Piecemeal Graph Exploration by a Mobile Robot *

Baruch Awerbuch† Margrit Betke Ronald L. Rivest Mona Singh
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

We study how a mobile robot can piecemeal learn an unknown environment. 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 s (for refueling, say). The environment is modelled as an arbitrary, undirected graph, which is initially unknown to the robot. We assume that the robot can distinguish vertices and edges which 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 G 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 address the related problem of searching a graph for a particular distinguished location or treasure. If this treasure is likely to be near s, then the robot should explore in a breadth-first manner. We show how the robot can explore while never being much further than δ away from s, where δ is the shortest path distance from s of the unvisited vertex nearest to s. We show that if the robot is never more than δ away from s (as in traditional BFS), then there are graphs for which the robot traverses $\Omega(E^2)$ edges. In the algorithm we give, the robot traverses $O(E + V^{1+o(1)})$ edges, and maintains the following property: if δ is the distance from s to the unvisited vertex nearest to s, then the robot is never further than $\delta + o(\delta)$ away from s.

*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, the Siemens Corporation, and a special grant from IBM. The authors can be reached at `baruch@blaze.cs.jhu.edu`, `margrit@theory.lcs.mit.edu`, `rivest@theory.lcs.mit.edu`, and `mona@theory.lcs.mit.edu`.

†Also at Johns Hopkins University, Baltimore, MD 21218.

1 Introduction

Environment learning and algorithmic motion planning for robots have recently become active research areas. The goal is to find efficient algorithms for a robot to learn about or navigate in its environment. Such algorithms are now useful in practice: there are working meal delivery robots in hospitals [15], and vehicles that navigate autonomously on highways [4]. More formal theoretical approaches to these problems have also been studied extensively (e.g., [20, 10, 22, 5]).

We study the problem of *piecemeal learning* of an unknown environment [8]. The robot's goal is to learn a complete map of its environment while satisfying the *piecemeal constraint* that learning must be done a "piece at a time," with the robot returning to the starting point s after each learning phase. Why might mobile robot exploration be done piecemeal? Robots may explore environments that are too risky or costly for humans: the inside of a volcano (e.g., CMU's Dante II robot) or the surface of Mars. Or, 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 s before it breaks down or runs out of power. At the start position s, the robot can cool off, recharge, or drop off samples collected.

Approaches to modelling a robot's environment come from graph theory, computational geometry, on-line algorithms, and the theory of finite automata. The model used here was introduced by Betke, Rivest, and Singh [8]. The robot's task is to learn an unknown environment modeled as an undirected graph $G = (V, E)$ in a piecemeal manner. The robot's efficiency (or running time) is measured in terms of the number of edges traversed. The main difficulty in our work lies in designing efficient, but analyzable, robot exploration algorithms. We give an almost linear time algorithm: it achieves $O(E + V^{1+o(1)})$ running time. The most efficient previously known algorithm has $O(E + V^2)$ running time. We also give a simpler but less efficient algorithm that runs in $O(E + V^{1.5})$ time.

A robot can explore grid-graphs with rectangular obstacles in a piecemeal manner in linear time, if the robot is given a bound on the number of edges it may traverse in each learning phase (Betke, Rivest, and Singh [8]). We extend these results to show that the robot can learn *any* undirected graph piecemeal in almost linear 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 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 goal location, it should explore in a breadth-first manner from its current position.

BFS is a classic technique for searching graphs [19, 18, 11]. However, standard BFS is efficient 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. We change the classical BFS model to a more difficult *teleport-free* exploration model, and give efficient *approximate* BFS algorithms: algorithms that satisfy the "approximate BFS constraint" (the robot does not move much further away from s than the distance from s to the unvisited vertex nearest to s). Our first teleport-free BFS algorithms never visit a vertex more than twice as far from s as the nearest unvisited vertex is from s. Our final teleport-free BFS algorithm 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. For the treasure hunting problem, 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. In contrast, we give a simple example to show that 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. Our final treasure hunting algorithm is also a solution to the piecemeal learning problem, but it is more complicated than our fastest piecemeal learning algorithm.

