Piecemeal Graph Exploration by a Mobile Robot *

Baruch Awerbuch
Department of Computer Science
Johns Hopkins University
Baltimore, MD 21218

Margrit Betke
Computer Science Department
Boston College
Chestnut Hill, MA 02167

Ronald L. Rivest
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Mona Singh
Whitehead Institute for Biomedical Research
Cambridge, MA 02142

*Most of this work was done while all authors were affiliated with the Laboratory of Computer Science at the Massachusetts Institute of Technology. We gratefully acknowledge support from NSF grant CCR-9310888, ARO grant DAAL03-86-K-0171, NSF grant 9217041-ASC, Air Force Contract TNDGAFOSR-86-0078, ARPA/Army contract DABT63-93-C-0038, NSF contract 9114440-CCR, DARPA contract N00014-J-92-1799, ARPA/ONR contract N00014-92-J-1310, the Siemens Corporation, and a special grant from IBM. The authors can be reached at baruch@blaze.cs.jhu.edu, betke@cs.bc.edu, rivest@theory.lcs.mit.edu, and mona@wi.mit.edu.

Abstract

We study how a mobile robot can learn an unknown environment in a piecemeal manner. The robot's goal is to learn a complete map of its environment, while satisfying the constraint that it must return every so often to its starting position (for refueling, say). The environment is modeled as an arbitrary, undirected graph, which is initially unknown to the robot. We assume that the robot can distinguish vertices and edges that it has already explored.

We present a surprisingly efficient algorithm for piecemeal learning an unknown undirected graph $G = (V, E)$ in which the robot explores every vertex and edge in the graph by traversing at most $O(E + V^{1+o(1)})$ edges. This nearly linear algorithm improves on the best previous algorithm, in which the robot traverses at most $O(E + V^2)$ edges.

We also give an application of piecemeal learning to the problem of searching a graph for a "treasure."

1 Introduction

We address the situation where a robot must explore an unknown environment. The robot's goal is to learn a complete map of this environment while satisfying the *piecemeal constraint* that the exploration must be done in phases of limited duration.

Why might mobile robot exploration be done piecemeal? Robots have limited power, and after some exploration they may need to recharge or refuel. In addition, robots are useful for exploring environments that are too risky or costly for humans to explore, such as the inside of a volcano (e.g., as attempted by CMU's Dante II robot), or a chemical waste site, or the surface of Mars. In these cases, the robot's hardware may be too expensive or fragile to stay long in dangerous conditions. Thus, it may be best to organize the learning into phases, allowing the robot to return to a start position for refueling and maintenance.

The piecemeal learning problem and the formal model used here were introduced by Betke, Rivest, and Singh [8]. The robot's environment is modeled as an unknown graph. The piecemeal constraint is a bound on the number of edges the robot is allowed to traverse in each exploration phase. In order to assure that the robot can reach any vertex in the graph and do some exploration, this bound must allow at least one round trip from the start vertex s to any vertex in the graph. The robot's efficiency (or running time) is measured in terms of the number of edges traversed. Betke, Rivest, and Singh [8] show that a robot can explore grid-graphs with rectangular obstacles in a piecemeal manner in linear time. In this paper, we extend these results to show that the robot can learn *any* undirected graph $G = (V, E)$ piecemeal in almost linear time. We first give a simple algorithm that runs in $O(E+V^{1.5})$ time. We then improve this algorithm and give an almost linear time algorithm that achieves $O(E + V^{1+o(1)})$ running time. The most efficient previously known algorithm has $O(E + V^2)$ running time. It is open whether arbitrary, undirected graphs can be learned piecemeal in linear time.

The piecemeal constraint is most naturally satisfied by requiring the robot to explore in a near breadth-first manner, so that it is never much further away from the start vertex s than necessary to visit any unexplored vertex. In this manner, returns to s are efficient. Breadth-first search (BFS) on unknown graphs is also an important problem in its own right, with many applications. We consider one such application, *treasure hunting*, where the goal is to find a treasure (or a lost child, or a particular landmark) that is believed to be near s. If the robot knows that the treasure is close to its current location, it should explore in a breadth-first manner from its current location.

BFS is a classic technique for searching graphs [18, 17, 11]. However, standard BFS is efficient for exploring unknown graphs only when the robot can efficiently switch or "teleport" from expanding one vertex to expanding another. In contrast, our model assumes a more natural scenario where the robot must *physically* move from one vertex to the next. In this case, if the robot exactly satisfies the traditional BFS constraint (i.e., it cannot move further away from s than the unvisited vertex nearest to s), then it may traverse up to $O(E^2)$ edges. Thus, for efficiency reasons, in the more difficult *teleport-free* exploration model, our algorithms for the piecemeal learning problem give *approximate* BFS algorithms where the robot does not move much further away from s than the shortest path distance from s to the unvisited vertex nearest to s.

In the teleport-free BFS algorithms we first present, the robot never visits a vertex more than twice as far from s as the nearest unvisited vertex is from s. Our final teleport-free BFS algorithm, for the treasure hunting problem, satisfies the stronger condition that if the closest unvisited vertex to s is distance δ away, the robot is never more than $\delta + o(\delta)$ away from s. This algorithm is also efficient: if the treasure is at a vertex that has shortest path distance δ_T away from s, then the robot traverses at most $O(E + V^{1+o(1)})$ edges, where E and V are the number of edges and vertices within radius $\Delta = \delta_T + o(\delta_T)$ from s. Our final treasure hunting algorithm is also a solution to the piecemeal learning problem.

Related work

Many researchers have studied problems in environment learning and robot motion planning. Papadimitriou and Yanakakis [19] developed one of the first formal models for exploring unknown environments. They show how to find a shortest path in an unknown, undirected graph. Deng and Papadimitriou [13] and Betke [6] address the problem of learning an unknown directed graph. Bender and Slonim [5] show how two cooperating robots can learn a directed graph. Rivest and Schapire [21] model the robot's unknown environment by a deterministic finite automaton. They describe algorithms that efficiently infer the structure of the automaton through experimentation. Deng, Kameda, and Papadimitriou [12] consider how to learn the interior of a two-dimensional room. Blum, Raghavan, and Schieber [10] consider a robot navigating in an unknown two-dimensional geometric terrain with convex obstacles. Bar-Eli, Berman, Fiat, and Yan [4] give an efficient algorithm for reaching the center of a two-dimensional room with obstacles. Betke and Gurvits [7], Kleinberg [16], and Romanik and Schuierer [22] address the problem of localizing a mobile robot in

its environment. Blum and Chalasani [9] consider the problem of finding a "k-trip" shortest path in the environment. There are many other related papers in the literature (e.g., [15, 14, 20]).

