two robots. Because of the strength of the homing sequence, our algorithm does not require an equivalence oracle. For any graph, the expected running time of our algorithm is $O(d^2n^5)$. In practice, our algorithm can use additional information such as indegree, outdegree, or color of nodes to find better homing sequences and to run faster.

Note that throughout the paper, the analyses of the algorithms account for only the number of steps that the robots take across edges in the graph. Additional calculations performed between moves are not considered, so long as they are known to take time polynomial in n. In practice, such calculations would not be a noticeable factor in the running time of our algorithms.

Two robots can learn specific classes of directed graphs more quickly, such as the class of graphs with high conductance. Conductance, a measure of the expansion properties of a graph, was introduced by Sinclair and Jerrum [SJ89]. The class of directed graphs with high conductance includes graphs with exponentially-large cover time. We present a randomized algorithm that learns graphs with conductance greater than $n^{-\frac{1}{2}}$ in $O(dn^4 \log n)$ steps with high probability.

2 Preliminaries

Let $G = (V, E)$ represent the unknown graph, where G has n nodes, each with outdegree d. An edge from node u to node v with label i is denoted $\langle u, i, v\rangle$. We say that an algorithm *learns* graph G if it outputs a graph isomorphic to G. Our algorithms maintain a graph *map* which represents the subgraph of G learned so far. Included in *map* is an implicit start node u_0. It is worth emphasizing the difference between the target graph G and the graph *map* that the learner constructs. The graph *map* is meant to be a map of the underlying environment, G. However, since the robots do not always know their exact location in G, in some cases *map* may contain errors and therefore may not be isomorphic to any subgraph of G. Much of the notation in this section is needed to specify clearly whether we are referring to a robot's location in the graph G or to its putative location in *map*.

A node u in *map* is called *unfinished* if it has any unexplored outgoing edges. Node u is *map-reachable* from node v if there is a path from v to u containing only edges in *map*. For robot k, the node in *map* corresponding to k's location in G if *map* is correct is denoted $Loc_M(k)$. Robot k's location in G is denoted $Loc_G(k)$.

Let f be an automorphism on the nodes of G such that

$$\forall a, b \in G, \langle a, i, b\rangle \in G \iff \langle f(a), i, f(b)\rangle \in G.$$

We say *nodes* c and d are *equivalent* (written $c \equiv d$) iff there exists such an f where $f(c) = d$.

We now present notation to describe the movements of k robots in a graph. An *action* A_i of the ith robot is either a label of an outgoing edge to explore, or the symbol r for "rest." A *k-robot sequence of actions* is a sequence of *steps* denoting the actions of the k robots; each step is a *k-tuple* $\langle A_0, \ldots, A_{k-1}\rangle$. For sequences s and t of actions, $s \circ t$ denotes the sequence of actions obtained by concatenating s and t.

A *path* is a sequence of edge labels. Let $|path|$ represent the length of *path*. A robot *follows* a path by traversing the edges in the path in order beginning at a particular start node in *map*. The node in *map* reached by starting at u_0 and following *path* is denoted $final_M(path, u_0)$. Let s be a two-robot sequence of actions such that if both robots start together at any node in any graph and execute s, they follow exactly the same path, although perhaps at different speeds. We call such a sequence a *lead-lag* sequence. Note that if two robots start together and execute a lead-lag sequence, they end together. The node in G reached if both robots start at node a in G and follow lead-lag sequence s is denoted $final_G(s, a)$.

For convenience, we name our robots Lewis and Clark. Whenever Lewis and Clark execute a lead-lag sequence of actions, Lewis leads while Clark lags behind.

3 Using a Reset to Learn

Learning a directed graph with indistinguishable nodes is difficult because once both robots have left a known portion of the graph, they do not know how to return. This problem would vanish if there were a reset function that could transport both robots to a particular start node u_0. We describe an algorithm for two robots to learn directed graphs given a reset. Having a reset is not a realistic model, but this algorithm forms the core of later algorithms, which learn without a reset.

Algorithm **Learn-with-Reset** maintains the invariant that if a robot starts at u_0, there is a directed path it can follow that visits every node in *map* at least once. To learn a new edge (one not yet in *map*) using algorithm **Learn-with-Reset**, Lewis crosses the edge and then Clark tours the entire known portion of the map. If they encounter each other, Lewis's position is identified; otherwise Lewis is at a new node. The depth-first strategy employed by **Learn-Edge** is essential in later algorithms. In **Learn-with-Reset**, as in all the procedures in this paper, variables are passed by reference and are modified destructively.

Lemma 1 *The variable* path *in* **Learn-with-Reset** *denotes a tour of length* $\leq dn^2$ *that starts at* u_0 *and traverses all edges in map.*

Learn-with-Reset( ):
$map := (\{u_0\}, \emptyset)$ — { map is the graph consisting of node u_0 and no edges }
$path :=$ empty path — { $path$ is the null sequence of edge labels }
$k := 1$ — { k counts the number of nodes in map }
while there are unfinished nodes in map
    do Learn-Edge(map,$path$,k)
        Reset
return map

Learn-Edge(map,$path$,k**):** — {$path$ = tour through all edges in map }
Lewis follows $path$ to $final_M(path, u_0)$
$u_i :=$ some unfinished node in map map-reachable from Loc_M(Lewis)
Lewis moves to node u_i; append the path taken to $path$
pick an unexplored edge l out of node u_i
Lewis moves along edge l; append edge l to $path$ — { Lewis crosses a new edge }
Clark follows $path$ to $final_M(path, u_0)$ — { Clark looks for Lewis }
if $\exists j < k$ such that Clark first encountered Lewis at node u_j
    then add edge $\langle u_i, l, u_j \rangle$ to map
    else add new node u_k and edge $\langle u_i, l, u_k \rangle$ to map
        $k := k + 1$

Proof: Every time Lewis crosses an edge, that edge is appended to *path*. Since no edge is added to *map* until Lewis has crossed it, *path* must traverse all edges in *map*. In each call to **Learn-Edge**, at most n edges are added to *path*. The body of the while loop is executed dn times, so $|path| \leq dn^2$. □

Lemma 2 *Map is always a subgraph of G.*

Proof: Initially *map* contains a single node u_0 and no edges. Assume inductively that *map* is a subgraph of G after the cth call to **Learn-Edge** (when *map* has c edges). To learn the next edge, the algorithm chooses a node u_i in *map* and explores a new edge $e = \langle u_i, l, v \rangle$. By Lemma 1 and the inductive hypothesis, if Clark encounters Lewis at u_j then v is identified as u_j. Otherwise v is recognized to be a new node and named u_k. Therefore the updated map is a subgraph of G. □

Lemma 3 *If map contains any unfinished nodes, then there is always some unfinished node in* map *map-reachable from* final_M *(*path,u_0*).*

Proof: Suppose this assumption were false. Then there is some unfinished node in *map*, but all nodes of *map* in the strongly-connected component of $final_M(path, u_0)$ are finished. Thus by Lemma 2, there are no additional edges of G leaving that component, so graph G is not strongly connected. □

Theorem 4 *After $O(d^2n^3)$ moves and dn calls to* **Reset**, **Learn-with-Reset** *halts and outputs a graph isomorphic to G.*

Proof: The correctness of the output follows from Lemmas 1 – 3. For each call to **Learn-Edge**, each robot takes *length*(*path*)$\leq dn^2$ steps. The algorithm **Learn-Edge** is executed at most dn times, so the algorithm halts within $O(d^2n^3)$ steps. □

4 Homing Sequences

In practice, robots learning a graph do not have access to a reset function. In this section we suggest an alternative technique: we introduce a new type of homing sequence for two robots.

Intuitively, a homing sequence is a sequence of actions whose observed output uniquely determines the final node reached in G. Rivest and Schapire [RS93] show how a single robot with a teacher can use homing sequences to learn strongly-connected minimal DFAs. The output at each node indicates whether that node is an accepting or rejecting state of the automaton. If the target DFA is not minimal, their algorithm learns the minimal encoding of the DFA. In other words, their algorithm learns the function that the graph computes rather than the structure of the graph.

In unlabeled graphs the nodes do not produce output. However, two robots can generate output indicating when they meet.

Definitions: Each step of a two-robot sequence of actions produces an output symbol T if the robots are together and S if they are separate. An *output sequence* is a string in $\{T, S\}^*$ denoting the observed output of a sequence of actions. Let s be a lead-lag sequence of actions and let a be a node in G. Then *output(s,a)* denotes the output produced by executing the sequence s, given that *both* robots start at a. A lead-lag sequence s of actions is a *two-robot homing sequence* iff $\forall$ nodes $u, v \in G$,

$$output(s,u) = output(s,v) \Rightarrow final_G(s,u) \equiv final_G(s,v).$$

Because the output of a sequence depends on the positions of both robots, it provides information about the underlying structure of the graph. Figure 1 illustrates the definition of a two-robot homing sequence. This new type of homing sequence is powerful. Unlike most previous learning results using homing sequences, our algorithms do not require a teacher to provide counterexamples.