Previous work

Papadimitriou and Yanakakis [20] developed one of the first 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 [22] 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 the 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 [3] give an efficient algorithm for reaching the center of a two-dimensional room with obstacles. Betke [7] and Kleinberg [17] 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 [16, 14]. Rao, Kareti, Shi, and Iyengar [21] give a survey of work on "robot navigation in unknown terrains."

Our techniques are inspired by the work of Awerbuch and Gallager [1, 2]. We observe that our learning model bears some similarity to the asynchronous distributed model. This similarity is surprising and has not been explored in the past.

2 Model and statement of main results

We model the robot's environment as a finite undirected graph $G = (V, E)$ with a distinguished start vertex s. The graph is initially unknown to the robot. Each vertex in the graph represents an accessible location, and each edge represents a connection between adjacent locations. During each step of exploration, the robot moves from its current location to an adjacent location; it is not allowed to "teleport" from one vertex to another distant vertex. The robot can recognize previously visited vertices. The robot can distinguish the edges incident to its current vertex and it knows which edges it has traversed already, but it has no vision or long-range sensors. The robot incurs a cost only for traversing an edge; thinking and path planning (computation) are free.

We consider two closely related constraints on the exploration: the "piecemeal constraint" to model learning unknown environments in phases, and the "approximate BFS constraint" to model exploring an unknown graph in order to find a treasure.

2.1 Piecemeal Learning

The robot's goal in piecemeal learning is to explore its entire (unknown) environment, while satisfying the piecemeal constraint that it must return every so often to its starting point. To assure that the learner can reach any vertex in the graph, do some exploration, and then get back to the start vertex, we assume the robot may traverse $(2+\alpha)R$ edges in one exploration phase, where $\alpha > 0$ is some constant and R is the *radius* of the graph (the maximum of all shortest path distances between s and any vertex in G).

We say an exploration *efficiently interruptible* if the robot always knows a path of explored edges of length at most R back to s.

Theorem 1 *An efficiently interruptible algorithm 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 $O(T(n, m))$.*

The proof of this theorem is similar to one shown by Betke, Rivest, and Singh in a previous paper [8].

All the algorithms we present in this paper are efficiently interruptible, and thus give efficient piecemeal learning algorithms for undirected graphs. Our main theorem is:

Theorem 2 *Piecemeal learning of a general undirected graph $G = (V, E)$ can be done in time $O(E + V^{1+o(1)})$.*

Proof sketch: In the Recursive Strip-Algorithm, given in Section 5, the robot always knows a path from its current location back to the source vertex of length at most the radius of the graph. We discuss the running time of this algorithm in Section 5. The bound is $O(E + V2^{O(\sqrt{\log V \log\log V})})$. By Theorem 1, this algorithm can be interrupted efficiently to give a piecemeal learning algorithm. $\square$

2.2 Treasure Hunting

If the robot's goal is to explore an unknown environment in order to find a treasure that is believed to be the near s, then the robot should explore in a breadth-first manner.

In traditional BFS, 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 $\Delta \leq \delta$ (actually $\Delta = \delta$) 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 which maintains $\Delta \leq \delta$ (such as 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 treasure hunting problem efficiently while maintaining the "approximate BFS constraint." Our initial algorithms Strip-Explore, Iterative-Strip, and Recursive-Strip, described in Sections 3, 4, and 5, maintain $\Delta \leq 2\delta$: the robot is never more than twice as far from s as is the nearest unvisited vertex. Our final algorithm Treasure-Search, given in Section 6, 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; however, it is more complicated. Our main theorem about treasure hunting is:

Theorem 3 *Given an unknown graph with a treasure at distance δ_T from s, a robot can find the treasure while getting at most distance $\Delta = \delta_T + o(\delta_T)$ away from the source vertex, with an algorithm of running time $O(E + V^{1+o(1)})$, where E and V are the total number of distinct edges and vertices within radius Δ from the source.*