Our techniques are inspired by the work of Awerbuch and Gallager [2, 3]. We observe that our learning model bears some similarity to the asynchronous distributed message passing model. This similarity is surprising and has not been explored in the past.

Model and Definitions

This section reviews the piecemeal exploration model introduced by Betke, Rivest, and Singh [8]. The robot's environment is modeled as a finite connected undirected graph $G = (V, E)$ with distinguished start vertex s. Vertices represent accessible locations. Edges represent accessibility: if $\{x, y\} \in E$ then the robot can move from x to y, or back, in a single step.

The robot can always recognize a previously visited vertex; it never confuses distinct locations. At any vertex the robot can sense only the edges incident to it; it has no vision or other long-range sensors. The robot can distinguish between incident edges at any vertex. Each edge has a label that distinguishes it from any other edge. Without loss of generality, we can assume that the edges are ordered. At a vertex, the robot knows which edges it has traversed already. The robot only incurs a cost for traversing edges; the time the robot spends "thinking and path planning" is free (although for all the algorithms in this paper, this planning can be done in time polynomial in the size of the graph). We also assume a uniform cost for an edge traversal. We measure the running time of a piecemeal exploration algorithm in terms of the number of edge traversals made by the robot.

The robot's goal in piecemeal exploration is to explore its entire unknown environment while satisfying the piecemeal constraint that it must return every so often to its starting point. The robot is given an upper bound B on the number of steps it can make (edges it can traverse) in one exploration phase. In order to assure that the robot can reach any vertex in the graph, do some exploration, and then get back to the start vertex, we assume B allows for at least one round trip between s and any other single vertex in G, and also allows for some number of exploration steps. More precisely, we assume $B = (2 + \alpha)r$, where $\alpha > 0$ is some constant, and r is the *radius* of the graph (i.e., the maximum of all shortest-path distances between s and any vertex in G). Note that our definition of the radius of the graph is relative to the start vertex s.

Initially all the robot knows is its starting vertex s, the bound B, and the radius r of the

graph. The robot's goal is to explore the entire graph: to visit every vertex and traverse every edge, minimizing the total number of edges traversed.

We say an exploration is *efficiently interruptible* if the robot always knows a path of explored edges of length at most r back to s. All the algorithms presented in this paper are efficiently interruptible, and, using the following theorem, give efficient piecemeal learning algorithms for undirected graphs.

Theorem 1 *An efficiently interruptible algorithm $\mathcal{A}$ for exploring an unknown graph $G = (V, E)$ with n vertices and m edges that takes time $T(n, m)$ can be transformed into a piecemeal learning algorithm that takes time $O(T(n, m))$.*

Proof: Assume that the radius of the graph is r and that the number of edges the robot is allowed to traverse in each phase of exploration is $B = (2 + \alpha)r$, for some constant α such that αr is a positive integer. In each exploration phase, the robot executes αr steps of the original search algorithm $\mathcal{A}$, interrupts its search, and returns to the start vertex s. At the beginning of the next phase, the robot returns from s to the appropriate vertex to resume exploration. Then, the robot traverses again αr edges as determined by the original search algorithm $\mathcal{A}$ and returns to s. Since the search algorithm $\mathcal{A}$ is efficiently interruptible, the robot knows a path of length at most r from s to any vertex in the graph. Thus during any exploration phase, the robot traverses at most $2r$ edges for relocation to s and back and αr edges for new exploration. The total number of edges traversed in each phase is at most $2r + \alpha r = B$. Since there are $\lceil \frac{T(n,m)}{\alpha r} \rceil$ segments, there are $\lceil \frac{T(n,m)}{\alpha r} \rceil - 1$ interruptions, and the number of edge traversals due to interruptions is:

$$\begin{aligned} \left(\left\lceil \frac{T(n,m)}{\alpha r} \right\rceil - 1 \right) 2r &\leq \frac{T(n,m)}{\alpha r} 2r \\ &\leq \frac{2T(n,m)}{\alpha}. \end{aligned}$$

Since α is a constant, the total number of edge traversals is still $O(T(n, m))$. □

BFS is an efficiently interruptible algorithm where the robot may not move further away from the source than the unvisited vertex nearest to the source. At any given time in the algorithm, let Δ denote the shortest-path distance from s to the vertex the robot is visiting, and let δ denote the shortest-path distance from s to the vertex nearest to s that is as yet unvisited. With traditional breadth-first search we have $\Delta \leq \delta$ at all times. With teleport-free exploration, it is generally

impossible to maintain $\Delta \leq \delta$ without a great loss of efficiency:

Lemma 1 *A robot that maintains $\Delta \leq \delta$ (such as one using a traditional BFS) may traverse $\Omega(E^2)$ edges.*

Proof: Consider a graph with vertices $\{-n, -n+1, \ldots, -1, 0, 1, 2, \ldots, n-1, n\}$, where $s = 0$ and edges connect consecutive integers. To achieve $\Delta \leq \delta$, a teleport-free BFS algorithm would run in quadratic time, traveling back and forth from 1 to -1 to -2 to 2 to 3 .... □

Given this lower bound, we solve the piecemeal learning problem and the treasure hunting problem efficiently while maintaining the *approximate BFS constraint* that the robot is never more than twice as far from s as is the nearest unvisited vertex from s (i.e., $\Delta \leq 2\delta$). Our final algorithm Treasure-Search satisfies the stronger condition $\Delta = \delta + o(\delta)$. Note that this algorithm is also efficiently interruptible and thus can also be used to solve the piecemeal learning problem.

In the remainder of this paper, we give three algorithms for piecemeal learning undirected graphs. In Section 2, we first give a simple algorithm that runs in $O(E + V^{1.5})$ time. In Section 3, we then give a modification of this algorithm that runs in $O((E + V^{1.5}) \log V)$ time. Although this algorithm has a slightly slower running time, we are able to make it recursive, and in Section 4, we describe this recursive algorithm which has a nearly linear running time: it achieves $O(E+V^{1+o(1)})$ running time. Finally, in Section 5, we give our algorithm for treasure hunting.

2 Algorithm Strip-Explore

This section describes an efficiently interruptible algorithm for undirected graphs with running time $O(E + V^{1.5})$. It is based on breadth-first search.

A *layer* in a BFS tree consists of vertices that have the same shortest path distance to the start vertex. A *frontier vertex* is a vertex that is incident to unexplored edges. A frontier vertex is *expanded* when the robot has traversed all the unexplored edges incident to it.