In fact, two robots on a graph define a DFA whose states are pairs of nodes in G and whose edges correspond to pairs of actions. Since the automata defined in this way form a restricted class of DFAs, our results are not inconsistent with Angluin's work [Ang81] showing that a teacher is necessary for learning general DFAs.

Theorem 5 *Every strongly-connected directed graph has a two-robot homing sequence.*

Proof: The following algorithm (based on that of Kohavi [Koh78, RS93]) constructs a homing sequence: Initially, let h be empty. As long as there are two nodes u and v in G such that *output(h,u)* = *output(h,v)* but *final(h,u)* $\not\equiv$ *final(h,v)*, let x be a lead-lag sequence whose output distinguishes *final(h,u)* from *final(h,v)*. Since *final(h,u)* $\not\equiv$ *final(h,v)* and G is strongly connected, such an x always exists. Append x to h.

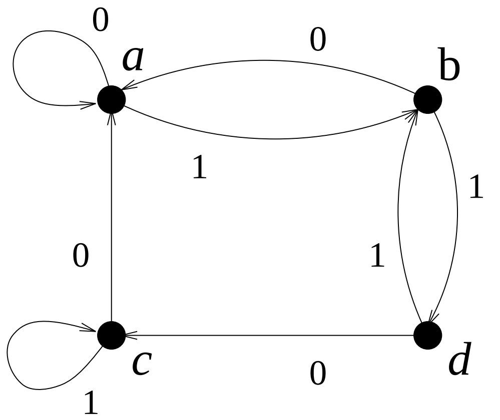
Each time a sequence is appended to h, the number of different outputs of h increases by at least 1. Since G has n nodes, there are at most n possible output sequences. Therefore, after $n-1$ iterations, h is a homing sequence. □

In Section 5 we show that it is possible to find a counterexample x efficiently. Given a strongly-connected graph G and a node a in G, a pair of robots can verify whether they are together at a node equivalent to a on some graph isomorphic to G. We describe a verification algorithm **Verify(**a, G**)** in Section 5. The sequence of actions returned by a call of **Verify(**u, G**)** is always a suitable counterexample x. Using the bound from Corollary 8, we claim that this algorithm produces a homing sequence of length $O(n^4)$ for all graphs. Note that shorter homing sequences exist; the homing sequence produced by algorithm **Learn-Graph** in Section 5 has length $O(dn^3)$.

4.1 Using a Homing Sequence to Learn

Given a homing sequence h, an algorithm can learn G by maintaining several running copies of **Learn-with-Reset**. Instead of a single start node, there are as many as n possible start nodes, each corresponding to a different output sequence of h. Note that many distinct output sequences may be associated with the same final node in G.

The new algorithm **Learn-with-HS** maintains several copies of *map* and *path*, one for each output sequence of h. Thus, graph map_c denotes the copy of the map associated with output



$$\text{output} = \begin{cases} T & \text{if Lewis and Clark are together at a node} \\ S & \text{if Lewis and Clark are at separate nodes.} \end{cases}$$

$$\text{homing sequence } h = \begin{cases} \text{Lewis:} & 0r1r \\ \text{Clark:} & r0r1 \end{cases} \qquad \text{sequence } s = \begin{cases} \text{Lewis:} & 0r \\ \text{Clark:} & r0 \end{cases}$$

start node	output	end node
a	$TTST$	b
b	$STST$	b
c	$STST$	b
d	$STTT$	c

start node	output	end node
a	TT	a
b	ST	a
c	ST	a
d	ST	c

Figure 1: Illustration of a two-robot homing sequence and a lead-lag sequence. Note that both h and s are lead-lag sequences. However, sequence h is a two-robot homing sequence, because for each output sequence there is a unique end node. (Note that the converse is not true.) Sequence s is not a two-robot homing sequence, because the robots may end at nodes a or c and yet see the same output sequence ST.

sequence c. Initially, Lewis and Clark are at the same node. Whenever algorithm **Learn-with-Reset** would use a reset, **Learn-with-HS** executes the homing sequence h. If the output of h is c, the algorithm learns a new edge in map_c as if it had been reset to u_0 in map_c (see Figure 2). After each execution of h, the algorithm learns a new edge in some map_c. Since there are at most n copies, each with dn edges to learn, eventually one map will be completed. Recall that a homing sequence is a lead-lag sequence. Therefore, at the beginning and end of every homing sequence the two robots are together.

Theorem 6 *If* **Learn-with-HS** *is called with a homing sequence h as input, it halts within $O(d^2n^4 + dn^2|h|)$ steps and outputs a graph isomorphic to G.*

Proof: The algorithm **Learn-with-HS** maintains at most n running versions of **Learn-with-Reset**, one for each output of the homing sequence. In particular, whenever the two robots execute

```
Learn-with-HS(h):
done := FALSE
while not done
    do execute h; c := the output sequence produced              { instead of a reset }
        if map_c is undefined
            then map_c := ({u_0}, ∅)    { map_c = graph consisting of node u_0 and no edges }
                path_c := empty path          { path_c is the null sequence of edge labels }
                k_c := 1                       { k_c counts the number of nodes in map }
        Learn-Edge(map_c,path_c,k_c)
        if map_c has no unfinished nodes
            then done := TRUE
return map_c
```

a homing sequence and obtain an output c, they have identified their position as the start node u_0 in map_c, and can learn one new edge in map_c before executing another homing sequence.

Eventually, one of the versions halts and outputs a complete map_c. Therefore, the correctness of **Learn-with-HS** follows directly from Theorem 4 and the definition of a two-robot homing sequence.

Let $r = O(d^2n^3)$ be the number of steps taken by **Learn-with-Reset**. Since there are at most n start nodes, **Learn-with-HS** takes at most $nr + dn^2|h|$ steps. □

5 Learning a Homing Sequence

Unlike a reset function, a two-robot homing sequence can be learned. The algorithm **Learn-Graph** maintains a candidate homing sequence h and improves h as it learns G.

Definition: Candidate homing sequence h is called a *bad* homing sequence if there exist nodes u, v, $u \neq v$, such that $\mathit{output}(h, u) = \mathit{output}(h, v)$, but $\mathit{final}_G(h, u) \not\equiv \mathit{final}_G(h, v)$.

Definition: Let a be a node in G. We say that map_c with start node u_0 is a *good representation* of $\langle a, G\rangle$ iff there exists an isomorphism f from the nodes in $map_c = (V^c, E^c)$ to the nodes in a subgraph $G' = (V', E')$ of G, such that $f(u_0) = a$, and

$$\forall u_i, u_j \in V^c, \langle u_i, \ell, u_j\rangle \in E^c \iff \langle f(u_i), \ell, f(u_j)\rangle \in E'.$$

In algorithms **Learn-with-Reset** and **Learn-with-HS**, the graphs *map* and map_c are *always* good representations of G. In **Learn-Graph** if the candidate homing sequence h is bad, a particular map_c may not be a good representation of G. However, the algorithm *can* test for such maps. Whenever a map_c is shown to be in error, h is improved and all maps are discarded. By the proof of Theorem 5, we know that a candidate homing sequence must be improved at most $n - 1$ times. In Section 5.1 we explain how to use adaptive homing sequences to discard only one map per improvement.

We now define a test that with probability at least $1/n$ detects an error in map_c if one exists.

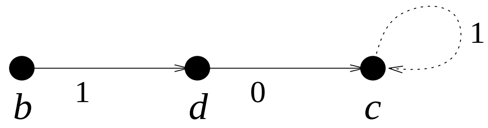
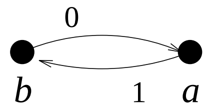
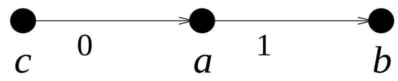
output of homing sequence	starting node of *map*	map
$TTST$	b	
$STST$	b	
$STTT$	c	

Figure 2: A possible "snapshot" of the learners' knowledge during an execution of **Learn-with-HS**. The robots are learning the graph G from Figure 1 using the two-robot homing sequence h from Figure 1. (Node names in maps are *not* known to the learner, but are added for clarity.) The following example demonstrates how the robots learn a new edge using **Learn-with-HS**. Suppose that the robots execute h and see output $TTST$. Then the robots are together at node b. Lewis follows path $1, 0$ to unfinished node c and then crosses the edge labeled 1. Now Clark follows path $1, 0, 1$. Since Clark sees Lewis after 2 steps, the dotted edge is added to map_{TTST}. Next, the robots execute h again and see output $STST$. Thus, they go on to learn some edge in map_{STST}.

Definition: Let $path_c$ be a path such that a robot starting at u_0 and following $path_c$ traverses every edge in $map_c = (V^c, E^c)$. Let $u_0 \ldots u_m$ be the nodes in V^c numbered in order of their first appearance in $path_c$. If both robots are at u_0 then $test_c(u_i)$ denotes the following lead-lag sequence of actions: (1) Lewis follows $path_c$ to the first occurrence of u_i; (2) Clark follows $path_c$ to the first occurrence of u_i; (3) Lewis follows $path_c$ to the end; (4) Clark follows $path_c$ to the end.