Proof sketch: The algorithm Treasure-Search given in Section 6 satisfies the theorem. We discuss the properties of this algorithm in Section 6. □

3 An exploration 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 $1 \dots i$ in time $O(V)$, and there are at most r layers. 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, the robot does not relocate to frontier vertices that have no unexplored edges. Second, it only explores vertices within a given distance-bound L of the given source vertex s. (The first modification, while seemingly straightforward, is essential for our analysis of our more complex algorithms that use Local-BFS as a subroutine at various source vertices). A procedure call of the form Local-BFS(s, r), where s is the source vertex of the graph and r is its radius, would cause the robot to explore the entire graph.

Awerbuch and Gallager [1, 2] give a distributed BFS algorithm which partitions the network in *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.

Local-BFS(s, L)

```
for i = 0 to L − 1 do
        let current-verts = all vertices at distance i from s
        for each u ∈ current-verts do
                if u has any incident unexplored edges
                        then
                          Relocate to u
                          Traverse each unexplored edge incident to u
relocate to s
```

Our algorithm, Strip-Explore, uses the idea of search in strips in a new way. See Figure 1. The robot explores the graph in strips of width L. First the robot does Local-BFS(s, L) to explore the first strip. It then explores the second strip as follows. Suppose there are k frontier vertices $v_1, v_2, \dots, v_k$ 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.

The naive algorithm described above is inefficient, due to the to overlap between the trees in the forest at a given level, causing portions of each strip to be repeatedly re-explored.