The traditional BFS algorithm expands frontier vertices layer by layer. In the teleport-free model, this algorithm runs in time $O(E+rV)$, since expanding all the vertices takes time $O(E)$, and visiting all the frontier vertices on layer i can be performed with a depth-first search of layers $1 \ldots i$ in time $O(V)$, and there are at most r layers. Since r can be $O(V)$, this can result in an $O(E + V^2)$ algorithm.

The procedure LOCAL-BFS describes a version of the traditional BFS procedure that has been modified for our teleport-free BFS model in two respects. First, when expanding vertices on layer i, the robot does not relocate to any vertices in that layer that no longer have any unexplored edges. Second, it only explores vertices within a given distance-bound L of the given start vertex s. (The first modification, while seemingly straightforward, is essential for our analysis of STRIP-EXPLORE which uses LOCAL-BFS as a subroutine.) A procedure call of the form LOCAL-BFS(s, r), where s is the start vertex of the graph and r is its radius, would cause the robot to explore the entire graph.

```
LOCAL-BFS(s, L)
1  For i = 0 To L − 1 Do
2      let verts = all vertices at shortest path distance i from s
3      For each u ∈verts Do
4          If u has any incident unexplored edges
5            Then
6                relocate to u
7                traverse each unexplored edge incident to u
8  relocate to s
```

Awerbuch and Gallager [2, 3] give a distributed BFS algorithm which partitions the network (i.e., graph) into *strips*, where each strip is a group of L consecutive layers. (Here L is a parameter to be chosen.) All vertices in strip $i-1$ are expanded before any vertices in strip i are expanded. Their algorithms use as a subroutine breadth-first type searches with distance L.

Our algorithm, STRIP-EXPLORE, searches in strips in a new way (see Figure 1). The robot explores the graph in strips of width L. First the robot follows LOCAL-BFS(s, L) to explore the first strip. It then explores the second strip as follows. Suppose there are k frontier vertices $v_1, v_2, \ldots, v_k$ in layer L; each such vertex is a *source vertex* for exploring the second strip. A naive way for exploring the second strip is for the robot for each i, to relocate to v_i, and then find all vertices that are within distance L of v_i by doing a BFS of distance-bound L from v_i within the second strip.

The robot thus traverses a forest of k BFS trees of depth L, completely exploring the second strip. The robot then has a map of the BFS tree of depth L for the first strip and a map of the BFS forest for the second strip, enabling it to create a BFS tree of depth $2L$ for the first two strips. The robot continues, strip by strip, until the entire graph is explored.

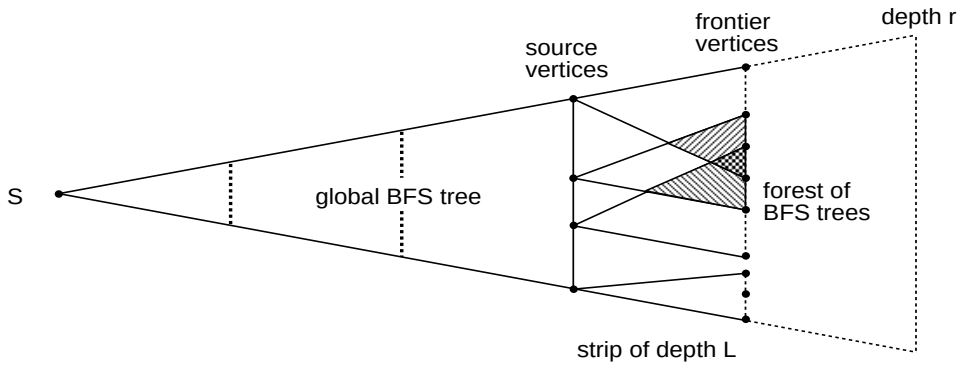

Figure 1: In STRIP-EXPLORE, the shaded areas are passed through more than once only if necessary to get to frontier vertices. In the naive algorithm, the shaded areas are retraversed completely.

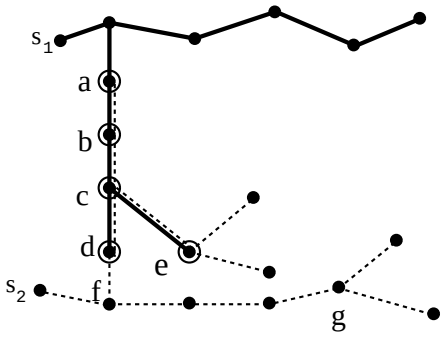

Figure 2: Contrasting BFS and Local-BFS: Consider a BFS of depth 5 from s_1, followed by a BFS of depth 5 from s_2. (The depth of the strip is $L = 5$.) The BFS from s_2 revisits vertices a, b, c, d, e. On the other hand, if the BFS from s_1 is followed by a LOCAL-BFS from s_2, then the LOCAL-BFS only revisits d, c, e. After edge (f, d) is found, vertex e is a frontier vertex that is expanded by LOCAL-BFS$(s_2, 5)$.

The naive algorithm described above is inefficient, due to the overlap between the trees in the forest at a given level, causing portions of each strip to be repeatedly re-explored (see Figure 2). The algorithm STRIP-EXPLORE presented below solves this problem by using the LOCAL-BFS procedure as the basic subroutine, instead of using a naive BFS.

In STRIP-EXPLORE, the robot searches in a breadth-first manner, but ignores previously explored territory. The only time the robot traverses edges that have been previously explored is when moving to a frontier vertex it is about to expand. This results in retraversal of some edges in previously explored territory, but not as many as in the naive algorithm.

```
STRIP-EXPLORE(s, L, r)
1  numstrips = ⌈r/L⌉
2  sources = {s}
3  For i = 1 To numstrips Do
4      For each u ∈ sources Do
5          relocate to u
6          LOCAL-BFS(u, L)
7      sources = all frontier vertices
```

Theorem 2 STRIP-EXPLORE *runs in* $O(E + V^{1.5})$ *time.*

Proof: First we count edge traversals for relocating between source vertices for a given strip. For these relocations, the robot can mentally construct a tree in the known graph connecting these vertices, and then move between source vertices by doing a depth-first traversal of this tree. Thus the number of edge traversals due to relocations between source vertices for this strip is at most $2V$. Since there are $\lceil r/L \rceil$ strips, the total number of edge traversals due to relocations between source vertices is at most $\lceil \frac{r}{L} \rceil 2V \leq (\frac{r}{L} + 1)\, 2V = \frac{2rV}{L} + 2V$.

