Piecemeal Learning of an Unknown Environment

(Extended Abstract) *

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

Abstract

We introduce a new learning problem: learning a graph by *piecemeal search*, in which the learner must return every so often to its starting point (for refueling, say). We present two linear-time piecemeal-search algorithms for learning *city-block graphs*: grid graphs with rectangular obstacles.

1 Introduction

We address the situation where a learner, to perform a task better, must learn a complete map of its environment. For example, the learner might be a security guard robot, a taxi driver, or a trail guide.

Exploration of unknown environments has been addressed by many previous authors, such as Papadimitriou and Yanakakis [7], Blum, Raghavan, and Schieber [2], Rivest and Schapire [8], Deng and Papadimitriou [4], Betke [1], Deng, Kameda, and Papadimitriou [3], and Bar-Eli, Berman, Fiat, and Yan [5].

This paper considers a new constraint: for some reason learning must be done "piecemeal"– that is, a little at a time. For example, a rookie taxi driver might learn a city bit by bit while returning to base between trips. A planetary exploration robot might need to return to base camp periodically to refuel, to return collected samples, to avoid nightfall, or to perform some other task. A tourist can explore a new section of Rome each day before returning to her hotel.

The "piecemeal constraint" means that *each of the learner's exploration phases must be of limited duration.* Between exploration phases the learner might perform other unspecified tasks. We assume that *each exploration phase starts and ends at a fixed start position s.* The piecemeal constraint can make efficient exploration surprisingly difficult. This paper gives our preliminary results on the general problem: two linear-time algorithms for the piecemeal search of grid graphs with rectangular obstacles.

2 The formal model

We model the learner's environment 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 learner can move from x to y, or back, in a single step. We assume that vertices are points in the plane, that edges are straight-line segments, and that G is planar (no edges cross).

At any vertex the learner can sense only its global position and the directions of the incident edges; it has no vision or long-range sensors.

The learner only incurs a cost for traversing edges; thinking and planning (computation) are free. The learner is given an upper bound B on the number of steps it can make (edges it can

*We gratefully acknowledge support from NSF grant CCR-8914428, ARO grant DAAL03-86-K-0171, NSF grant 9217041-ASC, and the Siemens Corporation. The authors can be reached at margrit@theory.lcs.mit.edu, rivest@theory.lcs.mit.edu, and mona@theory.lcs.mit.edu.

traverse) in one exploration phase. We assume B suffices for at least *two* round trips between s and any other single vertex in G; B is at least four times the radius r of the graph. (Actually, $1 + \epsilon$ round trips suffice.) Initially all the learner knows is its starting vertex s and the bound B. The learner's goal is to explore the entire graph: to visit every vertex and traverse every edge, minimizing the total number of edges traversed.

This paper focuses on *city-block graphs*: grid graphs containing some non-touching axis-parallel rectangular obstacles. Figure 1 gives an example. These graphs are also studied by Papadimitriou and Yanakakis [7], Blum, Raghavan, and Schieber [2], and Bar-Eli, Berman, Fiat and Yan [5]. An $m \times n$ city-block graph with no obstacles has exactly mn vertices (at points (i, j) for $1 \leq i \leq m, 1 \leq j \leq n$) and $2mn-(m+n)$ edges (between points at distance 1 from each other). Obstacles, if present, decrease the number of accessible locations (vertices) and edges in the city-block graph.

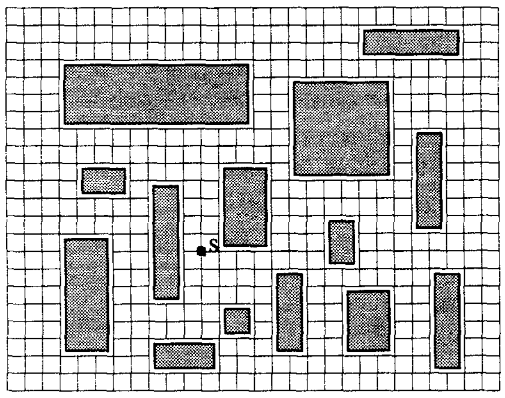

Figure 1: A city-block graph.

3 General remarks and basic observations

A simple approach is to use a known ordinary search algorithm—breadth-first search or depth-first search—and just interrupt the search as needed to return to visit s. This approach fails.