Definition: Given map_c and any lead-lag sequence t of actions, define $expected(t,\ map_c)$ to be the expected output if map_c is correct and if both robots start at node u_0 and execute sequence t. We abbreviate $expected(test_c(u_i), map_c)$ by $expected(test_c(u_i))$.

Lemma 7 *Suppose Lewis and Clark are both at some node a in G. Let* path_c *be a path such that a robot starting at u_0 and following* path_c *traverses every edge in* map_c. *Then* map_c *is a good representation of $\langle a, G\rangle$ iff* $\forall u_i \in V^c$, $\mathrm{output}(\mathrm{test}_c(u_i)) = \mathrm{expected}(\mathrm{test}_c(u_i))$.

Proof:
($\Longrightarrow$): By definition of good representation and $expected(\mathrm{test}_c(u_i))$.
($\Longleftarrow$): Suppose that all tests produce the expected output. We define a function f as follows: Let $f(u_0) = a$. Let $p(u_i)$ be the prefix of $path_c$ up to the first occurrence of u_i. Define $f(u_i)$ to be the

```
Learn-Graph():
done := FALSE
h := Λ (empty sequence)
while not done
    do execute h; c := the output sequence produced                         { instead of a reset }
        if map_c is undefined
            then map_c := ({u_0}, ∅)      { map_c is the graph consisting of node u_0 and no edges }
                path_c := empty path                 { path_c is the null sequence of edge labels }
                k_c := 1                              { k_c counts the number of nodes in map }
        if map_c has no unfinished node map-reachable from final_M(path_c)
            then Lewis and Clark move to final_M(path_c)
                comp := maximal strongly-connected component in map_c containing final_M(path_c)
                h-improve := Verify(final_M(path_c),comp)
                if h-improve = Λ
                    then done := TRUE                                       { map_c is complete }
                    else                                       { h-improve ≠ Λ. error detected }
                        append h-improve to end of h               { improve homing sequence ...}
                        discard all maps and paths      { ...and start learning maps from scratch }
            else v := value of a fair 0/1 coin flip             { learn edges or test for errors? }
                if v = 0                                                      { test for errors }
                    then u_i := a random node in map_c             { randomly pick node to test }
                        h-improve := Test(map_c, path_c, i)
                        if h-improve ≠ Λ                                       { error detected }
                            then append h-improve to end of h      { improve homing sequence ...}
                                discard all maps and paths   { ...start learning maps from scratch }
                    else Learn-Edge(map_c,path_c,k_c)
return map_c
```

```
Test(map_c, path_c, i):    {u_0, u_1, ..., u_k = the nodes in map_c indexed by first appearance in path_c }
h-improve := the following sequence of actions:
    Lewis follows path_c to the first occurrence of u_i in path_c
    Clark follows path_c to the first occurrence of u_i in path_c
    Lewis follows path_c to the end
    Clark follows path_c to the end
if output(h-improve) ≠ expected-output(h-improve)                              { if error detected }
    then return h-improve                                                       { return test_c(u_i) }
    else return Λ                                                          { return empty sequence }
```

```
Verify(v_0, map):             { v_0, v_1, ..., v_k are the nodes in map ordered by first appearance in p}
path := path such that a robot starting at v_0 in map and following path visits all nodes
in map and returns to v_0
for each i, 0 ≤ i < k
    do h-improve := Test(map, path, i)
        if h-improve ≠ Λ
            then return h-improve
return Λ
```

node in G that a robot reaches if it starts at a and follows $p(u_i)$. Let $G' = (V', E')$ be the image of $f(map_c)$ on (V, E).

We first show that f is an isomorphism from V^c to V'. By definition of G', f must be surjective. To see that f is injective, assume the contrary. Then there exist two nodes $u_i, u_j \in V^c$ such that $i \neq j$ but $f(u_i) = f(u_j)$. But then *output(*$\mathrm{test}_c(u_i)$*)* $\neq$ *expected(*$\mathrm{test}_c(u_i)$*)*, which contradicts our assumption that all tests succeed. Next, we show that $\langle u_i, \ell, u_j\rangle \in V^c \iff \langle f(u_i), \ell, f(u_j)\rangle \in V'$, proving that map_c is a good representation of $\langle a, G\rangle$.

($\Longleftarrow$): By definition of G', the image of map_c.
($\Longrightarrow$): Inductively assume that $\langle u_i, \ell, u_j\rangle \in V^c \iff \langle f(u_i), \ell, f(u_j)\rangle \in V'$ for the first m edges in $path_c$, and suppose that this prefix of the path visits only nodes $u_0 \ldots u_i$. Now consider the $(m+1)$st edge $e = \langle a, \ell, b\rangle$. There are two possibilities. In one case, edge e leads to some new node u_{i+1}. Then by definition $f(u_{i+1})$ is b's image in G, so $\langle f(a), \ell, f(b)\rangle \in G'$. Otherwise e leads to some previously-seen node u_{i-k}. Suppose that $f(u_{i-k})$ is *not* the node reached in G by starting at u_0 and following the first $m+1$ edges in $path_c$. Then *output(test*$_c(u_{i-k})$*)* $\neq$ *expected(test*$_c(u_{i-k})$*)*, so *test*$_c(u_{i-k})$ fails, and we arrive at a contradiction. Therefore $f(b) = f(u_{i-k})$ and $\langle f(a), \ell, f(b)\rangle \in G'$. □

Corollary 8 *Suppose Lewis and Clark are together at u_0 in* map_c*. Let* map_c *be strongly connected and have n nodes, $u_0, \ldots u_{n-1}$. Then the two robots can verify whether* map_c *is a good representation of $\langle Loc_G(Lewis), G\rangle$ in $O(n^3)$ steps.*

Proof: Since map_c is strongly connected, there exists a path $path_c$ with the following property: a robot starting at u_0 and following $path_c$ visits all nodes in map_c and returns to u_0. Index the remaining nodes in order of their first appearance in $path_c$. The two robots verify whether, for all u_i in order, *output(test*$_c(u_i)$*)* = *expected(test*$_c(u_i)$*)*. Note that Lewis and Clark are together at u_0 after each test. By Lemma 7, this procedure verifies map_c. Since $path_c$ has length $O(n^2)$, each test has length $O(n^2)$, so verification requires $O(n^3)$ steps. □

In **Learn-Graph** after the robots execute a homing sequence, they randomly decide either to learn a new edge or to test a random node in map_c. The following lemma shows that a test that failed can be used to improve the homing sequence.

Lemma 9 *Let h be a candidate homing sequence in* **Learn-Graph***, and let u_k be a node such that* $\mathrm{output}(\mathrm{test}_c(u_k)) \neq \mathrm{expected}(\mathrm{test}_c(u_k))$*. Then there are two nodes a, b in G that h does not distinguish but that $h \circ \mathrm{test}_c(u_k)$ does.*

Proof: Let a be a node in G such that when both robots start at a, *output(test*$_c(u_k)$*)* $\neq$ *expected(test*$_c(u_k)$*)*. Suppose that at step i in *test*$_c(u_k)$, the expected output is T (respectively S), but the actual output is S (resp. T). Each edge in $path_c$ and map_c was learned using **Learn-Edge**. If map_c indicates that the ith node in $path_c$ is u_k, there must be a start node b in G where u_k really is the ith node in $path_c$. Since *output(test*$_c(u_k)$*)* $\neq$ *expected(test*$_c(u_k)$*)*, the sequence $h \circ$ *test*$_c(u_k)$ distinguishes a from b. □

The algorithm **Learn-Graph** runs until there are no more map-reachable unexplored nodes in some map_c. If map_c is not strongly connected, then it is not a good representation of G. In this case, the representation of the last strongly-connected component on *path* must be incorrect. Thus, calling **Verify** on this component from the last node on *path* returns a sequence that improves h. If map_c is strongly connected, then either **Verify** returns an improvement to the homing sequence, or map_c is a good representation of G.

Before we can prove the correctness of our algorithm, we need one more set of tools. Consider the following statement of Chernoff bounds from Raghavan [Rag89].

Lemma 10 *Let $X_1, \ldots, X_m$ be independent Bernoulli trials with $E[X_j] = p_j$. Let the random variable $X = \sum_{j=1}^{m} X_j$, where $\mu = E[X] \geq 0$. Then for $\delta > 0$,*

$$Pr[X > (1+\delta)\,\mu] < \left[\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right]^{\mu},$$

and

$$Pr[X < (1-\delta)\,\mu] < e^{-\mu\delta^2/2}.$$

In our analysis in this section and in Section 6, the random variables may not be independent. However, the following corollary bounds the conditional probabilities. The proof of this corollary is exactly analogous to that of a similar corollary by Aumann and Rabin [AR94, Corollary 1].