Now we count edge traversals for repeatedly executing the LOCAL-BFS algorithm. First, for the robot to expand all vertices and explore all edges, it traverses $2E$ edges. Next, each time the relocate in line 8 of procedure LOCAL-BFS is called, at most L edges are traversed, thus resulting in at most LV edge traversals. To account for relocations in line 6 of procedure LOCAL-BFS, we use the following scheme for "charging" edge traversals. Say the robot is within a call of the LOCAL-BFS algorithm. It has just expanded a vertex u and will now relocate to a vertex v to expand it. Vertex v is charged for the edges traversed to relocate from u to v. (We are only considering relocations within the same call of the LOCAL-BFS algorithm; relocations between calls of the LOCAL-BFS

algorithm were considered above.) Source vertices are not charged anything. Moreover, the robot can always relocate from u to v by going from u to the source vertex of the current local BFS, and then to v, traversing at most $2L$ edges. Thus, each vertex is charged at most $2L$ when it is expanded. LOCAL-BFS never relocates to a vertex v unless it can expand vertex v (i.e., unless v is adjacent to unexplored edges). Thus, all relocations are charged to the expansion of some vertex, and the total number of edge traversals due to relocation is at most $2LV$.

Thus the total number of edge traversals is at most $2rV/L+2V+3LV+2E$, which is $O(rV/L+LV+E)$. When L is chosen to be $\sqrt{r}$, this gives $O(E+V^{1.5})$ edge traversals. □

Procedure STRIP-EXPLORE, and the generalizations of it given in later sections, maintain that $\Delta \leq 2\delta$ at all times—the robot never visits a vertex more than twice as far from s as the nearest unvisited vertex is from s. The worst case is while exploring the second strip.

3 Algorithm ITERATIVE-STRIP

We now describe ITERATIVE-STRIP, an algorithm similar to the STRIP-EXPLORE algorithm. It is an efficiently interruptible algorithm for undirected graphs inspired by Awerbuch and Gallager's [2] distributed iterative BFS algorithm. Although its running time of $O((V^{1.5}+E)\log V)$ is worse than the running time of STRIP-EXPLORE, its recursive version (described in Section 4) is more efficient. (It is not clear how to recursively implement STRIP-EXPLORE as efficiently, because the trees in a strip are not disjoint.)

With ITERATIVE-STRIP, the robot grows a global BFS tree with root s strip by strip, in a manner similar to STRIP-EXPLORE. Unlike STRIP-EXPLORE, here each strip is processed several times before it has correctly deepened the BFS tree by $\sqrt{r}$. We next explain the algorithm's behavior on a typical strip by describing how a strip is processed for the first time, and then for the remaining iterations.

In the first iteration, a strip is explored much as in STRIP-EXPLORE. The robot explores a tree of depth $\sqrt{r}$ from each source vertex, by exploring in breadth-first manner from each source vertex, without re-exploring previous trees. Whenever the robot finds a *collision* edge connecting the current tree to another tree in the same strip, it does not enter the other tree. Unlike STRIP-EXPLORE, the robot does not traverse explored edges to get to the frontier vertices on other trees. Therefore, after the first iteration, the trees explored are *approximate BFS trees* that may have

frontier vertices with path length less than $\sqrt{r}$ from some source vertex. We call these vertices *active frontier vertices* for the next iteration. A connected component within a strip is an *active connected component* if it contains active frontier vertices. After the first iteration, the current strip may not yet extend the global BFS tree by depth $\sqrt{r}$, so more iterations are needed until all frontier vertices are inactive and the global BFS tree is extended by depth $\sqrt{r}$ (see Figure 3).

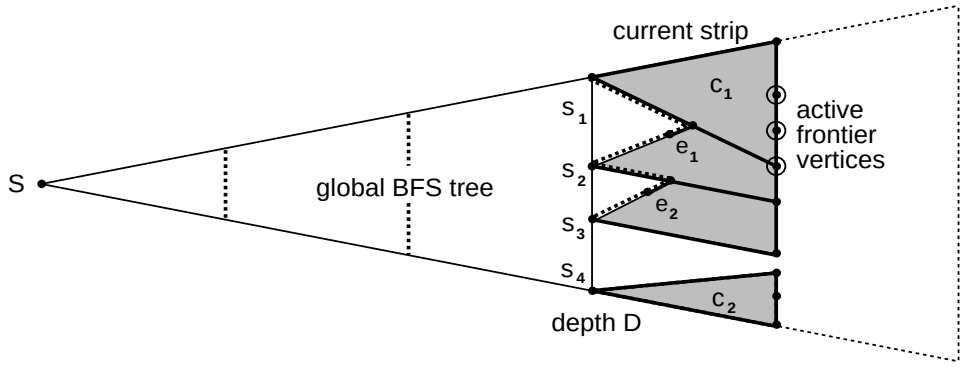

Figure 3: The iterative strip algorithm after the first iteration on the fourth strip. Two connected components c_1, c_2 have been explored. The collision edges e_1 and e_2 connect the first three approximate BFS trees. The dashed line shows how source vertices s_1, s_2, s_3 connect within the strip. There are three active frontier vertices with depth less than $D + \sqrt{r}$.

In the second iteration (see Figure 4), the robot uses the property that two trees connected by a collision edge form a connected component within the strip. (The graph to be explored is connected, and thus forms one connected component; but we refer to connected components of the explored portion of the graph contained within the strip.) The robot need not traverse any edges outside the current strip to relocate between these active frontier vertices in the same connected component. In the second and later iterations, the robot works on one connected component at a time.

The robot explores active frontier vertices in one connected component as follows. It computes (mentally) a spanning tree of the vertices in the current component. This spanning tree lies within the strip. Let d be the shortest known path length from any active frontier vertex in the component to any source vertex in the component. The robot visits the vertices in the strip in an order determined by a DFS of the spanning tree. As it visits active frontier vertices of depth d, it expands them. It then recomputes the spanning tree (since the component may now have new vertices) and again traverses the tree, expanding vertices of the appropriate next depth d'. Traversing a collision edge does not add the new vertex to the tree, since this vertex has been explored before. This process continues (at most $\sqrt{r}$ times) until no active frontier vertex in the connected component

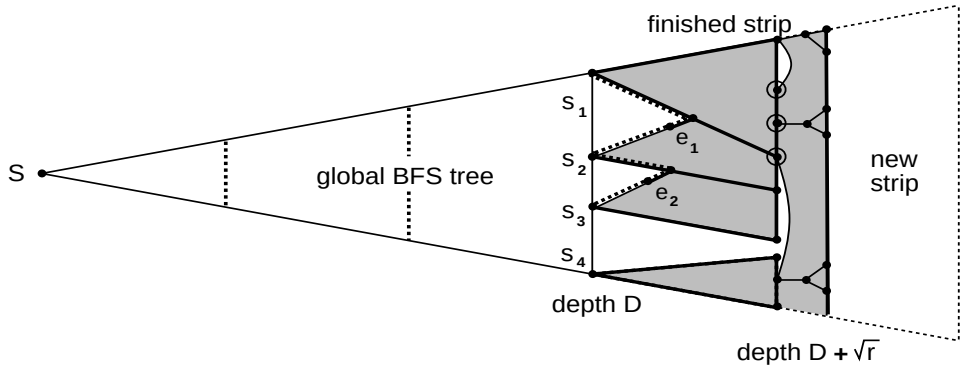

Figure 4: The iterative strip algorithm after the second iteration. Now the circled vertices which were active frontier vertices at the beginning of the iteration are expanded. One of the expansions resulted in a collision edge. Now the strip consists of only one connected component (shaded area). There are six frontier vertices which become source vertices of the next strip. All frontier vertices have depth $D + \sqrt{r}$.

has known path length less than $\sqrt{r}$ from some source vertex in the component.

