MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY
and
CENTER FOR BIOLOGICAL AND COMPUTATIONAL LEARNING
DEPARTMENT OF BRAIN AND COGNITIVE SCIENCES

A.I. Memo No. 1474
C.B.C.L. Memo No. 93

March, 1994

Piecemeal Learning of an Unknown Environment

Margrit Betke **Ronald L. Rivest** **Mona Singh**

This publication can be retrieved by anonymous ftp to publications.ai.mit.edu.

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.

This report describes research done at the Center for Biological and Computational Learning and the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Center is provided in part by a grant from the National Science Foundation under contract ASC–9217041.
The authors were also supported by NSF grant CCR - 8914428, NSF grant CCR - 9310888 and the Siemens Corporation.

The authors can be reached at margrit@ai.mit.edu, rivest@theory.lcs.mit.edu, and mona@theory.lcs.mit.edu.

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 [1], Blum, Raghavan, and Schieber [2], Rivest and Schapire [4], Deng and Papadimitriou [5], Betke [6], Deng, Kameda, and Papadimitriou [7], Rao, Kareti, Shi and Iyengar [11], and Bar-Eli, Berman, Fiat, and Yan [8].

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.* We assume that *each exploration phase starts and ends at a fixed start position s.* This special location might be the airport (for a taxi driver), a refueling station, a base camp, or a trailhead. Between exploration phases the learner might perform other unspecified tasks (for example, a taxi driver might pick up a passenger). Piecemeal learning thus enables "learning on the job", since the phases of piecemeal learning can help the learner improve its performance on the other tasks it performs. This is the "exploration/exploitation tradeoff": spending some time exploring (learning) and some time exploiting what one has learned.

The piecemeal constraint can make efficient exploration surprisingly difficult. This paper presents our preliminary results on piecemeal learning of arbitrary undirected graphs and gives two linear-time algorithms for the piecemeal search of grid graphs with rectangular obstacles. The first algorithm, the "wavefront" algorithm, can be viewed as an optimization of breadth-first search for our problem. The second algorithm, the "ray" algorithm, can be viewed as a variation on depth-first search. Although the ray algorithm is simpler, the wavefront algorithm may prove a more fruitful foundation for generalization to more complicated graphs.

We now give a brief summary of the rest of the paper. Section 2 gives the formal model and introduces city-block graphs. Section 3 discusses piecemeal search on arbitrary graphs and gives an approximate solution to the off-line version of this problem. Section 4 discusses shortest paths in city-block graphs. Section 5 introduces the notion of a wavefront, gives the wavefront algorithm for piecemeal search of city-block graphs, proves it correct, and derives its running time. Section 6 introduces the ray algorithm as another way to do piecemeal search of city-block graphs. Section 7 concludes with some open problems.

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 the learner can always recognize a previously visited vertex; it never confuses distinct locations. At any vertex the learner can sense only the edges incident to it; it has no vision or long-range sensors. It can also distinguish between incident edges at any vertex. Without loss of generality, we can assume that the edges are ordered. At a vertex, the learner knows which edges it has traversed already. The learner only incurs a cost for traversing edges; thinking (computation) is free. We also assume a uniform cost for an edge traversal.

The learner 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 learner 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 (the maximum of all shortest-path distances between s and any vertex in G).

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.

2.1 City-block graphs

We model environments such as cities or office buildings in which efficient on-line robot navigation may be needed. We focus on grid graphs containing some non-touching axis-parallel rectangular "obstacles". We call these graphs *city-block graphs.* They are rectangular planar graphs in which all edges are either vertical (north-south) or horizontal (east-west), and in which all faces (city blocks) are axis-parallel rectangles whose opposing sides have the same number of edges. A 1×1 face might correspond to a standard city block; larger faces might correspond to obstacles (parks or shopping malls). Figure 1 gives an example. City-block graphs are also studied by Papadimitriou and Yanakakis [1], Blum, Raghavan, and Schieber [2], and Bar-Eli, Berman, Fiat and Yan [8].

An $m \times n$ city-block graph with no obstacles has exactly mn vertices (at points (i, j) for $1 \le i \le m$, $1 \le j \le n$) 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. In city-block graphs the vertices and edges are deleted such that all remaining faces are rectangles.

We assume that the directions of incident edges are apparent to the learner.