DFS is efficient, since it is linear in the number of edges in the graph, but it does not guarantee (efficient) returns to s. That is, it may be that for some vertex the best return path to s in the explored graph has length larger than B.

BFS, on the other hand, always enables efficient returns to s, since it always knows a shortest path back to s via explored edges. However, it fails to be efficient since it explores all the vertices at the same distance from s before exploring any vertices which are further away from s. In Figure 2 we give an example of a graph in which vertices of the same shortest path distance from s are far away from each other. For such graphs the cost of relocating between vertices can make the overall cost of BFS quadratic in the number of edges in the graph.

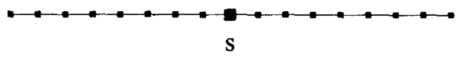

Figure 2: A simple graph for which the cost of BFS is quadratic in the number of edges.

3.1 Off-line piecemeal search

The *off-line* piecemeal search problem, where the learner knows the graph initially, is similar to the well-known *Chinese Postman problem* [6], but where the postman must return to the post-office every so often. (Our off-line problem is the *Weak Postman problem*, for postmen who cannot carry much mail.) The same problem arises when many postmen must cover the same city with their routes.

The simple "interrupted-DFS" approach provides a good approximation algorithm for the off-line piecemeal search problem. To traverse every edge, the learner does a DFS of the entire graph, and returns to s after every $B/2$ steps in the DFS algorithm using shortest paths (computed using its initial knowledge of G). We do not know an optimal off-line algorithm; this may be an NP-hard problem. The following theorem and proof show that we can find a good approximate solution for the off-line problem.

Theorem 1 *There exists an approximate solution to the off-line piecemeal search problem for an arbitrary undirected graph $G = (V, E)$ which traverses at most $O(|E|)$ edges.*

Proof: Assume that the radius of the graph is r and the explorer is allowed to traverse $B > 4r$ edges during each phase of the exploration. The explorer looks at the map of the region to be explored, and (mentally) does a depth first search of the graph. The path defined by traversing the entire depth first tree has length $2|E|$. The explorer then segments this path into pieces of length $2r$. The piecemeal exploration then consists of taking a shortest path from s to the start of each segment, traversing the edges in the segment, and taking a shortest path back to the start vertex. For each segment, the robot traverses at most $2r$ edges to get to and from the segment. Since there are at most $\lceil \frac{2|E|}{2r} \rceil$ segments, the number of edge traversals due to interruptions is:

$$\begin{aligned} \left\lceil \frac{|E|}{r} \right\rceil 2r &\leq \left(\frac{|E|}{r} + 1 \right) 2r \\ &\leq 2|E| + 2r \\ &\leq 4|E| \end{aligned}$$

Thus the off-line piecemeal search problem for an arbitrary undirected graph can be solved so as to traverse at most $6|E|$ steps. □

3.2 On-line piecemeal search

We can use the strategy outlined above to obtain an efficient on-line piecemeal search algorithm. We will refer to a search as *compact* if it always knows a shortest path via explored edges back to s. We will refer to a search as *approximately compact* if it always knows a path back to s via explored edges of length at most the radius of the graph.

Using the strategy described in the proof of Theorem 1, we can interrupt an ordinary search algorithm that is both *efficient* (time $O(E)$) and approximately compact to obtain an efficient piecemeal search algorithm.

Theorem 2 *An efficient, approximately compact algorithm for searching an undirected graph can be transformed into an efficient piecemeal search algorithm.* □

For arbitrary undirected planar graphs, we can show that any compact search algorithm requires $\Omega(|E|^2)$ edge traversals. For example, exploring the graph in Figure 2 (known initially only to be an arbitrary undirected planar graph) would result in $|E|^2$ edge traversals if the search is required to be compact. For city-block graphs, however, we present two efficient $O(|E|)$ compact search algorithms. Since a compact search algorithm is also an approximately compact search algorithm, these two algorithms give efficient piecemeal search algorithms for city-block graphs. The *wavefront algorithm* is based on BFS, but overcomes the problem of relocation cost. The *ray algorithm* is a variant of DFS that always knows a shortest path back to s. First, however, we develop some properties of shortest paths in city-block graphs, based on an analysis of BFS.

4 Shortest paths in city-block graphs

A compact algorithm maintains at all times knowledge of a shortest path back to s. Since BFS is compact, we study BFS in some detail to understand the characteristics of shortest paths in city-block graphs. Also, our wavefront algorithm is a modification of BFS. Figure 3 illustrates the operation of BFS. Our algorithms depend on the special properties that shortest paths have in city-block graphs.