Corollary 11 *Let $X_1, \ldots, X_m$ be 0/1 random variables (not necessarily independent), and let $b_j \in \{0,1\}$ for $1 \leq j \leq m$. Let the random variable $X = \sum_{j=1}^{m} X_j$. For any $b_1, \ldots, b_{j-1}$ and $\delta > 0$, if $Pr[X_j = 1 | X_1 = b_1, X_2 = b_2, \ldots, X_{j-1} = b_{j-1}] \leq p_j$ and $\mu = \sum_{j=1}^{m} p_j > 0$, then*

$$Pr[X > (1+\delta)\,\mu] < \left[\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right]^{\mu},$$

and for any $b_1, \ldots, b_{j-1}$ and $\delta > 0$, if $Pr[X_j = 1 | X_1 = b_1, X_2 = b_2, \ldots, X_{j-1} = b_{j-1}] \geq p_j$ and $\mu = \sum_{j=1}^{m} p_j > 0$, then

$$Pr[X < (1-\delta)\,\mu] < e^{-\mu\delta^2/2}.$$

Theorem 12 *The algorithm* **Learn-Graph** *always outputs a map isomorphic to G and halts in $O(d^2 n^6)$ steps with overwhelming probability ($1 - e^{-cn}$, where constant $c > 0$ can be chosen as needed).*

Proof: Since **Learn-Graph** verifies map_c before finishing, if the algorithm terminates then by Corollary 8 it outputs a map isomorphic to G. It is therefore only necessary to show that the algorithm runs in $O(d^2 n^6)$ steps with overwhelming probability.

In each iteration of the while loop in **Learn-Graph**, if there are no map-reachable unfinished nodes, then the algorithm attempts to verify the map. Otherwise, the algorithm decides randomly whether to learn a new edge or to test a random node in the graph. It follows from Lemma 10 that a constant fraction of the random decisions are for learning and a constant fraction are for testing.

By Theorem 4 the total number of steps spent learning edges in each version of *map* is $O(d^2n^3)$. For each candidate homing sequence, there are n versions of *map*, and the candidate homing sequence is improved at most n times. Thus, $O(d^2n^5)$ steps are spent learning nodes and edges.

We consider the number of steps taken testing nodes. Each test requires $|path| = O(dn^2)$ steps. Once a map contains an error, the probability that the robots choose to test a node that is in error is at least $1/n$. A map with more than dn edges must be faulty. Note that the candidate homing sequence is improved at most $n-1$ times. Thus by Corollary 11, with overwhelming probability after $O(n^2)$ tests of maps with at least dn nodes, the candidate sequence h is a homing sequence. Overall, the algorithm has to learn $O(dn^3)$ edges, and therefore it executes $O(dn^3)$ tests. Thus the total number of steps spent testing is $O(d^2n^5)$.

After each test or verification, the algorithm executes a candidate homing sequence. Since there are $O(dn)$ edges in each map, candidate homing sequences are executed $O(dn^3)$ times. Each improvement of the candidate homing sequence extends its length by $|path|$, so the time spent executing homing sequences is $O(d^2n^6)$. Thus, the total running time of the algorithm is $O(d^2n^6)$. $\square$

5.1 Improvements to the Algorithm

The running time for **Learn-Graph** can be decreased significantly by using two-robot *adaptive* homing sequences. As in Rivest and Schapire [RS93], an *adaptive homing sequence* is a decision tree, so the actions in later steps of the sequence depend on the output of earlier steps. With an adaptive homing sequence, only one map_c needs to be discarded each time the homing sequence is improved. Thus the running time of **Learn-Graph** decreases by a factor of n to $O(d^2n^5)$.

Any additional information that distinguishes nodes can be included in the output, so homing sequences can be shortened even more. For example, a robot learning an unfamiliar city could easily count the number of roads leading into and out of intersections. It might also recognize stop signs, traffic lights, railroad tracks, or other common landmarks. Therefore, in any practical application of this algorithm we expect a significantly lower running time than the $O(d^2n^5)$ bound suggests.

Graphs with high conductance can be learned even faster using the algorithm presented in Section 6.

5.2 Limitations of a Single Robot with Pebbles

We now compare the computational power of two robots to that of one robot with a constant number of pebbles. Note that although **Learn-Graph** runs in time polynomial in n, the algorithm requires no prior knowledge of n. We argue here that a single robot with a constant number of pebbles cannot efficiently learn strongly-connected directed graphs without prior knowledge of n.

As a tool we introduce a family $\mathcal{C} = \cup_n \mathcal{C}_n$ of graphs called *combination locks.* [1] For a graph

[1] Graphs of this sort have been used in theoretical computer science for many years (see [Moo56], for example). More recently they have reemerged as tools to prove the hardness of learning problems. We are not sure who first coined the term "combination lock."

$C = (V, E)$ in $\mathcal{C}_n$ (the class of n-node combination locks), $V = \{u_0, u_1, \ldots, u_{n-1}\}$ and either $\langle u_i, 0, u_{i+1 \, mod \, n}\rangle$ and $\langle u_i, 1, u_0\rangle \in E$, or $\langle u_i, 1, u_{i+1 \, mod \, n}\rangle$ and $\langle u_i, 0, u_0\rangle \in E$, for all $i \leq n$ (see Figure 3a). In order for a robot to "pick a lock" in $\mathcal{C}_n$ — that is, to reach node u_{n-1} — it must follow the unique n-node simple path from u_0 to u_{n-1}. Thus any algorithm for a single robot with *no* pebbles can expect to take $\Theta(2^n)$ steps to pick a random combination lock in $\mathcal{C}_n$.

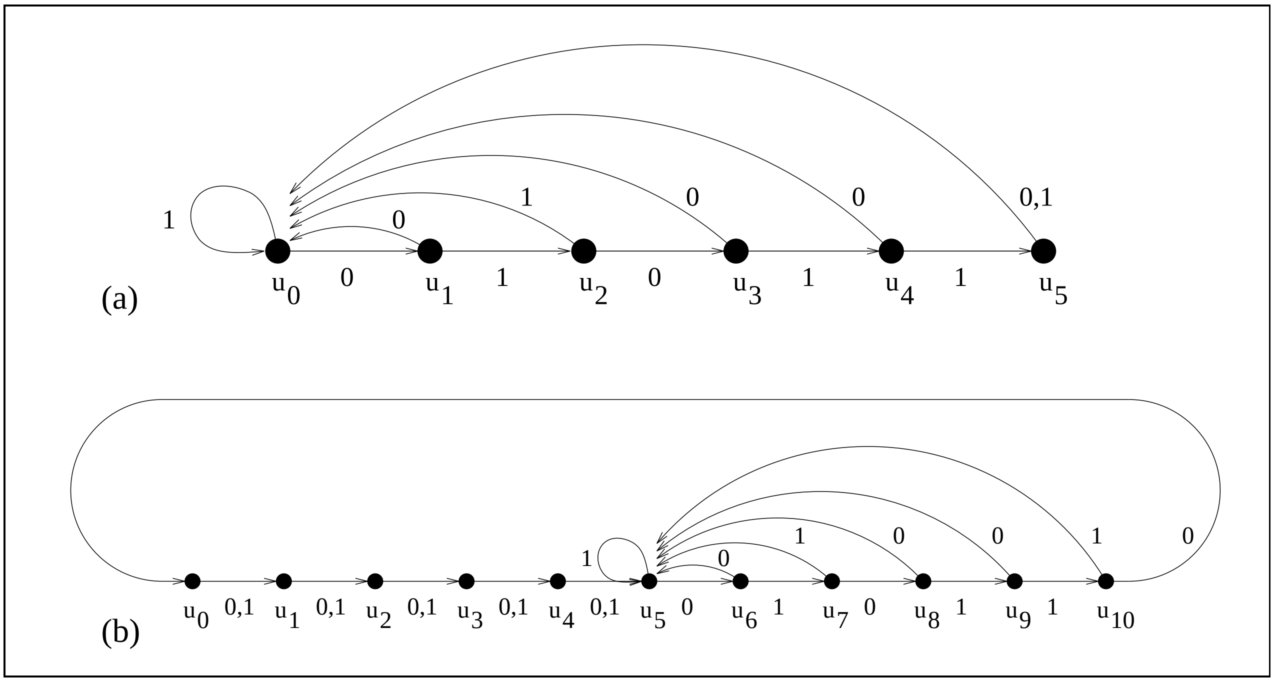


Figure 3: (a) A combination-lock, whose combination is $0, 1, 0, 1, 1$. (b) A graph in $\mathcal{R}_{11}$. Graphs in $\mathcal{R} = \cup_{n=1}^{\infty} \mathcal{R}_n$ cannot be learned by one robot with a constant number of pebbles.

We construct a restricted family $\mathcal{R}$ of graphs and consider algorithms for a single robot with a single pebble. For all positive integers n, the class $\mathcal{R}_n$ contains all graphs consisting of a directed ring of $n/2$ nodes with an $n/2$-node combination lock inserted into the ring (as in Figure 3b). Let $\mathcal{R} = \cup_{n=1}^{\infty} \mathcal{R}_n$. We claim that there is no probabilistic algorithm for one robot and one pebble that learns arbitrary graphs in $\mathcal{R}$ in polynomial time with high probability.

To see the claim, consider a single robot in node u_0 of a random graph in $\mathcal{R}$. Until the robot drops its pebble for the first time it has no information about the graph. Furthermore, with high probability the robot needs to take $\Theta(2^n)$ steps to emerge from a randomly-chosen n-node combination lock unless it drops a pebble in the lock. But since the size of the graph is unknown, the robot always risks dropping the pebble before entering the lock. If the pebble is dropped outside the lock, the robot will not see the pebble again until it has passed through the lock. A robot that cannot find its pebble has no way of marking nodes and cannot learn.