The Strip-Explore presented below solves this problem by using the Local-BFS procedure as the basic subroutine, instead of using a naive BFS. (See Figure 2.) Using this algorithm, the explorer searches in a breadth-first manner, but ignores previously explored territory. The only time the robot traverses edges which 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)
numstrips = ⌈r/L⌉
sources = {s}
for i = 1 to numstrips do
        for each u ∈ sources do
                Relocate to u
                LOCAL-BFS(u, L)
        sources = all frontier vertices
```

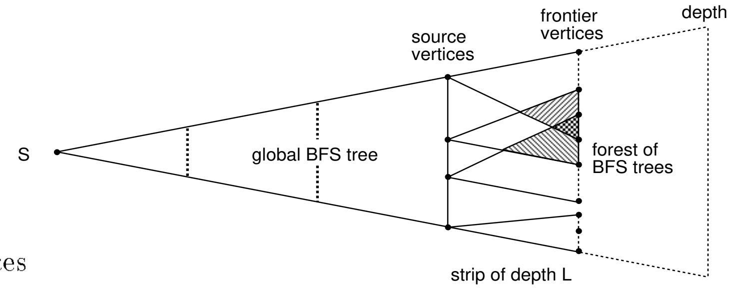

Figure 1: The STRIP-EXPLORE. In the naive algorithm, the shaded areas are retraversed completely. In the strip algorithm, the shaded areas are passed through more than once only if necessary to get to frontier vertices.

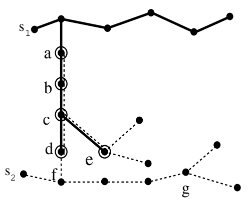

Figure 2. Contrasting BFS and Local-BFS: Consider doing a BFS of depth 5 from s_1, followed by a BFS of depth 5 from s_2. (The depth of the strip is $L = 5$.) The BFS from s_2 revisits vertices a, b, c, d, e. On the other hand, if the BFS from s_1 is followed by a LOCAL-BFS from s_2, then it only revisits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

Theorem 4 STRIP-EXPLORE *runs in time* $O(E + V^{1.5})$.

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 $2rV/L + 2r$.

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 line 8 of procedure LOCAL-BFS is called, at most L edges are traversed. 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 $2rV/L + 2r + 3LV + 2E$, which is $O(rV/L + r + LV)$. When L is chosen to be $\sqrt{r}$, this gives $O(E + V^{1.5})$ edge traversals. □

It is easy to show for STRIP-EXPLORE, and the generalizations of it given in later sections, that $\Delta \leq 2\delta$ at all times; the worst case is when the treasure is at the beginning of the second strip.

4 Iterative strip algorithm

In this section, we describe ITERATIVE-STRIP, an algorithm similar to the STRIP-BFS algorithm. It is an efficiently interruptible algorithm for undirected graphs inspired by Awerbuch and Gallager's [1] distributed iterative BFS algorithm. Although its running time of $O((V^{1.5} + E)\log V)$ is worse than the running time of STRIP-BFS, its recursive version (described in the next section) is more efficient than STRIP-BFS. (It is not clear how to recursively implement STRIP-BFS as efficiently, because the search trees in a strip are not disjoint.)

Following the ITERATIVE-STRIP, the robot grows a global BFS tree with root s strip by strip, similar to STRIP-BFS. Unlike STRIP-BFS, 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.

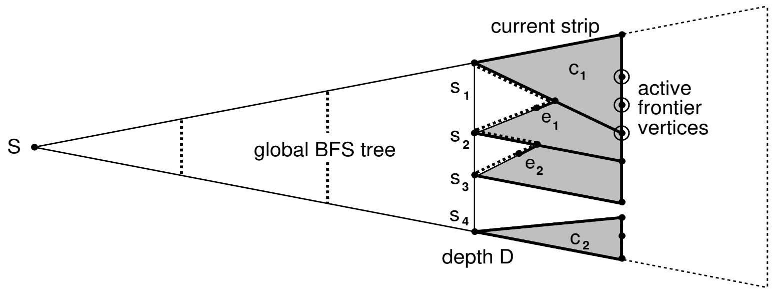

Figure 3: The iterative strip algorithm after the first iteration on the fourth strip. Two connected components c_1, c_2 have been explored. The collision edges e_1 and e_2 connect the first three approximate BFS trees. The dashed line shows how source vertices s_1, s_2, s_3 connect within the strip.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