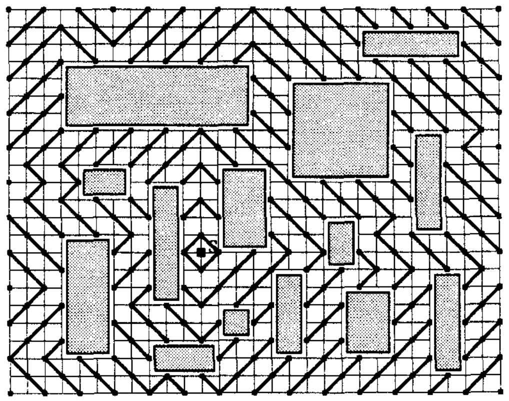

Figure 3: Environment explored by breath-first search, showing only "wavefronts" at odd distance to s.

Let $\delta(v, v')$ denote the length of the shortest path between v and v', and let $d[v]$ denote $\delta(v, s)$, the length of the shortest path from v back to s.

4.1 Monotone paths and the four-way decomposition

A city-block graph can be usefully divided into four regions (north, south, east, and west) by four monotone paths: an east-north path, an east-south path, a west-north path, and a west-south path. The east-north path starts from s, proceeds east until it hits an obstacle, then proceeds north until it hits an obstacle, and so on. The other paths are similar (see Figure 4).

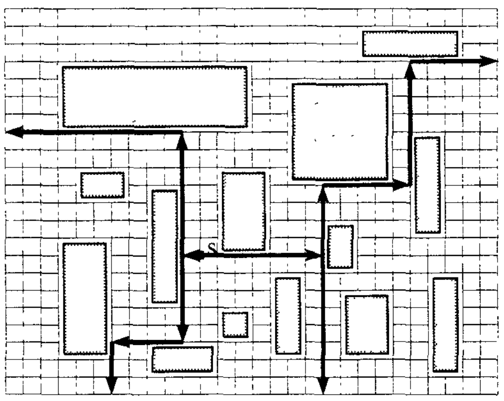

Figure 4: The four monotone paths and the four regions.

It is easy to show that monotone paths are shortest paths, and that for any vertex, there is a shortest path to s through only one region. Without loss of generality, we therefore only consider compact search algorithms that divide the graph into these four regions, and search these regions separately. In this paper, we only discuss what happens in the northern region; the other regions are similar. We also assume that each obstacle has a unique marker; that is, if the explorer hits an obstacle twice, it knows that it is at the same obstacle. Once the explorer hits the obstacle, it also knows the dimensions of the obstacle. (We do not need these assumptions, but use them to simplify the exposition of our algorithm.)

4.2 BFS and wavefronts

A BFS can be viewed as exploring the graph in waves that expand outward from s, much as waves expand from a pebble thrown into a pond. Figure 3 illustrates the wavefronts that can arise.

A *wavefront* w can then be defined as an ordered list of explored vertices $\langle v_1, v_2, \ldots, v_m \rangle$, such that $d[v_i] = d[v_1]$ for all i, and such that $\delta(v_i, v_{i+1}) = 2$ for all i. We call $d[w] = d[v_1]$ the distance of the wavefront. In a city-block graph wavefronts always consist of a sequence of diagonally adjacent vertices.

There is a natural "successor" relationship between BFS wavefronts, as a wavefront at distance t generates a successor at distance $t+1$. We informally consider a *wave* to be a sequence of successive wavefronts. Because of obstacles, however, a wave may split (if it hits an obstacle) or merge (with another wave, on the far side of an obstacle). Two wavefronts that merge are called *sibling* wavefronts, and the point on an obstacle where they first meet is called the *meeting point* of the two wavefronts. In the northern region, meeting points are always on the north side of obstacles, and each obstacle has exactly one meeting point on its northern side. See Figure 5.

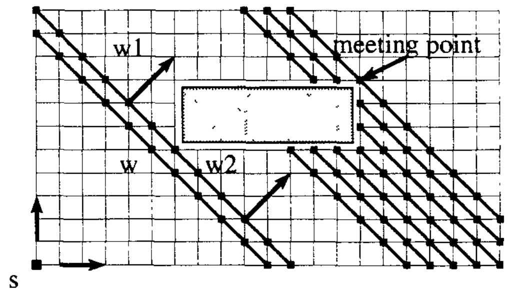

Figure 5: Splitting and merging of wavefronts along a corner of an obstacle. Illustration of meeting point and sibling wavefronts.