More formally, suppose that there were some probabilistic algorithm for one robot and a pebble to learn random graphs in $\mathcal{R}$ in polynomial time with probability greater than $1/2$. Then there must be some constant c such that the probability that the robot drops its pebble in its first c steps is greater than 1/2. (Otherwise, the probability that the algorithm *fails* to learn in time polynomial in n is greater than 1/2.) Therefore, the probability that the robot loses its pebble and fails to learn a random graph in $\mathcal{R}_{2c}$ efficiently is at least 1/2.

A similar argument holds for a robot with a constant number of pebbles. We conjecture that even if the algorithm is given n as input, a single robot with a constant number of pebbles cannot learn strongly-connected directed graphs. However, using techniques similar to those in Section 6, one robot with a constant number of pebbles and prior knowledge of n *can* learn high-conductance graphs in polynomial time with high probability.

6 Learning High Conductance Graphs

For graphs with good expansion, learning by walking randomly is more efficient than learning by using homing sequences. In this section we define *conductance* and present an algorithm that runs more quickly than **Learn-Graph** for graphs with conductance greater than $\sqrt{\log n/dn^2}$.

6.1 Conductance

The conductance [SJ89] of a graph characterizes the rate at which a random walk on the graph converges to the stationary distribution π. For a given directed graph $G = (V, E)$, consider a weighted graph $G' = (V, E, W)$ with the same vertices and edges as G, but with edge weights defined as follows. Let $M = \{m_{i,j}\}$ be the transition matrix of a random walk that leaves i by each outgoing edge with probability $1/(2 \cdot \mathrm{degree}(i))$ and remains at node i with probability 1/2. Let P^0 be an initial distribution on the n nodes in G, and let $P^t = P^0 M^t$ be the distribution after t steps of the walk defined by M. (Note that π is a *steady state* distribution if for every node $i, P_i^t = \pi_i \longrightarrow P_i^{t+1} = \pi_i$. For irreducible and aperiodic Markov chains, π exists and is unique.) Then the edge weight $w_{i,j} = \pi_i m_{i,j}$ is proportional to the steady state probability of traversing the edge from i to j. Note that the total weight entering a node is equal to the total weight leaving it; that is, $\sum_j w_{i,j} = \sum_j w_{j,i}$.

Consider a set $S \subseteq V$ which defines a cut $(S, \overline{S})$. For sets of nodes S and T, let $W_{S,T} = \sum_{s\in S, t\in T} w_{s,t}$. We denote $W_{S,V}$ by W_S, so W_V represents the total weight of all edges in the graph. Then the conductance of S is defined as $\phi_S = W_{S,\overline{S}} / \sum_{i \in S} \pi_i = W_{S,\overline{S}}/W_S$.

The conductance of a graph is the least conductance over all cuts whose total weight is at most $W_V/2$: $\phi(G) = \min_S \{\max(\phi_S, \phi_{\overline{S}})\}$. The conductance of a directed graph can be exponentially small.

Mihail [Mih89] shows that after a walk of length $\phi^{-2}\log(2n/\epsilon^2)$, the L_1 norm of the distance between the current distribution P and the stationary distribution π is at most ϵ (i.e. $\sum_i |P_i - \pi_i| \leq \epsilon$). In the rest of this section, a choice of $\epsilon = 1/n^2$ is sufficient, so a random walk of length $\phi^{-2}\log(2n^5)$ is used to approximate the stationary distribution. We call $T = \phi^{-2}\log(2n^5)$ the *approximate mixing time* of a random walk on an n-node graph with conductance ϕ.

6.2 An Algorithm for High Conductance Graphs

If a graph has high conductance it can be learned more quickly. In a high-conductance graph, we can estimate the steady state probability of node i by leaving Clark at node i while Lewis takes w random walks of $\phi^{-2}\log(2n^5)$ steps each. Let x be the number of times that Lewis sees Clark at the last step of a walk. If w is large enough, x/w is a good approximation to π_i.

Definitions: Call a node i a *likely* node if $\pi_i \geq 1/2n + 1/n^2$. Note that every graph must have at least one such node. (The $1/n^2$ term appears because of the distance $\epsilon = 1/n^2$ from the stationary distribution. Its inclusion here simplifies the analysis later.) A node that is not likely is called *unlikely*.

Algorithm **Learn-Graph2** uses this estimation technique to find a likely node u_0 and then calls the procedure **Build-Map** to construct a map of G starting from u_0. The procedure **Build-Map** learns at least one new edge each iteration by sending Lewis across an unexplored edge $\langle u, \ell, v\rangle$ of some unfinished node u in *map*. Clark waits at start node u_0 while Lewis walks randomly until he meets Clark. (If u_0 is a likely node, this walk is expected to take $O(Tn)$ steps.) Lewis stores this random walk in the variable *path*. Thus, $path_i$ is the label of the edge traversed at the ith step of the random walk, $path[i \ldots j]$ represents edges $path_i$ to $path_j$, and $|path|$ represents the length of path. We say that *path-step*(Lewis) $= i$ if Lewis has just crossed the ith edge on *path*.

```
Learn-Graph2(w, B, M, T):
done := FALSE
T := φ^{-2} log (2n^5)                                      { the mixing time }
while (not (done))
    do map := ({u_0}, ∅)              { map is the graph consisting of node u_0 and no edges }
        lost := FALSE
        Lewis and Clark together take a random walk of length T
        Lewis takes w random walks of length T     { approx. stationary prob. of Loc_G(Clark) }
        x := number of walks where Lewis and Clark are together at the last step
        path := the path Lewis followed since leaving Clark
        if x/w ≤ B                               { bound B < 1/n chosen for ease of proof }
            then Clark follows path to catch up to Lewis     { not at a frequently-visited node }
            else Lewis moves randomly until he sees Clark      { call node where they meet u_0 }
                done := Build-Map(map,M, T)
return map
```

The procedure **Compress-Path** returns the shortest subpath of *path* that connects v to u_0. Finally, **Truncate-Path-at-Map** compares nodes on the path with all nodes in *map* and returns the shortest subpath connecting v to some node in *map*. By adding the final path to the map, **Build-Map** connects the new node v to *map*, so *map* always represents a strongly connected subgraph of G. Figure 4 illustrates a single iteration of the main loop in **Build-Map**.

Algorithm **Learn-Graph2** takes as input parameters the number of random walks w, a bound B to separate the probability of likely and unlikely nodes (we choose B to be approximately $3/4n$), the mixing time T, and a quantity M. This quantity is chosen so that the probability of a robot's

Build-Map(map, M, T):

while there are unfinished nodes in map **and not** $lost$ { Lewis and Clark both at u_0 }
do u_i := an unfinished node in map
$restart$:= a minimal path in map from u_0 to u_i
m := largest index of the nodes in map
$path$:= empty path
Lewis follows $restart$ and traverses unexplored edge ℓ { Lewis crosses a new edge }
while $length(path) < MT$ **and** robots are not together { Lewis walks randomly ...}
do Lewis traverses a random edge ℓ' and adds ℓ' to end of $path$
if robots are together { ...until he sees Clark at u_0 }
then both robots follow $restart$ to u_i and cross edge ℓ
$path$:= **Compress-Path**($path$, $restart$) { removes loops from $path$ }
$path$, u_j := **Truncate-Path-At-Map**($path$, map) { shortest path back to map }
if $|path| = 0$
then add edge $\langle u_i, \ell, u_j \rangle$ to map
else add nodes $u_{m+1}, \ldots, u_{m+|path|}$ to map
add edges $\langle u_i, \ell, u_{m+1} \rangle$ and $\langle u_{m+|path|}, path_{|path|}, u_j \rangle$ to map
$\forall k, 1 \leq k < |path|$ add edges $\langle u_{m+k}, path_k, u_{m+k+1} \rangle$
both robots move to u_0
else { if Lewis walks MT steps without seeing Clark }
Clark follows $restart$, ℓ, $path$ to catch up to Lewis
$lost$:= TRUE
if $lost$
then return FALSE
else return TRUE

Compress-Path ($path$, $restart$):

while Clark not at end of path { Lewis and Clark both at u_0 }
do while Lewis not at end of path
do Lewis traverses the next edge of $path$
if Lewis and Clark are together { found a loop in $path$ — remove it }
then $path$:= $path[1 \ldots path\text{-}step(\text{Clark})] \circ path[path\text{-}step(\text{Lewis}) + 1 \ldots |path|]$
Lewis follows $restart$ and edge ℓ
Lewis traverses edges $path[1 \ldots path\text{-}step(\text{Clark})]$
both robots are now together and traverse one edge of path
return $path$ { Lewis and Clark both at u_0 }

Truncate-Path-At-Map($path$, map):