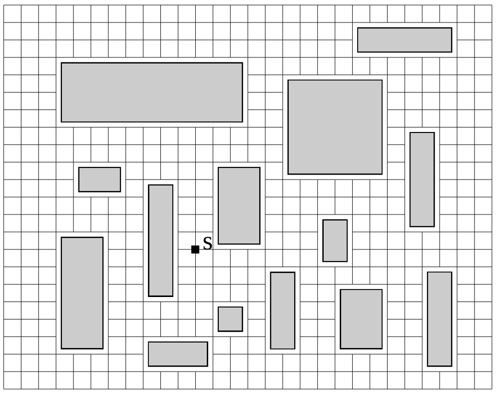

Figure 1: A city-block graph with distinguished start vertex s.

3 Piecemeal search on general graphs

In this section, we discuss piecemeal search on general graphs. In particular, we show why "standard" approaches to this problem do not work. We also define the off-line version of this problem, and give an approximate solution for it. Finally, we give a general method for converting certain types of search algorithms into piecemeal search algorithms.

3.1 Initial approaches using DFS and BFS

A simple approach to piecemeal search on arbitrary undirected graphs is to use an ordinary search algorithm—breadth-first search (BFS) or depth-first search (DFS)—and just interrupt the search as needed to return to visit s. (Detailed descriptions of BFS and DFS can be found in algorithms textbooks [3].) Once the learner has returned to s, it goes back to the vertex at which search was interrupted and resumes exploration.

In depth-first search, edges are explored out of the most recently discovered vertex v that still has unexplored edges leaving it. When all of v's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which v was discovered. This process continues until all edges are explored. This search strategy, without interruptions due to the piecemeal constraint, is efficient since at most $2|E|$ edges are traversed. Interruptions, or exploration in phases of limited duration, complicate matters. For example, suppose in the first phase of exploration, at step $B/2$ of a phase the learner reaches a vertex v as illustrated in Figure 2. Moreover, suppose that the only path the learner knows from s to v has length $B/2$. At this point, the learner must stop exploration and go back to the start location s. In the second phase, in order for the learner to resume a depth-first search, it should go back to v, the most recently discovered vertex. However, since the learner only knows a path of $B/2$ to v, it cannot proceed with exploration from that point. Other variations of DFS that we have looked at seem to suffer from the same problem.

On the other hand, breadth-first search with interruptions does guarantee that all vertices in the graph are ultimately explored. Whereas a DFS strategy cannot resume exploration at vertices to which it only knows

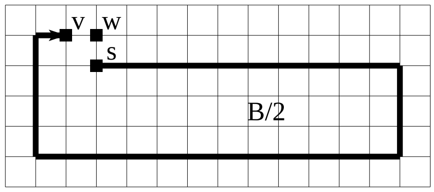

Figure 2: The learner reaches vertex v after $B/2$ steps in a depth-first search. Then it must interrupt its search and return to s. It cannot resume exploration at v to get to vertex w, because the known return path is longer than $B/2$, the remaining number of steps allowed in this exploration phase. DFS fails.

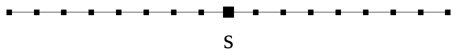

Figure 3: A simple graph for which the cost of BFS is quadratic in the number of edges.

a long path, a BFS strategy can always resume exploration. This is because BFS ensures that the learner always knows a *shortest* path from s to any explored vertex. However, since a BFS strategy explores all the vertices at the same distance from s before exploring any vertices that are further away from s, the resulting algorithm may not be efficient. Note that in the usual BFS model, the algorithm uses a queue to keep track of which vertex it will search from next. Thus, searching requires extracting a vertex from this queue. In our model, however, since the learner can only search from its current location, extracting a vertex from this queue results in a *relocation* from the learner's current location to the location of the new vertex. In Figure 3 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.

3.2 Off-line piecemeal search

We now develop a strategy for the off-line piecemeal search problem which we can adapt to get a strategy for the on-line piecemeal search problem.

In the *off-line* piecemeal search problem, the learner is a given a finite connected undirected graph $G = (V, E)$, a start location $s \in V$, and a bound B on the number of edges traversed in any exploration phase. The learner's goal is to plan an optimal search of the graph that visits every vertex and traverses every edge, and also satisfies the piecemeal constraint (i.e., each exploration phase traverses at most B edges and starts and ends at the start location).