The shape of a wave stays more or less constant, unless the wave splits, merges with another wave, or curls around the back edge of an obstacle.

In the northern region, the *front* of an obstacle is its southern side, the *back* of an obstacle is its northern side, and the *sides* of an obstacle are its east and west sides. A wave always hits the front of an obstacle first. Consider the shape of a wave before it hits an obstacle and its shape after it passes the obstacle. If a peak of the wavefront hits the obstacle, this peak will not be part of the shape of the wavefront after it "passes" the obstacle. Instead, this wavefront may have one or two new peaks which have the same x-coordinates as the sides of the obstacle (see Figure 6).

The above properties of the shapes of wavefronts in the northern region can be shown by induction on the distance of the wavefronts.

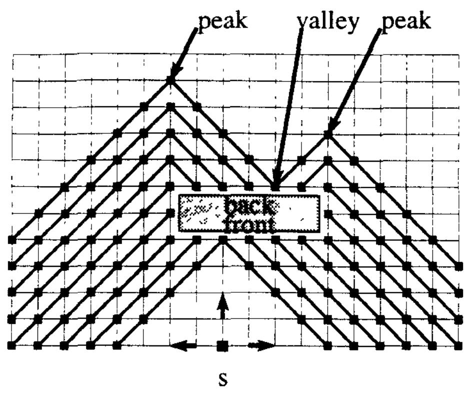

Figure 6: Shapes of wavefronts. Illustration of peaks and valleys, and front and back of an obstacle. The meeting point is the lowest point in the valley.

4.3 Canonical shortest paths and meeting points

We now make a fundamental observation on the nature of shortest paths (in the northern region) from a vertex v back to s. Without loss of generality, this path goes south whenever possible. If the path hits the north side of an obstacle, it goes east if it hits east of the meeting point for that obstacle. Similarly, it goes west if it hits on or west of the meeting point. Once it reaches a back corner of the obstacle, it then continues south. If it hits a monotone path, it follows that back to s.

Meeting points are thus required to compute canonical shortest paths; we now give a way for computing meeting points. Let v_w and v_e denote the front-west and front-east corners of an obstacle. The canonical shortest path from any vertex on the sides or back of the obstacle will go through one of v_w or v_e, whichever yields the shortest path overall. A shortest path from a meeting point to s can go either way, for the same overall distance. Thus, we can determine the meeting point on an obstacle if the distances $d[v_w]$ and $d[v_e]$ are both known.

5 The wavefront algorithm

The wavefront algorithm, presented in this section, mimics BFS in that it computes exactly the same set of wavefronts. However, in order to minimize relocation costs, the wavefronts may be computed in a different order. Rather than computing all the wavefronts at distance t before computing any wavefronts at distance $t + 1$ (as BFS does), the wavefront algorithm will continue to follow a particular wave persistently, before it relocates and pushes another wave along.

We define *expanding* a wavefront $w = \langle v_1, v_2, \ldots, v_m \rangle$ as computing a set of zero or more successor wavefronts by looking at the set of all unexplored vertices at distance one from any vertex in w. Every vertex v in a successor wavefront has $d[v] = d[w] + 1$. It is easy to argue that a wavefront of m vertices can be expanded in time $O(m)$.

There is really only one incorrect way to expand a wavefront and get something other than what BFS obtained as a successor: to expand a wavefront that is touching a meeting point before its sibling wavefront has merged with it. Operationally, this means that the wavefront algorithm is blocked in the following two situations: (a) it cannot expand a wavefront from the side around to the back of an obstacle before the meeting point for that obstacle has been set (see Figure 7), and (b) it cannot expand a wavefront that touches a meeting point until its sibling has arrived there as well (see Figure 8). A wavefront w_2 *blocks* a wavefront w_1 if w_2 must be expanded before w_1 can be safely expanded. We also say w_2 and w_1 interfere.