$earliest$:= $|path|$ { first position on $path$ that is a node already in map }
$earliest\text{-}node$:= u_0 { the name of this node }
for each node u_k in map
Clark moves to u_k
Lewis follows $restart$ and edge ℓ
while Lewis not at end of $path$
do if Lewis and Clark are together **and** $path\text{-}step(\text{Lewis}) < earliest$
then $earliest$:= $path\text{-}step(\text{Lewis})$
$earliest\text{-}node$:= u_k
Lewis traverses next edge on $path$
both robots move to u_0
return $path$ $[1 \ldots earliest]$, $earliest\text{-}node$

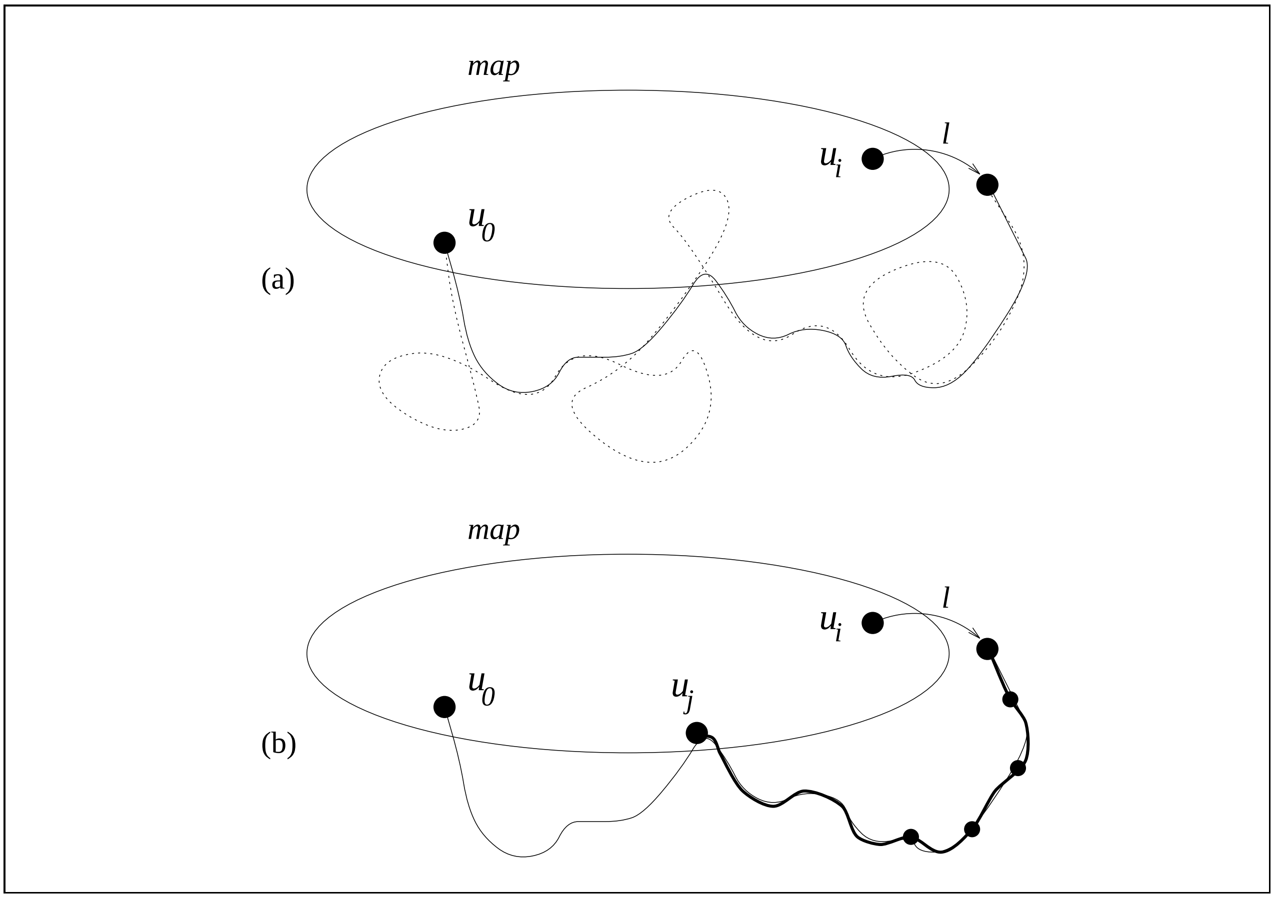


Figure 4: Procedure **Build Map** during one execution of the **while** loop. The ovals represent *map*, the portion of graph G learned so far. Note that *map* is strongly connected. Node u_0, the first node added to *map*, is with high probability a node with a large stationary probability (a *likely node*). The robots find u_0 in procedure **Learn-Graph2** using random walks in line 2 of **Build Map**. The robots agree on a node u_i with unexplored outgoing edges (an *unfinished node*). Then Lewis moves to u_i and follows the unexplored edge ℓ, while Clark stays at u_0. Since ℓ is unexplored, Lewis is now at an unknown node. Lewis walks randomly until, visiting u_0, he finds Clark. The dotted line of Figure 4 [a] depicts this random walk, denoted *path*. Random walk *path* may pass through the same node many times. In procedure **Compress-Path**, the robots collectively remove all of the loops from the path (reducing the path to the solid line in [a] and [b]). In procedure **Truncate-Path-at-Map**, the robots find u_j, the first node in *path* already in *map*. All the nodes and edges of *path* until u_j (the bold line in [b]) are added to *map*.

starting at a likely node and walking randomly for MT steps without returning to its start node is very small.

In sections 6.3 and 6.4 we prove the following theorems.

Theorem 13 *When* **Learn-Graph2** *halts, it outputs a graph isomorphic to* G*.*

Theorem 14 *Suppose* **Learn-Graph2** *is run on a graph* G *with* $w = \sqrt{(4+\tau)dn^3}$ *and* $M = (4+\tau)dn^2$ *for some constant* $\tau > 0$*, and* $B = \frac{3}{4n}\left(1+\frac{2}{n}\right)$*. Then for sufficiently large* n*, with probability at least* $1-\epsilon$ **Learn-Graph2** *halts within* $O((4+\tau)dn^3T)$ *steps, where* $\epsilon = e^{-\frac{1}{20}\sqrt{(4+\tau)nd}} + e^{-\frac{1}{2}\sqrt{(4+\tau)nd}} + e^{-\frac{dn\tau}{4}}$*.*

6.3 Correctness of Learn-Graph2

In this section, we prove the correctness of each procedure in **Learn-Graph2**.

Lemma 15 *The procedure* **Compress-Path** *halts in* $O(n\,|\mathrm{path}|)$ *steps and returns a path in which no node occurs more than once.*

Proof: We prove the following invariant by induction: in **Compress-Path**, whenever Lewis reaches the end of *path*, each node in *path*[1 ... *path-step*(Clark)] appears at most once in the entire *path*.

Assume that this claim holds after Clark has crossed the first k edges in *path*. By the inductive hypothesis, we know that when Clark crosses the $(k+1)$st edge, he arrives at some new node not previously encountered along *path*. Now Lewis follows the entire path. Whenever the path loops back to $\mathrm{Loc}_G(\mathrm{Clark})$, the loop is removed from the path. Thus, all repeated occurrences of the new node are removed from the path, proving the inductive step.

Since there are n nodes in the graph, Clark can only make n moves before he returns to u_0. Lewis can move at most $|path|$ steps for every move of Clark's, so the total running time is $O(n\,|path|)$. □

Lemma 16 *The procedure* **Truncate-Path-At-Map** *finds the index of the first path step leading to a node already in* map*. The algorithm runs in* $O(n^2)$ *steps.*

Proof: For each node u_k in *map*, Lewis traverses the path once while Clark waits at u_k. The procedure keeps track of the earliest node found that is already in *map*, so the procedure's correctness follows. Clark takes at most n steps to reach each node u_k. Lewis needs at most n steps to follow the compressed *path* and n more to return to the start of the path. Thus the algorithm requires no more than $3n^2$ steps. □

The algorithm **Learn-Graph2** halts only when **Build-Map** returns TRUE. The following lemma shows that whenever **Build-Map** returns TRUE, *map* is isomorphic to G.

Lemma 17 *In* **Build-Map***, whenever Clark is at* u_0*,* map *is a good representation of* $\langle Loc_G(Clark), G\rangle$*.*

Proof: We inductively build a subgraph $G' = (V', E')$ of $G = (V, E)$ and construct an isomorphism f from *map* to G'. Initially, Lewis and Clark are both at u_0 and *map* consists of the single node u_0 and no edges. Let $V' = Loc_G(\text{Clark}), E' = \emptyset$, and $f(u_0) = Loc_G(\text{Clark})$. Then *map* is a good representation of $\langle Loc_G(\text{Clark}),\text{G}\rangle$.

Consider the robots starting an iteration of the first **while** loop in **Build-Map**. Both robots are together at u_0. Inductively assume that *map* is a good representation of $\langle Loc_G(\text{Clark}),\text{G}\rangle$ and that *map* is strongly-connected. Thus, Lewis can always reach an unfinished node in *map* if one exists. Lewis walks to an unfinished node u_i, crosses a new edge ℓ to an unknown node v, and then walks randomly until he returns to u_0, where Clark is waiting. From Lemmas 15 and 16, after the algorithm executes subroutines **Compress-Path** and **Truncate-Path**, $\ell \circ path$ is a path that begins at u_i, crosses edge ℓ, ends at u_j, and whose intermediate nodes are not represented in *map*.