The robot handles each connected component in turn, as described above. In the next iteration it combines the components now connected by collision edges, and explores the new active frontier vertices in these combined components. Lemma 2 states that at most $\log V$ iterations cause all frontier vertices to become inactive. That is, all frontier vertices are depth $\sqrt{r}$ from the source vertices of this strip. These frontier vertices are the new sources for the next strip.

```
ITERATIVE-STRIP(s, r)
 1  For i = 1 To √r Do
 2      For each source vertex u in strip i Do
 3          relocate to u
 4          BFS from u to depth √r, but do not enter previously
              explored territory
 5      While there are any active connected components Iterate
 6          For each active connected component c Do
 7              Repeat
 8                  let v1, v2, v3, ... be active frontier vertices
                      exclusively in c with smallest depth among
                      active frontier vertices in c
 9                  relocate to each of v1, v2, v3, ..., and expand
10              Until no more active frontier vertices exclusively in c
11          determine new and active connected components
```

Lemma 2 *At most* $\log V$ *iterations per strip are needed to explore a strip and extend the global BFS tree by depth* $\sqrt{r}$.

Proof: If there are initially l source vertices, then after the first iteration there are at most l connected components. If a component does not collide with another active component, then it will have no active frontier vertices for the next iteration. The only active components in the next iteration are those that have collided with other components, and thus, each iteration halves the number of components with active frontier vertices. After at most $\log V$ iterations there is no connected component with active frontier vertices left. The robot then has a complete map of the current strip and of the global BFS tree built in previous strips, so it can combine this information and extend the global BFS tree by depth $\sqrt{r}$. □

Theorem 3 ITERATIVE-STRIP *runs in time* $O((E + V^{1.5}) \log V)$.

Proof: We first count the number of edge traversals within a strip. Let V_i and E_i be the number of vertices and edges explored in strip i. For each component, the robot computes a spanning tree of the component, does a DFS of the spanning tree, and expands all vertices that have known shortest path length t from some source vertex (line 9). At each iteration (line 5), components are disjoint, so relocating to all these vertices takes at most $O(V_i)$ edge traversals. Thus, in one iteration, relocating to all vertices in the strip within distance $\sqrt{r}$ takes at most $O(\sqrt{r}V_i)$ edge traversals. Moreover, note that in order for the robot to expand each vertex, it traverses at most $O(E_i)$ edges. Thus, the total number of edge traversals for strip i in one iteration is $O(E_i + \sqrt{r}V_i)$. Combining this with Lemma 2, the total number of edge traversals within strip i to completely explore strip i takes $O((E_i + \sqrt{r}V_i) \log V)$ edge traversals.

Now we count edge traversals for relocating between source vertices in strip i. As in the proof of Theorem 2, in each iteration the robot traverses at most $2V$ edges to relocate between source vertices. Since there are at most $\log V$ iterations, this results in $2V \log V$ edge traversals between source vertices to explore strip i. Thus, the total number of edge traversals to explore strip i is $O((E_i + \sqrt{r}V_i) \log V + 2V \log V)$. Summing over the $\sqrt{r}$ disjoint strips gives $O((E + \sqrt{r}V) \log V + 2V\sqrt{r} \log V) = O((E + \sqrt{r}V) \log V) = O((E + V^{1.5}) \log V)$. □

4 A nearly linear time algorithm for exploring undirected graphs

This section describes an efficiently interruptible algorithm RECURSIVE-STRIP, which gives a piecemeal exploration algorithm with running time $O(E + V^{1+o(1)})$. RECURSIVE-STRIP is the recursive version of ITERATIVE-STRIP; it provides a recursive structure that coordinates the exploration of strips, of approximate BFS trees, and of connected components in a different manner. The robot still, however, builds a global BFS tree from start vertex s strip by strip. The robot expands vertices at the bottom level of recursion.

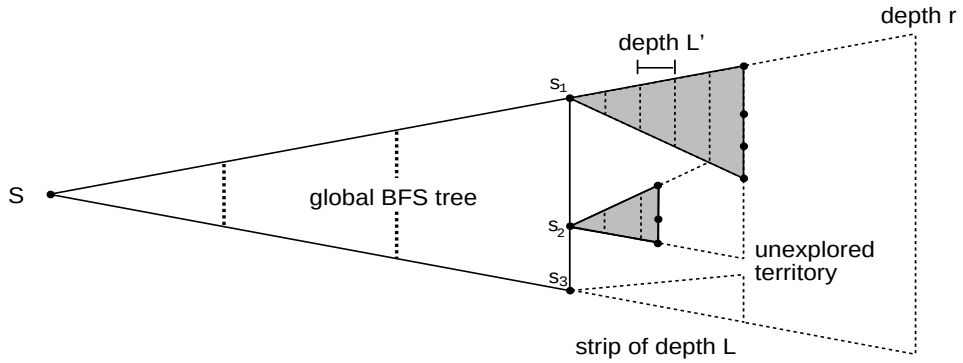

Figure 5: The recursive strip algorithm processing an approximate BFS tree from source vertex s_2 to depth $d_{k-1} = L$. Recursive calls within the tree are of depth $d_{k-2} = L'$.

In RECURSIVE-STRIP, the depth of each strip depends on the level of recursion (see Figure 5). If there are k levels of recursion, then the algorithm starts at the top level by splitting the exploration of G into r/d_{k-1} strips of depth d_{k-1}. Each of these strips is split into d_{k-1}/d_{k-2} searches of strips of depth d_{k-2}, etc. We have $r = d_k > d_{k-1} > \ldots > d_1 > d_0 = 1$.

Each recursive call of the algorithm is passed a set of source vertices *sources*, the *depth* to which it must explore, and a set T of all vertices in the strip already known to have path length less than *depth* from one of the sources. The set of vertices T are known to be connected within the strip being explored. The robot traverses all edges and visits all vertices within *depth* of the sources that have not yet been processed by other recursive calls at this level. RECURSIVE-STRIP$(\{s\}, r, \{s\})$ is called to explore the entire graph.