In the first iteration, a strip is explored much as in the STRIP-BFS. 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 penetrate into the other tree. Unlike STRIP-BFS, the robot does not traverse explored edges to get to the active frontier vertices on other trees. Therefore, after the first iteration, the trees explored are *approximate BFS trees*, which may have frontier vertices with depth less than $\sqrt{r}$ from some source vertex. These vertices become *active frontier vertices* for the next iteration. Thus, 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).

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 does not have to traverse any edges outside of 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.

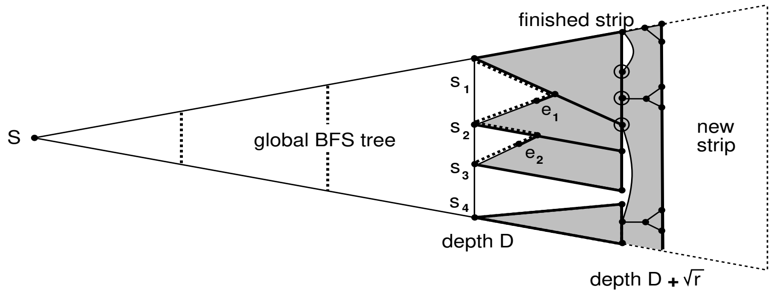

Figure 4: The iterative strip algorithm after the second iteration. Now the circled vertices which were active frontier vertices at the beginning of the iteration are expanded. One of the expansions resulted in a collision edge. Now the strip consists of only one connected component (shaded area). There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

The robot explores active frontier vertices in one connected component as follows. He computes (mentally) a spanning tree of the vertices in the current strip. This spanning tree lies within the strip. Let d be the least depth of any active frontier vertex in the component from a source vertex. He 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. He 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

ITERATIVE-STRIP(s,r)

for $i = 1$ **to** $\sqrt{r}$ **do**
    for each source vertex u in strip i **do**
        Relocate to u
        BFS from u to depth $\sqrt{r}$, but do not enter previously explored territory
    while there are any active connected components **iterate**
        for each active connected component c **do**
            repeat
                let $v_1, v_2, v_3, \ldots$ be active frontier vertices exclusively in c
                  with smallest depth among active frontier vertices in c
                relocate to each of $v_1, v_2, v_3, \ldots$, and expand
            until no more active frontier vertices exclusively in c
        determine new and active connected components

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

The robot processes each connected component in turn, as described above. Then the next iteration starts in which it combines the components now connected by new collision edges and explores the new active frontier vertices in those components. Lemma 2 states that at most $\log V$ iterations cause all frontier vertices to not be active any more; then the only active frontier vertices are the new sources of the next strip.

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

Theorem 5 ITERATIVE-STRIP *runs in time* $O((E + V^{1.5}) \log V)$.

Proof sketches are included in the Appendix. □

5 The recursive strip algorithm

In this section, we give an efficiently interruptible algorithm RECURSIVE-STRIP, which gives a piecemeal learning algorithm which traverses at most $O(E + V^{1+o(1)})$ edges. RECURSIVE-STRIP is the recursive version of ITERATIVE-STRIP; it provides a recursive structure which 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.

In this algorithm, 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 search of G into V/d_{k-1} strips of depth d_{k-1}. Each of these is split into d_{k-1}/d_{k-2} searches of strips of depth d_{k-2}, etc. We have $V = d_k > d_{k-1} > \ldots > d_1 > d_0 = 1$.