If *path* is empty after **Truncate-Path-At-Map**, then v is node u_j already in *map*. Adding edge $\langle f(u_i), \ell, f(u_j)\rangle$ to E' and $\langle u_i, \ell, u_j\rangle$ to *map* and therefore maintains the invariant that G' is a subgraph of G and preserves the isomorphism between *map* and G'.

If *path* is not empty, then by lemmas 15 and 16 all nodes reached by starting at u_i and following *path* to the end are distinct, and only the last node reached is already in *map*. Let m be the highest index so far of any node in *map*. The algorithm adds new nodes $u_{m+1}, \ldots, u_{m+|path|}$ and new edges $\langle u_{m+k}, path_k, u_{m+k+1}\rangle$ $\forall k, 1 \leq k < |path|$, $\langle u_i, \ell, u_{m+1}\rangle$, and $\langle u_{m+|path|}, path_{|path|}, u_j\rangle$ to *map*. Let $f(u_{m+k})$ be the location of Lewis in G after Lewis has crossed the kth edge of the path. Add the $|path| - 1$ new nodes to V', and edges $\langle f(u_{m+k}), path_k, f(u_{m+k+1})\rangle$ to E'. Then f is an isomorphism from *map* to $G' \subseteq G$, so *map* is a good representation of $\langle Loc_G(\text{Clark}),\text{G}\rangle$. Since *path* connects an unfinished node to another node in *map*, *map* remains strongly-connected. □

When **Build-Map** halts and returns TRUE, there are no unfinished nodes in *map* . Since *map* is isomorphic to $G' \subseteq G$ and has no unfinished nodes, *map* must have the same number of nodes and edges as G. Therefore, *map* is isomorphic to G when **Build-Map** returns TRUE, proving Theorem 13. □

6.4 Running Time and Failure Probability of Learn-Graph2

We proved that when the algorithm terminates it is correct. In this section, we prove Theorem 14 by analyzing the probability that the algorithm terminates in a reasonable amount of time. We say the algorithm fails if any of the following cases holds:

1. Algorithm **Learn-Graph2** finds an unlikely node but estimates that it is a likely node.
2. Algorithm **Learn-Graph2** fails to find a likely node in the allotted time. We allow $w = \sqrt{(4+\tau)dn^3}$ iterations, each consisting of w random walks.
3. Algorithm **Learn-Graph2** calls **Build-Map** from a likely node, but **Build-Map** returns FALSE.

In fact, these conditions overestimate the probability that the algorithm fails to run in $O((4 + \tau)dn^3T)$ steps. The next three lemmas bound the probabilities of each of the three failure conditions.

At the end of the section, we analyze the running time of **Learn-Graph2** when no failure condition occurs.

Lemma 18 (Failure Condition 1) *Suppose that* **Learn-Graph2** *is run with* $w = \sqrt{(4+\tau)dn^3}$ *and* $M = (4+\tau)dn^2$. *Assume that the algorithm estimates node u's steady-state probability to be greater than* $B = \frac{3}{4n}(1+\frac{2}{n})$. *Then the probability that u is not a likely node is at most* $e^{-\frac{1}{20}\sqrt{(4+\tau)nd}}$.

Proof: Call each random walk in **Learn-Graph2** a *phase*. Let X_i be the random variable where

$$X_i = \begin{cases} 1 & \text{if Lewis and Clark are together at the end of phase } i \\ 0 & \text{otherwise.} \end{cases}$$

Then $X = \sum_{i=1}^{w} X_i$ is the number of phases where Lewis and Clark end together. If u is an unlikely node, then $E[X] \leq (w/2n)(1+(2/n))$, because the estimation of π_u could be inaccurate by at most $\epsilon = 1/n^2$. We therefore bound the quantity

$$Pr\left[X > \frac{3w}{4n}\left(1+\frac{2}{n}\right) \;\middle|\; E[X] \leq \frac{w}{2n}\left(1+\frac{2}{n}\right)\right].$$

Using the Chernoff bound from Lemma 11 with $\delta = 1/2$, we get:

$$Pr\left[X \geq \frac{3w}{4n}\left(1+\frac{2}{n}\right)\right] \leq \left[\frac{e^{\frac{1}{2}}}{(\frac{3}{2})^{\frac{3}{2}}}\right]^{\frac{w}{2n}\left(1+\frac{2}{n}\right)}$$

$$\leq \left(\sqrt{\frac{8e}{27}}\right)^{\frac{w}{2n}} \leq e^{\ln\left(\sqrt{\frac{8e}{27}}\right)^{\frac{\sqrt{(4+\tau)dn}}{2}}} \leq e^{\frac{-\sqrt{(4+\tau)dn}}{2}\ln\left(\sqrt{\frac{27}{8e}}\right)} \leq e^{\frac{-\sqrt{(4+\tau)dn}}{20}}.$$

$\square$

Lemma 19 (Failure Condition 2) *The probability that* **Learn-Graph2** *fails to find and recognize a likely node after* $\sqrt{(4+\tau)dn^3}$ *iterations is at most* $e^{-\frac{1}{2}\sqrt{(4+\tau)dn}}$ *for sufficiently large n.*

Proof: Define a *good* node to be a node with steady state probability at least $1/n$. (Note that every graph has at least one good node.) We can bound the probability that we fail to find and recognize a likely node by the probability that we fail to identify a good node within w iterations.

First, we bound the probability that the algorithm fails to recognize a good node when testing one. Random variables X_i and X are defined as in Lemma 18. Then, since the stationary probability of a good node is greater than $1/n$,

$$E(X) \geq w\left(\frac{1}{n} - \frac{1}{n^2}\right).$$

To simplify the math, note that for sufficiently large n,

$$w\left(\frac{1}{n} - \frac{1}{n^2}\right) \geq \frac{15w}{16n}\left(1+\frac{2}{n}\right).$$

Then by the Chernoff corollary in Lemma 11, for $\delta = 1/5$,

$$Pr\left[X < (1-\delta)\frac{15w}{16n}\right] = Pr\left[X < \frac{3w}{4n}\right] \le e^{-\frac{1}{2}\frac{15}{16}\frac{1}{25}\frac{w}{n}} \le e^{-\frac{3}{160}\sqrt{(4+\tau)dn}}.$$

Define γ to be the quantity

$$\left(1-\frac{1}{n}\right)\left(1-e^{-\frac{3}{160}\sqrt{(4+\tau)dn^3}}\right).$$

The probability that the node reached at the end of a random walk of both robots is a good node and is identified as such is at least

$$\left(\frac{1}{n}-\frac{1}{n^2}\right)\left(1-e^{-\frac{3}{160}\sqrt{(4+\tau)dn^3}}\right) = \frac{\gamma}{n}.$$

Therefore the probability that after w trials, no likely node is found and recognized is at most

$$\left(1-\frac{\gamma}{n}\right)^{\sqrt{(4+\tau)dn^3}} = \left(1-\frac{\gamma}{n}\right)^{\frac{-n}{\gamma}\left(-\gamma\sqrt{(4+\tau)dn}\right)} = e^{-\gamma\sqrt{(4+\tau)dn}}.$$

Note that γ approaches 1 as n increases. For sufficiently large $n, \gamma > 1/2$, so the probability that the robots fail to find a likely node after w trials is at most $e^{-\frac{1}{2}\sqrt{(4+\tau)dn}}$. $\square$

Now we analyze the running time of **Build-Map**. The procedure **Build-Map** executes the main while loop at most once for each of the dn edges in the graph. Let k_i be the length of the random walk in the ith iteration of the while loop. Then let $K = \sum_{i=1}^{dn} k_i$ be the total length of all the random walks in the algorithm. Since by Lemmas 15 and 16 **Compress-Path** runs in $O(nk_i)$ steps and **Truncate-Path-At-Map** runs in $O(n^2)$ steps, the ith iteration takes $O(n+k_i+nk_i+n^2)$ steps. Therefore the total running time of the algorithm is $O(dn^3 + nK)$.

Lemma 20 (Failure Condition 3) *If u_0 is a likely node, then with probability at least $1-e^{-\frac{\tau dn}{4}}$, $K \le (4+\tau)dn^2T$.*

Proof: We use an amortized analysis to prove the bound on K. First we subdivide all of Lewis' random walks during **Build-Map** into periods of $T = \phi^{-2}\log 2n/\epsilon^2$ steps each, where T is the approximate mixing time. Recall that if Lewis starts from any node, after T steps Lewis is at node u_k with probability between $\pi_k - 1/n^2$ and $\pi_k + 1/n^2$. Thus, Lewis' position after the kth period is almost independent of Lewis' position after the $(k+1)$st period.