At recursion level i, the algorithm divides the exploration into strips and processes each strip in turn, as follows. Suppose the strip has l source vertices $v_1, \ldots, v_l$. Then the strip is processed in at most $\log l = O(\log V)$ iterations. In each iteration, the algorithm partitions T into maximal sets $T_1, T_2, \ldots, T_k$ such that each set is known to be connected within the strip. Let S_c denote the set

of source vertices in T_c. A DFS of the spanning tree of the vertices T gives an order for the source vertices in $S_1, S_2, \ldots, S_k$; this spanning tree is used for efficient relocations between these source vertices. Note that all source vertices are known to be connected through the spanning tree of the vertices in T, but they might not be connected within the substrips. Since relocations between the vertices in S_c in the next level of recursion use a spanning tree of T_c, for efficiency the vertices of T_c must be connected within the substrip. After partitioning the vertices into connected components within the strip, for each connected component T_c, the robot relocates (along a spanning tree) to some arbitrary source vertex in S_c. It then calls the algorithm recursively with S_c, the depth of the substrip, and the vertices T_c which are connected to the sources S_c within the substrip.

The remaining iterations in the strip combine the connected components until the strip is finished. Then the robot continues with the next strip in the same level of recursion. Or, if it finished the last strip, it relocates to its starting position and returns to the next higher level of recursion.

```
RECURSIVE-STRIP(sources, depth, T)
 1  If depth = 1
 2     Then
 3          let v_1, v_2, ..., v_k be the depth-first ordering of sources
              in spanning tree
 4          For i = 1 To k Do
 5            relocate to v_i
 6            If v_i has adjacent unexplored edges
 7                 Then traverse v_i's incident edges
 8          T = T ∪ {newly discovered vertices}
 9          Return
10  Else
11     determine next depth as in proof of Theorem 4
12     number-of-strips ← depth/next-depth
13     For i = 1 To number-of-strips Do
14          determine set of source vertices
15          For j = 1 To number-of-iterations Do
16            partition vertices in T into maximal sets T_1, T_2, ..., T_k
                   such that vertices in each T_c are known to be
                   connected within strip i
17            For each T_c in suitable order Do
18                 let S_c be the source vertices in T_c
19                 relocate to some source s ∈ S_c
20                 RECURSIVE-STRIP(S_c, next-depth, T_c)
21                 T = T ∪ T_c
22  relocate to some s ∈ sources
23  Return
```

Theorem 4 RECURSIVE-STRIP *runs in time* $O(E + V^{1+o(1)})$.

Proof: At a particular call of RECURSIVE-STRIP, there are four situations in which the robot traverses edges:

1. expansion of vertices in line 7
2. relocating to sources in lines 5 and 19
3. relocations due to recursive calls in line 20
4. relocation back to a beginning source vertex in line 22

We count edge traversals for each of these cases. First we give some notation. We consider the top level of recursion to be a level-k recursive call, and the bottom level of recursion to be a level-0 recursive call. For a particular level-i call of RECURSIVE-STRIP, let C_i denote the number of edge traversals due to relocations, and let E_i denote the number of distinct edges that are traversed due to relocation. Let V_i denote the number of vertices incident to these edges and whose incident edges are all known at the end of this call. Let ρ_i be a uniform upper bound on C_i/V_i. Thus, if the depth of recursion is k then the total number of edge traversals is bounded by $O(V\rho_k)$.

First we observe that each vertex is expanded at most once, so there are at most $O(E+V)$ edge traversals due to exploration at line 7 in the bottom level of recursion.

Second, for a level-i call, we count the number of edge traversals for relocation between source vertices. Since all the source vertices in the call are connected by a tree of size $O(V_i)$, relocating to all source vertices at the start of one strip takes $O(V_i)$ edge traversals. With d_i/d_{i-1} strips and $\log V$ iterations per strip, there are $V_i \log V \frac{d_i}{d_{i-1}}$ edge traversals for relocations between source vertices.

Third, we now count traversals for recursive calls within a level-i call. Note that our algorithm avoids re-exploring previously explored edges. Thus, for a level-i call, when working on a particular strip l, for each iteration within this strip, the sets of vertices whose edges are explored in each recursive call are disjoint. Suppose that, in this strip, in one iteration the procedure makes k recursive calls, each at level $i-1$. Then let $C_{i-1}^{(j)}$, $1 \le j \le k$, denote the number of edge traversals due to relocations resulting from the j-th recursive call, and let $V_{i-1}^{(j)}$ denote the number of vertices adjacent to these edges. Furthermore, let $V_{l,i}$ denote the number of vertices which are in strip l of this procedure call at recursion level i. Then we would like first to calculate

$$\sum_{j=1}^{k} C_{i-1}^{(j)},$$

which is the number of edge traversals due to relocation in recursive calls in one iteration within this strip. This is at most

$$\sum_{j=1}^{k} \rho_{i-1} V_{i-1}^{(j)} = \rho_{i-1} \sum_{j=1}^{k} V_{i-1}^{(j)}.$$

Since the recursive calls are disjoint, $\sum_{j=1}^{k} V_{i-1}^{(j)} = V_{l,i}$, and thus the number of edge traversals due to relocations in recursive calls in one iteration within this strip is at most $\rho_{i-1} V_{l,i}$. Finally, since there are $\log V$ iterations in each strip, and all strips are disjoint from each other, the number of

edge traversals due to recursive calls is at most $\rho_{i-1} V_i \log V$.

Fourth, note that we relocate once at the end of each procedure call of RECURSIVE-STRIP (see line 22). This results in at most V_i edge traversals.