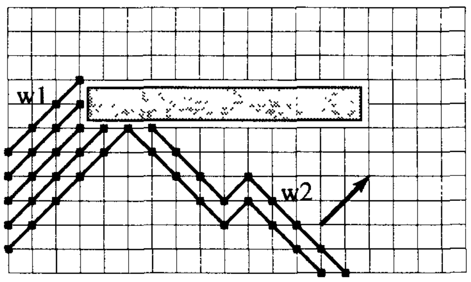

Figure 7: Blockage of w_1 by w_2. Wavefront w_1 has finished covering one side of the obstacle and the meeting point is not set yet.

A wavefront w is an *expiring* wavefront if its descendant wavefronts can never interfere with the expansion of any other wavefronts that now exist or any of their descendants. For example, a wavefront w is an expiring wavefront if its endpoints are both on the front of the same obstacle; w will expand into the region surrounded by the wavefront and the obstacle, and then disappear or "expire."

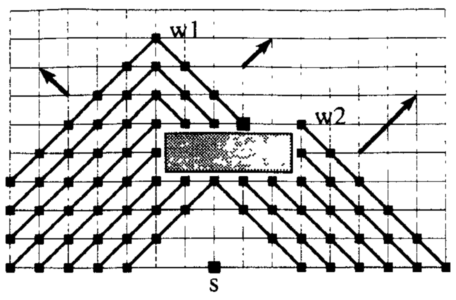

Figure 8: Blockage of w_1 by w_2. Wavefront w_1 has reached the meeting point on the obstacle, but the sibling wavefront w_2 has not.

We say that a wavefront expires if it consists of just one vertex with no unexplored neighbors. The area delineated by an expiring wavefront can be explored independent of the rest of the region. This area may contain obstacles.

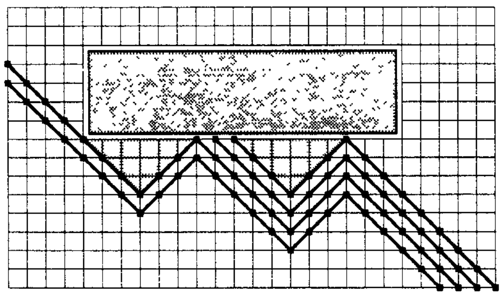

Figure 9: Triangular areas (shaded) delineated by two expiring wavefronts.

Procedure WAVEFRONT-ALGORITHM is an efficient compact search algorithm that can be used to create an efficient piecemeal search algorithm. It repeatedly expands one wavefront until it splits, merges, expires, or is blocked. The WAVEFRONT-ALGORITHM takes as an input a start point s and the boundary coordinates of the environment. It calls procedure CREATE-MONOTONE-PATHS to explore four monotone paths (see section 4.1) and define the four regions. Then procedure EXPLORE-AREA is called for each region.

```
WAVEFRONT-ALGORITHM($s$, boundary)
1 CREATE-MONOTONE-PATHS
2 for region = north, south, east, and west
3     initialize current wavefront $w \leftarrow \langle s \rangle$
4     EXPLORE-AREA($w$, region)
```

For each region we keep an ordered list L of non-expiring wavefronts to be expanded. In the northern region, the wavefronts are ordered by their west-most x-coordinate. Each obstacle also has a list of the expiring wavefronts that hit it. We treat the boundaries as large obstacles. The north region has been fully explored when all the lists of wavefronts are empty.

Note that vertices on the monotone paths are considered initially to be unexplored, and that expanding a wavefront returns a successor that is entirely within the same region.

Each iteration of EXPLORE-AREA (see next page) expands a wavefront. When EXPAND is called on a wavefront w, the learner starts expanding w from its current location, which is a vertex at either one of the end points of wavefront w. It is convenient, however to think of EXPAND as finding the unexplored neighbors of the vertices in w in parallel.

Depending on what happens during the expansion, the successor wavefront can be split, merged, blocked, or may expire. (More than one of these cases may apply.) Procedures MERGE and SPLIT (see following pages) handle the (not necessarily disjoint) cases of merging and splitting wavefronts. Note that we use call-by-reference conventions for the wavefront w and the list L of wavefronts. When the RELOCATE(w) is called, the learner moves from its current location to the closest end point of w, via a shortest path in the explored area of the graph.

In line 15 of EXPLORE-AREA, the learner may stay at w, if it is unblocked. Otherwise, it relocates to the unblocked wavefront that is nearest to the current position as measured by distance within list L.