The off-line piecemeal search problem is similar to the well-known *Chinese Postman Problem* [9], but where the postman must return to the post-office every so often. (We could call the off-line problem 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 Chinese Postman Problem can be solved by a polynomial time algorithm if the graph is either undirected or directed [9]. The Chinese Postman problem for a mixed graph that has undirected and directed edges was shown to be NP-complete by Papadimitriou [10]. We do not know an optimal off-line algorithm for the Weak Postman Problem; this may be an NP-hard problem. This is an interesting open problem.

We now give an approximation algorithm for the off-line piecemeal search problem using a simple "interrupted-DFS" approach.

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

Proof: Assume that the radius of the graph is r and that the number of edges the learner 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. Before the learner starts traversing any edges in the graph, it looks at the graph to be explored, and computes a depth-first search tree of the graph. A depth first traversal of this depth-first search tree defines a path of length $2|E|$ which starts and ends at s and which goes through every vertex and edge in the graph. The learner breaks this path into segments of length αr. The learner also computes (off-line) a shortest path from s to the start of each segment.

The learner then starts the piecemeal exploration of the graph. Each phase of the exploration consists of taking a shortest path from s to the start of a segment, traversing the edges in the segment, and taking a shortest path back to the start vertex. For each segment, the learner traverses at most $2r$ edges to get to and from the segment. Since there are $\lceil \frac{2|E|}{\alpha r} \rceil$ segments, there are $\lceil \frac{2|E|}{\alpha r} \rceil - 1$ interruptions, and the number of edge traversals due to interruptions is at most:

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

Thus the total number of edge traversals is at most $(4/\alpha + 2)|E| = O(E)$. □

3.3 On-line piecemeal search

We now show how we can change the strategy outlined above to obtain an efficient on-line piecemeal search algorithm.

We call an on-line search *optimally interruptible* if it always knows a shortest path via explored edges back to s. We refer to a search as *efficiently interruptible* if it always knows a path back to s via explored edges of length at most the radius of the graph. We say a search algorithm is a *linear time* algorithm if the learner traverses $O(E)$ edges during the search.

Theorem 2 *An efficiently interruptible, linear time algorithm for searching an undirected graph can be transformed into a linear-time piecemeal search algorithm.*

Proof: The proof of this theorem is similar to the proof of Theorem 1. However, there are a few differences. Instead of using an ordinary search algorithm (like DFS) and interrupting as needed to return to s, we use an efficiently interruptible, linear time search algorithm. Moreover, the search is on-line and is being interrrupted during exploration. Finally, the cost of the search is not $2|E|$ as in DFS, but at most $c|E|$ for some constant c.

Assume that the radius of the graph is r and that the number of edges the learner 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. Since the search algorithm is efficient, the length of the path defined by the search algorithm is at most $c|E|$, for some constant c, $c > 0$. In each exploration phase, the learner will execute αr steps of the original search algorithm. At the beginning of each phase the learner goes to the appropriate vertex to resume exploration. Then the learner traverses αr edges as determined by the original search algorithm, and finally the learner returns to s. Since the search algorithm is efficiently interruptible, the learner knows a path of distance at most r from s to any vertex in the graph. Thus the learner traverses at most $2r + \alpha r = B$ edges during any exploration phase.

Since there are $\lceil \frac{c|E|}{\alpha r} \rceil$ segments, there are $\lceil \frac{c|E|}{\alpha r} \rceil - 1$ interruptions, and the number of edge traversals due to interruptions is:

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

Thus, the total number of edge traversals is $|E|\,(2c/\alpha + c) = O(E)$. □

For arbitrary undirected planar graphs, we can show that any optimally interruptible search algorithm requires $\Omega(|E|^2)$ edge traversals in the worst case. For example, exploring the graph in Figure 3 (known initially only to be an arbitrary undirected planar graph) would result in $|E|^2$ edge traversals if the search is required to be optimally interruptible.

For city-block graphs, however, we present two efficient $O(|E|)$ optimally interruptible search algorithms. Since an optimally interruptible search algorithm is also an efficiently interruptible 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

An optimally interruptible algorithm maintains at all times knowledge of a shortest path back to s. Since

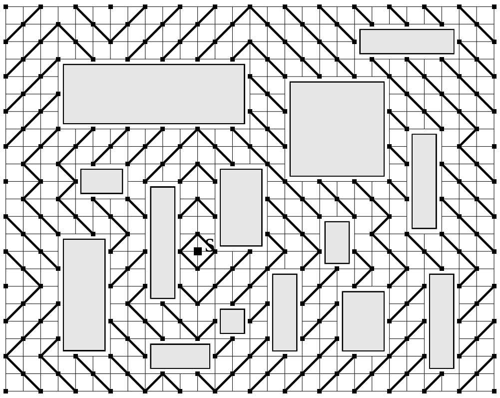

Figure 4: Environment explored by breath-first search, showing only "wavefronts" at odd distance to s.