Thus, the number of edge traversals due to relocation (not including relocations for expanding vertices) is described by the recurrence $C_i \leq V_i \log V \frac{d_i}{d_{i-1}} + \rho_{i-1} V_i \log V + V_i$. Normalizing by V_i, we get the following recurrence:

$$\rho_i = \left(\frac{d_i}{d_{i-1}} + \rho_{i-1}\right) \log V + O(1)$$

Solving the recurrence for ρ_k gives:

$$\begin{aligned}
\rho_k &\leq \left(\frac{d_k}{d_{k-1}}\right) \log V + \left(\frac{d_{k-1}}{d_{k-2}}\right) \log^2 V + \ldots + \left(\frac{d_1}{d_0}\right) \log^k V + \rho_0 \log^k V + \sum_{i=0}^{k-1} \log^i V \\
&\leq \left(\frac{d_k}{d_{k-1}}\right) \log V + \left(\frac{d_{k-1}}{d_{k-2}}\right) \log^2 V + \ldots + \left(\frac{d_1}{d_0}\right) \log^k V + O(\log^k V) \\
&\leq \sum_{i=1}^{k} \left(\frac{d_{k+1-i}}{d_{k-i}}\right) \log^i V + O(\log^k V)
\end{aligned}$$

We note that $\rho_0 = O(1)$, since at the bottom level, if there are V' vertices expanded, then the number of edge traversals due to relocation is $O(V')$. The product of the first k terms in the recurrence is

$$\prod_{i=1}^{k} \log^i V \left(\frac{d_{k+1-i}}{d_{k-i}}\right) = \frac{d_k}{d_0} (\log V)^{(k+1)k/2} = r (\log V)^{(k+1)k/2}.$$

We choose $d_{k-1}, d_{k-2}, \ldots$ by setting each of the first k terms equal to the k-th root of this product. (Note that this also specifies how to calculate depth d_{i-1} from depth d_i in line 11.) Substituting, we get:

$$\begin{aligned}
\rho_k &\leq k\, r^{1/k} (\log V)^{(k+1)/2} + O(\log^k V) \\
&\leq 2^{\log k} \cdot 2^{\frac{\log r}{k}} \cdot 2^{\frac{k+1}{2} \log\log V} + O(2^{k \log\log V})
\end{aligned}$$

Choosing $k = \left(\frac{\log V}{\log\log V}\right)^{1/2}$ gives us

$$\begin{aligned}
\rho_k &\leq 2^{\log \sqrt{\log V} - \log \sqrt{\log\log V} + \frac{\sqrt{\log\log V}}{\sqrt{\log V}} \log V + (\frac{\sqrt{\log V}}{\sqrt{\log\log V}} + 1) \frac{1}{2} \log\log V} + O(2^{\frac{\sqrt{\log V}}{\sqrt{\log\log V}} \log\log V}) \\
&\leq 2^{\log\log V - \frac{1}{2}\log\log\log V + \frac{3}{2}\sqrt{\log V \log\log V}} + O(2^{\sqrt{\log V \log\log V}})
\end{aligned}$$

$$= \quad 2^{O(\sqrt{\log V \log\log V})}$$

and thus

$$C_k = V 2^{O(\sqrt{\log V \log\log V})}$$

which is $V^{1+o(1)}$. Adding the edge traversals for relocation to the edge traversals for exploration gives us $O(E + V^{1+o(1)})$ edge traversals total. □

5 Treasure Hunting

We now consider an application of our algorithms to the problem of finding a treasure (or a lost child, or a particular landmark) in an unknown, potentially infinite graph $G = (V, E)$. If the robot searching for the treasure knows that the treasure is close to its start location, it should explore in a manner such that it does not get too far away from this location.

We give the procedure Treasure-Search, which uses the Recursive-Strip algorithm as a subroutine. If the treasure is shortest path distance δ_T away from the source vertex, this algorithm maintains the condition that the robot is never further from the source than Δ, where $\Delta \leq \delta_T + o(\delta_T)$. Following procedure Treasure-Search, the robot traverses $O(E + V^{1+o(1)})$ edges, where E and V are the total number of distinct edges and vertices within radius Δ from the source.

The robot explores the graph for the treasure in phases. In each phase, the size of the strip to be explored changes. The change at phase i depends on $\epsilon_i = 1/\sqrt{i}$. Initially, the robot explores the graph out to distance $r_1 = 1 + \epsilon_1$. Next, the robot extends its exploration by a factor of $1 + \epsilon_2$. That is, the size of the next strip is $(1 + \epsilon_1)(1 + \epsilon_2) - (1 + \epsilon_1)$, and at the end of the second phase, the robot has learned the graph out to distance $r_2 = (1 + \epsilon_1)(1 + \epsilon_2)$. After extending the next strip, the robot has learned the graph out to distance $r_3 = (1 + \epsilon_1)(1 + \epsilon_2)(1 + \epsilon_3)$, and so on. In each phase i, the robot initially calls Recursive-Strip from the set of source vertices (vertices at distance r_{i-1}). When the robot finds collision edges, it does not re-explore edges. Thus, within each phase, it may take up to $\log V$ iterations (as in Iterative-Strip and Recursive-Strip) before it has explored the entire strip.

Lemmas 3 and 4 bound the number of phases in the Treasure-Search procedure. Using Lemma 3, Theorem 5 shows that the robot does not get too far away from the source vertex, and using Lemma 4, Theorem 6 bounds the number of edges the robot traverses.

```
TREASURE-SEARCH(s)
 1  i = 0
 2  r_0 = 1
 3  Do until treasure is found
 4      i = i + 1
 5      ε_i = 1/√i
 6      r_i = r_{i-1} · (1 + ε_i)
 7      If i = 1
 8          Then
 9              RECURSIVE-STRIP({s}, r_1, {s})
10      Else
11          let T be be the set of source vertices distance
              r_{i-1} away from s
12          For j = 1 To number-of-iterations Do
13              partition vertices in T into maximal sets T_1, ..., T_k
                    such that vertices in each T_c are known to be
                    connected within strip i
14              For each T_c in suitable order Do
15                  let S_c be the source vertices in T_c
16                  relocate to some source s ∈ S_c
17                  RECURSIVE-STRIP(S_c, r_i − r_{i-1}, T_c)
18                  T = T ∪ T_c
```

Lemma 3 *The number of phases in* TREASURE-SEARCH *is at least* $\log \delta_T$.

Proof: Since $\epsilon_1 > \epsilon_2 > \epsilon_3 \ldots$, we know that, for any j, $(1+\epsilon_1)(1+\epsilon_2)\ldots(1+\epsilon_j) \leq (1+\epsilon_1)^j$. Thus, if we let j be the smallest number such that $(1+\epsilon_1)^j \geq \delta_T$, then we know that the number of phases i to reach the treasure at δ_T is at least j. Since $\epsilon_1 = 1$, we have $2^j \geq \delta_T$, or $j \geq \log \delta_T$. □

Lemma 4 *The number of phases in* TREASURE-SEARCH *is at most* $4 \ln^2 \delta_T + 1$.

Proof: A treasure at depth $\delta_T = 1$ is found in the first phase, so we consider only $\delta_T > 1$. We know that for any j, $(1+\epsilon_j)^j \leq (1+\epsilon_1)(1+\epsilon_2)\ldots(1+\epsilon_j)$. Thus, if $(1+\epsilon_j)^j \geq \delta_T$, we know that the number of phases i is at most j. So we prove the lemma by showing that $(1+\epsilon_{4\ln^2\delta_T})^{4\ln^2\delta_T} \geq \delta_T$, or equivalently, that $4\ln^2\delta_T \ln(1+\epsilon_{4\ln^2\delta_T}) = 4\ln^2\delta_T \ln(1+\frac{1}{2\ln\delta_T}) \geq \ln\delta_T$.