Wavefronts are merged when exploration continues around an obstacle. A wavefront can be merged with two wavefronts, one on each end. After a merge, the learner first checks if there is a list of expiring wavefronts for the obstacle where the merge occurred. If so, the learner explores the areas delineated by these wavefronts by calling procedure EXPIRE. The result of a merger can also be an expiring wavefront; in this case, the wavefront is placed on the ordered list of expiring wavefronts for the obstacle onto which it will expire (line 9 in MERGE).

```
EXPLORE-AREA($w$, region)
 1  initialize list of wavefronts $L \leftarrow \langle w \rangle$
 2  repeat
 3          EXPAND current wavefront $w$ to successor wavefront $w_s$
 4          current wavefront $w \leftarrow w_s$
 5          if $w$ is a single vertex with no unexplored neighboring vertices
 6             then remove $w$ from ordered list $L$ of wavefronts
 7             else replace $w$ by $w_s$ in ordered list $L$ of wavefronts
 8                   if both front corners of any obstacle(s) have been explored
 9                      then set meeting points for those obstacle(s)
10                   if $w$ can be merged with adjacent wavefront(s)
11                      then MERGE($w, L$, region)
12                   if $w$ hits obstacle(s)
13                      then SPLIT($w, L$, region)
14          if $L$ not empty
15             then $w \leftarrow$ unblocked wavefront nearest in $L$ to learner's location
16                   RELOCATE($w$)
17  until $L$ is empty
18  if boundary of area being explored has list $L_e$ of expiring wavefronts
19     then EXPIRE($L_e$, region)
```

```
SPLIT($w, L$, region)
1  split $w$ into appropriate wavefronts
       in order $w_0, \ldots, w_n$
2  remove $w$ from ordered list $L$ of wavefronts
3  for $i = 0$ to $n$
4      if $w_i$ is an expiring wavefront
5      then put $w_i$ on ordered list of expiring
               wavefronts of appropriate obstacle
6      else put $w_i$ on ordered list $L$
               of wavefronts
```

When procedure SPLIT is called on wavefront w, we note that the wavefront is either the successor wavefront, or the successor wavefront merged with some wavefront which is adjacent to it. Once wavefront w is split into $w_0, \ldots, w_n$, we classify each of these wavefronts as either expiring or non-expiring wavefronts.

```
EXPIRE($L_e$, region)
1  for every wavefront $w$ in list $L_e$ of
         expiring wavefronts
2      RELOCATE($w$)
3      EXPLORE-AREA($w$, region)
```

6 Correctness and analysis

The following theorems establish the correctness of our algorithm, and also show the running time. Details of the proofs are available in a full version of this paper.

Theorem 3 *The wavefront algorithm is a compact search algorithm for city-block graphs.*

Proof: In Theorem 4 we show by induction on shortest path length that the wavefront algorithm mimics breadth-first search. Thus it is compact. We show in Theorem 5 that the algorithm does not terminate until all vertices have been explored. Completeness follows. □

Theorem 4 *The algorithm* EXPLORE-AREA *expands the wavefronts so as to maintain compactness.*

Proof: This is shown by induction on the distance of the wavefronts. The key observations are (1) there is a canonical shortest path from any vertex v to s which goes south whenever possible, but east

```
MERGE($w$, $L$, region)
 1  remove $w$ from list $L$ of wavefronts
 2  while there is a sibling wavefront $w'$ with which $w$ can merge
 3       do if obstacle which $w$ and $w'$ are merging around
                  has a list $L_e$ of expiring wavefronts
 4             then EXPIRE($L_e$, region)
 5           remove $w'$ from list $L$ of wavefronts
 6           merge $w$ and $w'$ into wavefront $w''$
 7           $w \leftarrow w''$
 8  if $w$ is an expiring wavefront which does not split
 9     then put $w$ on ordered list of expiring wavefronts of appropriate obstacle
10     else  put $w$ in ordered list $L$ of wavefronts
```

or west around obstacles and (2) a wavefront is never expanded beyond a meeting point. □

Theorem 5 *There is always a wavefront which is not blocked.*

Proof: We consider exploration in the north region. First note that sibling wavefronts cannot block each other. Moreover, the east-most wavefront in the northern region cannot be blocked by anything to its east, and the west-most wavefront in the north region cannot be blocked by anything to its west. Thus the explorer can always "follow a chain of wavefronts" to either its east or west to find an unblocked wavefront. □

Theorem 6 *The wavefront algorithm is linear in the number of edges in the city-block graph.*

Proof: We show that the total number of edge traversals is no more than $10|E|$. Note that when the procedures CREATE-MONOTONE-PATHS, EXPAND, and RELOCATE are called, the learner actually walks through the environment. Thus, we count the edge traversals for each call of these procedures.