```
Recursive-Strip(sources, depth, T)
if depth = 1
   then let v_1, v_2, ..., v_k be the depth-first ordering of sources in spanning tree of sources
        for i = 1 to k do
             relocate to v_i
             if v_i has adjacent unexplored edges then traverse v_i's incident edges
        T = T ∪ {newly discovered vertices}
        return
   else determine next depth
        number-of-strips ← depth/next-depth
        for i = 1 to number-of-strips do
             determine set of source vertices
             for j = 1 to number-of-iterations do
                  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
                  for each T_c in suitable order do
                       let S_c be the source vertices in T_c
                       Relocate to some source s ∈ S_c
                       Recursive-Strip(S_c, next-depth, T_c)
                       T = T ∪ T_c
Relocate to some s ∈ sources
return
```

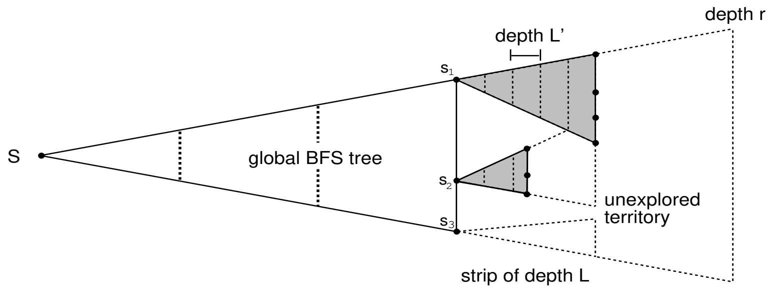

Figure 5: The recursive strip algorithm processing an approximate BFS tree from source vertex s_2 to depth $d_{k-1} = L$. Recursive calls within the tree are of depth $d_{k-2} = L'$.

Each recursive call of the algorithm is passed a set of source vertices *sources*, the *depth* to which it must search, and a set T of all vertices in the strip already known to be less than distance *depth* from one of the sources. The algorithm traverses all edges and visits all vertices within distance *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 search into strips and processes each strip in turn, as follows. Suppose the strip has l source vertices $v_1, \ldots, v_l$. 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 known to be connected within the strip. Let S_c denote the 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 strip. After partitioning the vertices into connected components within the strip, for each connected component T_c, the algorithm 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 strip, and the vertices T_c which are connected to the sources S_c within the strip.

The remaining iterations in the strip combine the connected components until the strip is finished. Then the algorithm 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.

Theorem 6 RECURSIVE-STRIP *runs in time* $O(E + V^{1+o(1)})$.

A proof sketch included in the Appendix. □

6 Searching a graph for a treasure

We now consider the problem of searching for a treasure in a potentially infinite graph $G = (V, E)$. We give the procedure TREASURE-SEARCH, which uses the RECURSIVE-STRIP algorithm as a subroutine. If the treasure is at a location which is 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 $\Delta \leq \delta_T + o(\delta_T)$. Procedure TREASURE-SEARCH 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 searches the graph for the treasure in phases. In each phase i, the robot calls RECURSIVE-STRIP to search a strip in the graph. The size of the strips changes over time. The change depends on $\epsilon_1, \epsilon_2, \epsilon_3, \ldots$, where $\epsilon_i = 1/\sqrt{i}$. Initially, the robot explores the graph out to distance $r_1 = 1 + \epsilon_1$. Next, the robot extends his search 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 the robot now knows the graph out to distance $r_2 = (1+\epsilon_1)(1+\epsilon_2)$. After extending the next strip, the robot knows the graph out to distance $r_3 = (1+\epsilon_1)(1+\epsilon_2)(1+\epsilon_3)$, and so on.

Treasure-Search(s)
$i = 0$
$r_0 = 1$
Do until treasure is found:
    $i = i + 1$
    $\epsilon_i = 1/\sqrt{i}$
    $r_i = r_{i-1} \cdot (1 + \epsilon_i)$
    let S be set of source vertices distance r_{i-1} away from s
    Recursive-Strip($S, r_i - r_{i-1}, S$)

The correctness of this procedure can be shown by the following lemmas, which can be used to prove Theorem 3.

Lemma 3 *The number of phases of procedure* Treasure-Search *is at least* $\log \delta_T$ *and at most* $\ln^2 \delta_T$.

Proof sketch: The number of phases is at least j where $(1+\epsilon_1)^j \geq \delta_T$. The number of phases is at most m where $(1+\epsilon_m)^m \geq \delta_T$. We can show the lemma using these inequalities.

Lemma 4 *The robot is never further than* $\delta_T + \delta_T/\sqrt{\log \delta_T}$ *away from the source.*

Proof sketch: Let Δ be the furthest distance the robot gets from the source vertex. In the Treasure-Search algorithm, the most that Δ and δ_T differ is $\delta_T \epsilon_{max}$, where max is the number of phases that need to be explored to get out to depth δ_T. Lemma 3 shows that the total number of strips explored is at least $\log \delta_T$. Thus, ϵ_{max} is at most $1/\sqrt{\log \delta_T}$, and $\Delta \leq \delta_T + \delta_T/\sqrt{\log \delta_T} = \delta_T + o(\delta_T)$. □

7 Open problems

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 linear bound $\Omega(E+V)$. It is open whether there is a linear-time algorithm for piecemeal learning of general graphs.

We have also given an algorithm for the application of treasure hunting on potentially infinite graphs that runs in time nearly linear in E and V, where E and V are the number of distinct edges and vertices within radius Δ from the start vertex. 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?

Appendix