For $|x| < 1$, using a Taylor expansion, we have $\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \cdots$. For $0 < x < 1$, we have $\ln(1+x) > x - \frac{x^2}{2}$. So $4\ln^2\delta_T \ln(1+\frac{1}{2\ln\delta_T}) > (4\ln^2\delta_T)(\frac{1}{2\ln\delta_T} - \frac{1}{8\ln^2\delta_T}) = 2\ln\delta_T - 1/2$,

which is at least $\ln \delta_T$ for $\delta_T \geq 2$. □

Theorem 5 *The robot is never further than $\delta_T + \delta_T/\sqrt{\log \delta_T}$ from the source vertex.*

Proof: Let Δ be the furthest distance the robot gets from the source vertex. Let i be the number of phases that need to be explored to get out to depth δ_T. Then $\Delta - \delta_T$ is at most the depth of the strip in i-th phase. That is, $\Delta - \delta_T \leq (1+\epsilon_1)(1+\epsilon_2)\ldots(1+\epsilon_i) - (1+\epsilon_1)(1+\epsilon_2)\ldots(1+\epsilon_{i-1}) = (1+\epsilon_1)(1+\epsilon_2)\ldots(1+\epsilon_{i-1})\epsilon_i < \delta_T\epsilon_i$. Lemma 3 shows that the total number of strips explored is at least $\log \delta_T$. Thus, ϵ_i is at most $1/\sqrt{\log \delta_T}$, and $\Delta \leq \delta_T + \delta_T/\sqrt{\log \delta_T} = \delta_T + o(\delta_T)$. □

Theorem 6 *Given a treasure at distance δ_T from the source, procedure* Treasure-Search *traverses at most $O(E + V^{1+o(1)})$ edges, where E and V are the total number of distinct edges and vertices within radius $\Delta \leq \delta_T + o(\delta_T)$ from the source.*

Proof: Since the edges traversed in the different phases are disjoint, the number of edges traversed, ignoring relocations between source vertices in line 16, is at most $O(E + V^{1+o(1)})$. To get between source vertices in line 16, a spanning tree of the known vertices can be used. (Note that for recursive calls of Recursive-Strip, the algorithm relocates between source vertices using the vertices connected within the appropriate strip.) By Lemma 4, we know the number of phases is at most $4 \ln^2 \delta_T$, and in each phase it may take up to $\log V$ iterations to explore the entire strip. Thus there are an additional $4V \ln^2 \delta_T \log V$ edge traversals due to relocations between source vertices, and this gives a total of $O(E+V^{1+o(1)})$ edge traversals for the entire Treasure-Search procedure.

□

6 Concluding Remarks

We have presented an efficient $O(E + V^{1+o(1)})$ algorithm for piecemeal learning of arbitrary, undirected graphs. The only lower bound known for this problem is the trivial bound $\Omega(E + V)$, and it is not known whether a linear-time algorithm exists.

We have also given an algorithm for the application of treasure hunting on potentially infinite graphs. Is it possible (we conjecture not) to find a treasure in time nearly linear in the number of those vertices and edges whose distance to the source is less than or equal to that of the treasure?

References

[1] Baruch Awerbuch, Bonnie Berger, Lenore Cowen, and David Peleg. Near-linear time construction of sparse neighborhood covers. *SIAM Journal of Computing*, 28(1):254–262, 1998.

[2] Baruch Awerbuch and Robert G. Gallager. Distributed BFS algorithms. In *Proceedings of the 26th Symposium on Foundations of Computer Science*, pages 250–256, October 1985.

[3] Baruch Awerbuch and Robert G. Gallager. A new distributed algorithm to find breadth first search trees. *IEEE Transactions on Information Theory*, IT-33(3):315–322, 1987.

[4] E. Bar-Eli, Piotr Berman, A. Fiat, and Peiyuan Yan. On-line navigation in a room. *Journal of Algorithms*, 17:319–341, 1994.

[5] Michael A. Bender and Donna K. 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, November 1994.

[6] Margrit Betke. Algorithms for exploring an unknown graph. Master's thesis, MIT Department of Electrical Engineering and Computer Science, February 1992. (Published as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-536, March 1992).

[7] Margrit Betke and Leonid Gurvits. Mobile robot localization using landmarks. *IEEE Transactions on Robotics and Automation*, 13(2):251–263, April 1997.

[8] Margrit Betke, Ronald L. Rivest, and Mona Singh. Piecemeal learning of an unknown environment. *Machine Learning*, 18(2/3):231–254, February/March 1995.

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

[10] Avrim Blum, Prabhakar Raghavan, and Baruch Schieber. Navigating in unfamiliar geometric terrain. *Siam Journal of Computing*, 26(1):110–137, 1997.

[11] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press/McGraw-Hill, 1990.

[12] Xiaotie Deng, Tiko Kameda, and Christos H. Papadimitriou. How to learn an unknown environment I: The rectilinear case. *Journal of the ACM*, 45(2):215–245, March 1998.

[13] 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.

[14] Gregory Dudek, Michael Jenkin, Evangelos Milios, and David Wilkes. Using multiple markers in graph exploration. In *SPIE Vol. 1195 Mobile Robots IV*, pages 77–87, 1989.

[15] Rolf Klein. Walking an unknown street with bounded detour. *Computational Geometry: Theory and Applications*, 1(6):325–351, June 1992.

[16] Jon M. Kleinberg. The localization problem for mobile robots. In *Proceedings of the 35th IEEE Symposium on Foundations of Computer Science*, pages 521–531, 1994.

[17] C. Y. Lee. An algorithm for path connection and its applications. *IRE Transactions on Electronic Computers*, EC-10(3):346–365, 1961.

[18] Edward F. Moore. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching*, pages 285–292, 1959.

[19] Christos H. Papadimitriou and Mihalis Yanakakis. Shortest paths without a map. *Theoretical Computer Science*, 84:127–150, 1991.

[20] Nagewara S. V. Rao, Srikumar Kareti, Weimin Shi, and S. Sitharama Iyengar. Robot navigation in unknown terrains: Introductory survey of non-heuristic algorithms. Technical Report ORNL/TM-12410, Oak Ridge National Laboratory, July 1993.

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

[22] Kathleen Romanik and Sven Schuierer. Optimal robot localization in trees. In *Proceedings of the 12th Symposium on Computational Geometry*, 1996.