The learner traverses the edges on the monotone paths *once* when it explores them, and *once* to get back to the start point. This is clearly at most $2|E|$ edge traversals. The total number of edge traversals caused by procedure EXPAND is at most $2|E|$. We now only need to consider the edge traversals due to procedure RELOCATE.

Edges are traversed at most once due to relocations to the nearest wavefront when there is no blockage. The total number of edges traversed due to relocations after wavefronts have expired is at most $2|E|$. Essentially we have a linear sweep along the boundary of an obstacle for exploration of expiring wavefronts. We can limit the number of edge traversals along the boundary of the obstacle due to recursive calls when expiring wavefronts split.

Edges are traversed at most three times due to relocations after blockage. We know that proceeding along a "chain of wavefronts" leads the learner to an unblocked wavefront. The learner does not traverse, due to blockage, any wavefront in this chain again. Moreover, the learner does not move back and forth between wavefronts on different sides of an obstacle. The total number of edges traversed due to procedure RELOCATE is at most $6|E|$. □

Theorem 7 *A piecemeal algorithm based on the wavefront algorithm runs in time linear in the number of edges in the city-block graph.*

Proof: This follows immediately from Theorem 2, Theorem 3 and Theorem 6. □

7 Ray algorithm

We now give another efficient compact search algorithm, called the *ray algorithm.* This thus yields another efficient piecemeal algorithm for searching a city-block graph. This algorithm is simpler than the wavefront algorithm, but seems less suitable for generalization.

The ray algorithm also starts by finding the four monotone paths, and splitting the graph into four regions to be searched separately. The algorithm explores in a manner similar to depth-first search, but maintains compactness by exploring primarily in a northern direction (in the northern region), with east/west excursions limited to one step, except for at the back side of obstacles. That is, the ray algorithm greedily explores just one north-south *ray* as far as it knows that compactness is maintained.

Rays start on one of the monotone paths, proceeding north until an obstacle is reached. Then the learner backtracks to the monotone path and starts exploring a neighboring ray. Once the two front corners of an obstacle are explored, the shortest paths to the vertices at the back of an obstacle are known; the "meeting point" is then determined. Once the meeting point for an obstacle is known, the learner explores the vertices at the back of the obstacle, and continues exploration by following rays from the back of the obstacle. (Note that not all paths to s in the "search tree" defined by the ray algorithm are shortest paths; the tree path may go one way around an obstacle while the algorithm knows that the shortest path goes the other way around.)

Theorem 8 *The ray algorithm is a linear-time compact search algorithm that can be transformed into a linear-time piecemeal search of a city-block graph.* □

8 Conclusions

We have presented efficient algorithms for the piecemeal search of city-block graphs. We leave as open problems finding algorithms for the piecemeal search of:

- grid graphs with non-convex obstacles,
- other tesselations, such as triangular tesselations with triangular obstacles, and
- more general classes of graphs, such as the class of planar graphs.

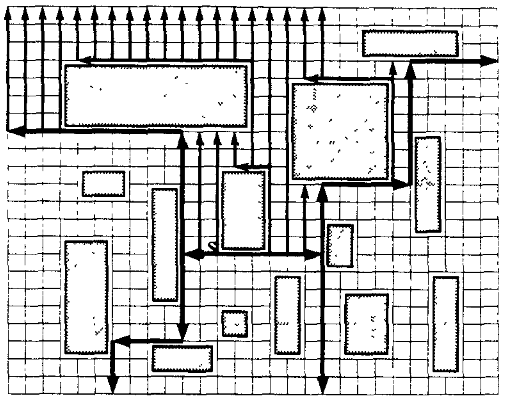

Figure 10: Operation of the ray algorithm.

References

[1] 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)).

[2] 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.

[3] 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.

[4] 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.

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

[6] Jack Edmonds and Ellis L. Johnson. Matching, Euler tours and the Chinese Postman. *Mathematical Programming*, 5:88–124, 1973.

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

[8] Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. In *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, pages 411–420, Seattle, Washington, May 1989. ACM. (To appear in *Information and Computation*).