We include in this Appendix some proof sketches of the theorems in the paper.

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 sketch: 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. 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 he can combine this information and extend the global BFS tree by depth $\sqrt{r}$. □

Theorem 5 ITERATIVE-STRIP-BFS *runs in time* $O((E + V^{1.5}) \log V)$.

Proof sketch: We first count the number of edge traversals per strip. Let V_i and E_i be the number of vertices and edges explored in strip i. For each component, vertices of distance t from some source vertex are expanded by computing a spanning tree of the component, doing a DFS of the spanning tree, and expanding all vertices of distance t from some source vertex (lines 8, 9). At each iteration (line 6), components are disjoint, so relocating to all vertices in the strip of distance exactly t 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, he traverses at most $O(E_i)$ edges. Thus, the total number of edge traversals for strip i is $O(E_i + \sqrt{r}V_i)$. Combining this with Lemma 2, and noting that $r \leq V$, proves the lemma. □

Theorem 6 RECURSIVE-STRIP-BFS *runs in time* $O(E + V^{1+o(1)})$.

Proof sketch: 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 5 in the bottom level of recursion.

We now count the edge traversals for relocations. For a particular level-i call of RECURSIVE-STRIP-BFS, 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 $O(V\rho_k)$.

Consider a level-i call. First 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.

We now count traversals for recursive calls within the 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 $C_{i-1}^{(j)}$, $1 \leq j \leq k$, 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 $\rho_{i-1} V_i \log V$.

Finally, note that we relocate once at the end of each procedure call of Recursive-Strip-BFS (see line 19). This results in at most V_i edge traversals.

Thus, the number of edge traversals due to relocation 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, 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:

$$\begin{aligned} \rho_k &\leq \log V \left(\frac{d_k}{d_{k-1}}\right) + \log^2 V \left(\frac{d_{k-1}}{d_{k-2}}\right) + \ldots + \log^k V \left(\frac{d_1}{d_0}\right) + \log^k V c_0 + \sum_{i=0}^{k-1} \log^i V \\ &\leq \log V \left(\frac{d_k}{d_{k-1}}\right) + \log^2 V \left(\frac{d_{k-1}}{d_{k-2}}\right) + \ldots + \log^k V \left(\frac{d_1}{d_0}\right) + O(\log^k V) \end{aligned}$$

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 $O(V')$. Furthermore, notice that the product of the k terms in the recurrence is $d_k (\log V)^{(k+1)k/2} = V (\log V)^{(k+1)k/2}$ (in the worst case), and that the sum of these terms is minimized by setting each of theses terms to the k-th root of the product. (Note that this also specifies how to calculate depth d_{i-1} from depth d_i.) Minimizing, we get: $\rho_k \leq k r^{1/k} (\log V)^{(k+1)/2} + O(\log^k V)$. Choosing $k = \left(\frac{2 \log V}{\log \log V}\right)^{1/2}$ gives us $\rho_k = 2^{O(\sqrt{\log V \log \log V})}$, and thus C_k is at most $V 2^{O(\sqrt{\log V \log \log V})}$, 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. □

References

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

[2] 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.

[3] E. Bar-Eli, P. Berman, A. Fiat, and P. Yan. On-line navigation in a room. In *Symposium on Discrete Algorithms*, pages 237–249, 1992.

[4] R. Behringer and S. Hötzel. Simultaneous estimation of pitch angle and lane width from the video image of a marked road. In *IEEE/RSJ/GI International Conference on Intelligent Robots and Systems*, September 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/RSJ/GI International Conference on Intelligent Robots and Systems*, September 1994. Also published as Technical Report SCR-94-TR-474, Siemens Corporate Research.

[8] 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, vol. 18, 2/3, pp. 1–24, February 1995.).

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

[10] Avrim Blum, Prabhakar Raghavan, and Baruch Schieber. Navigating in unfamiliar geometric terrain. In *Proceedings of Twenty-Third ACM Symposium on Theory of Computing*, pages 494–504. ACM, 1991.

[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. In *Proceedings of the 32nd Symposium on Foundations of Computer Science*, pages 298–303. IEEE, 1991.

[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] John Evans. HelpMate^{TM}: An autonomous mobile robot courier for hospitals. In *IEEE/RSJ/GI International Conference on Intelligent Robots and Systems*, September 1994.

[16] Rolf Klein. Walking an unknown street with bounded detour. *The 32nd Symposium on Foundations of Computer Science*, pages 304–313, October 1991.

[17] Jon M. Kleinberg. The localization problem for mobile robots. In *Proceedings of the Thirty-Fifth Annual Symposium on Foundations of Computer Science*, pages 521–531, May 1994.

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

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

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

[21] 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.

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