We associate a 0/1-valued random variable, X_k, with the kth period of Lewis' random walk.

$$X_k = \begin{cases} 1 & \text{if Lewis is at node } u_0 \text{ at the end of the } k\text{th period} \\ 0 & \text{otherwise.} \end{cases}$$

Since u_0 is a likely node, $X_k = 1$ with probability at least $1/2n$. Let $X = \sum_{k=1}^{(4+\tau)dn^2} X_k$ be the number of times Lewis returns to u_0 in $(4+\tau)dn^2$ periods. Note that $E(X)$ is at least $(4+\tau)dn/2$.

Using Chernoff bounds with $\delta = 1-(2/(4+\tau))$ we find:

$$Pr[X < (1-\delta)E[X]] < Pr[X < dn] < e^{\frac{-(4+\tau)dn}{4}\left(1-\frac{2}{4+\tau}\right)^2}$$

$$< e^{\frac{-(4+\tau)dn}{4}\left(1-\frac{4}{4+\tau}+\frac{4}{(4+\tau)^2}\right)} < e^{\frac{-(4+\tau)dn}{4}+dn} < e^{\frac{-dn\tau}{4}}.$$

□

Thus, with high probability **Build-Map** runs in $O(dn^3 + dn^3(4+\tau)T)$ steps. Suppose none of the failure conditions occurs in a run of **Learn-Graph2**. Then the execution never calls **Build-Map** on an unlikely node, *does* call **Build-Map** on a likely node, and when it does, **Build-Map** returns TRUE. Therefore **Learn-Graph2** makes at most w steady-state probability estimates, each taking $O(wT)$ steps, before calling **Build-Map** once. Therefore, the running time is $O((4+\tau)dn^3T)$, proving Theorem 14. □

6.5 Exploring Without Prior Knowledge

Prior knowledge of n is used in two ways in **Learn-Graph2**: to estimate the stationary probability of a node and to compute the mixing time T. If T is known but n is not, the algorithm can forego estimating π_i entirely and simply run **Build-Map** after step 6. The removal of lines $7-12$ from **Learn-Graph2** yields a new algorithm whose expected running time is polynomially slower than the original. If we know neither n nor T, we can run this new algorithm using standard doubling to estimate the quantity MT. This quantity can be used in line 6 of **Learn-Graph2** and also in line 7 of **Build-Map** as an upper bound on the length of the random walks. Thus no prior knowledge of the graph is necessary.

7 Conclusions and Open Problems

Note that with high probability, a single robot with a pebble can simulate algorithm **Learn-Graph2** with a substantial but polynomial slowdown. However, **Learn-Graph2** does not run in polynomial expected time on graphs with exponentially-small conductance. An open problem is to establish tight bounds on the running time of an algorithm that uses one robot and a constant number of pebbles to learn an n-node graph G. We conjecture that the lower bound will be a function of $\phi(G)$, but there may be other graph characteristics (*e.g.*, cover time) which yield better bounds. It would also be interesting to establish tight bounds on the number of pebbles a single robot needs to learn graphs in polynomial time.

Another direction for future work is to find other special classes of graphs that two robots can learn substantially more quickly than general directed graphs, and to find efficient algorithms for these cases.

Acknowledgements

We are particularly grateful to Ron Rivest, who patiently listened to many half-baked ideas, helped us work through some of the rough spots, and suggested names for the robots. We gratefully thank Michael Rabin, Mike Sipser and Les Valiant for many enjoyable and helpful discussions. We also thank Rob Schapire for his insightful comments on an earlier draft of the paper.

References

[Ang81] Dana Angluin. A note on the number of queries needed to identify regular languages. *Information and Control*, 51:76–87, 1981.

[Ang87] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, November 1987.

[AR94] Y. Aumann and M. O. Rabin. Clock construction in fully asynchronous parallel systems and PRAM simulation. *Theoretical Computer Science*, 128:3–30, 1994.

[ABRS95] Baruch Awerbuch, Margrit Betke, Ronald L.Rivest, and Mona Singh. Piecemeal graph exploration by a mobile robot. In *Proceedings of the 1995 ACM Conference on Computational Learning Theory*, pages 321–328, Santa Cruz, CA, July 1995.

[BB+90] Paul Beame, Allan Borodin, Prabhakar Raghavan, Walter Ruzzo, and Martin Tompa. Time-space tradeoffs for undirected graph traversal. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, pages 429–430, 1990.

[Bet92] Margrit Betke. Algorithms for exploring an unknown graph. Master's thesis. Published as MIT Laboratory for Computer Science *Technical Report MIT/LCS/TR-536*, March, 1992.

[BRS93] Margrit Betke, Ronald Rivest, and Mona Singh. Piecemeal learning of an unknown environment. In *Proceedings of the 1993 Conference on Computational Learning Theory*, pages 277–286, Santa Cruz, CA, July 1993. (Published as MIT AI-Memo 1474, CBCL-Memo 93; to be published in Machine Learning).

[BK78] Manuel Blum and Dexter Kozen. On the power of the compass (or, why mazes are easier to search than graphs). In *19th Annual Symposium on Foundations of Computer Science*, pages 132–142. IEEE, 1978.

[BS77] M. Blum and W. J. Sakoda. On the capability of finite automata in 2 and 3 dimensional space. In *18th Annual Symposium on Foundations of Computer Science*, pages 147–161. IEEE, 1977.

[CBF+93] N. Cesa-Bianchi, Y. Freund, D. Helmbold, D. Haussler, R. Schapire, and M. Warmuth. How to use expert advice. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, pages 382–391, San Diego, CA, May 1993.

[CR80] Stephen A. Cook and Charles W. Rackoff. Space lower bounds for maze threadability on restricted machines. *SIAM Journal on Computing*, 9:636–652, 1980.

[DA+92] Thomas Dean, Dana Angluin, Kenneth Basye, Sean Engelson, Leslie Kaelbling, Evangelos Kokkevis, and Oded Maron. Inferring finite automata with stochastic output functions and an application to map learning. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 208–214, July 1992.

[DP90] Xiaotie Deng and Christos H. Papadimitriou. Exploring an unknown graph. In *Proceedings of the 31st Symposium on Foundations of Computer Science*, volume I, pages 355–361, 1990.

[DP+91] Robert Daley, Leonard Pitt, Mahendran Velauthapillai, and Todd Will. Relations between probabilistic and team one-shot learners. In *Proceedings of COLT '91*, pages 228–239. Morgan Kaufmann, 1991.

[Edm93] Jeff Edmonds. Time-space trade-offs for undirected *st*-connectivity on a JAG. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, pages 718–727, San Diego, CA, May 1993.

[FK+93] Yoav Freund, Michael Kearns, Dana Ron, Ronitt Rubinfeld, Robert Schapire, and Linda Sellie. Efficient learning of typical finite automata from random walks. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, pages 315–324, San Diego, CA, May 1993.

[Koh78] Zvi Kohavi. *Switching and Finite Automata Theory.* McGraw-Hill, second edition, 1978.

[KS93] Michael J. Kearns and H. Sebastian Seung. Learning from a population of hypotheses. In *Proceedings of COLT '93*, pages 101–110, 1993.

[Mih89] Milena Mihail. Conductance and convergence of Markov chains – a combinatorial treatment of expanders. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 526–531, 1989.

[Moo56] Edward F. Moore. *Gedanken-Experiments on Sequential Machines*, pages 129–153. Princeton University Press, 1956. Edited by C. E. Shannon and J. McCarthy.

[Poo93] C.K. Poon. Space bounds for graph connectivity problems on node-named JAGs and node-ordered JAGs. In *Proceedings of the Thirty-Fourth Annual Symposium on Foundations of Computer Science*, pages 218–227, Palo Alto, California, November 1993.

[PW95] James Gary Propp and David Bruce Wilson. Exact sampling with coupled Markov chains and applications to statistical mechanics. To appear in the *Proceedings of the 1995 Conference on Random Structures and Algorithms.*

[Rab67] Michael O. Rabin. Maze threading automata. October 1967. Seminar talk presented at the University of California at Berkeley.

[RS87] Ronald L. Rivest and Robert E. Schapire. Diversity-based inference of finite automata. In *Proceedings of the Twenty-Eighth Annual Symposium on Foundations of Computer Science*, pages 78–87, Los Angeles, California, October 1987.

[Rag89] Prabhakar Raghavan. *CS661: Lecture Notes on Randomized Algorithms.* Yale University, Fall 1989.

[RS93] Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, April 1993.

[RR95] Dana Ron and Ronitt Rubinfeld. Learning fallible deterministic finite automata. *Machine Learning*, 18:149–185, February/March 1995.

[RR95b] Dana Ron and Ronitt Rubinfeld. Exactly learning automata with small cover time. In *Proceedings of the 1995 ACM Conference on Computational Learning Theory*, pages 427–436, Santa Cruz, CA, July 1995.

[Sav73] Walter J. Savitch. Maze recognizing automata and nondeterministic tape complexity. *JCSS*, 7:389–403, 1973.

[SJ89] Alistair Sinclair and Mark Jerrum. Approximate counting, uniform generation and rapidly mixing markov chains. *Information and Computation*, 82(1):93–133, July 1989.

[Smi94] Carl H. Smith. Three decades of team learning. To appear in *Proceedings of the Fourth International Workshop on on Algorithmic Learning Theory*. Springer Verlag, 1994.