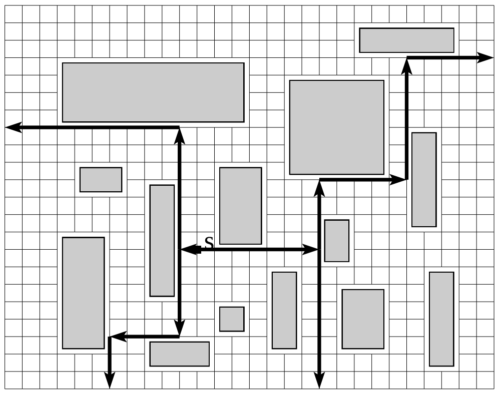

Figure 5: The four monotone paths and the four regions.

BFS is optimally interruptible, 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 4 illustrates the operation of BFS. Our algorithms depend on the special properties that shortest paths have in city-block graphs.

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, then turns and proceeds east again, and so on. The other paths are similar (see Figure 5). Note that all monotone paths are shortest paths. Furthermore, note that s is included in all four regions, and that each of the four monotone paths (east-north, east-south, west-north, west-south) is part of all regions to which it is adjacent.

In Lemma 1 we show that for any vertex, there is a shortest path to s through only one region. Without loss of generality, we therefore only consider optimally interruptible 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 handled similarly.

Lemma 1 *There exists a shortest path from s to any point in a region that only goes through that region.*

Proof: Consider a point v in some region A. Let p be any shortest path from s to the point v. If p is not entirely contained in region A, we can construct another path p' that is entirely contained in region A. We note that the vertices and edges which make up the monotone paths surrounding a region A are considered to be part of that region.

Since path p starts and ends in region A but is not entirely contained in region A, there must be a point u that is on p and also on one of the monotone paths bordering A. Note that u may be the same as v. Without loss of generality, let u be the last such point, so that the portion of the path from u to v is contained entirely within region A. Then the path p' will consist of the shortest path from s to u along the monotone path that u is on, followed by the portion of p from u to v. This path p' is a shortest path from s to v because p was a shortest path and p' can be no longer than p. $\square$

4.2 Canonical shortest paths of city-block graphs

We now make a fundamental observation on the nature of shortest paths from a vertex v back to s. In this section, we consider shortest paths in the northern region; properties of shortest paths in other region are similar.

Lemma 2 *For any vertex v in the northern region, there is a canonical shortest path from v to the start vertex s which goes south whenever possible. The canonical shortest path goes east or west only when it is prevented from going south by an obstacle or by the monotone path defining the northern region.*

Proof: We call the length $d[v]$ of the shortest path from v to s the *depth* of vertex v. We show this lemma by induction on the depth of a vertex.

For the base case, it is easy to verify that any vertex v such that $d[v] = 1$ has a canonical shortest path that goes south whenever possible.

For the inductive hypothesis, we assume that the lemma is true for all vertices that have depth $t-1$, and we want to show it is true for all vertices that have depth t. Consider a vertex p at depth t. If there is an obstacle obstructing the vertex that is south of point p or if p is on a horizontal segment of the mononotone path defining the northern region, then it is impossible for the canonical shortest path to go south, and the claim holds. Thus, assume the point south of p is not obstructed by an obstacle or by the monotone path defining the northern region. Then we have the following cases:

Case 1: Vertex p_s directly south of p has depth $t-1$. In this case, there is clearly a canonical shortest path from p to s which goes south from p to p_s and then follows the canonical shortest path of p_s, which we know exists by the inductive assumption.

Case 2: Vertex p_s directly south of p has depth not equal to $t-1$. Then one of the remaining adjacent vertices must have depth $t-1$ (otherwise it is impossible for p to have depth t). Furthermore, none of these vertices has depth less than $t-1$, for otherwise vertex p would have depth less than t.

Note that the point directly north of p cannot have depth $t-1$. If it did, then by the inductive hypothesis, it has a canonical shortest path which goes south. But then p has depth $t-2$, which is a contradiction.

Thus, either the point west of p or the point east of p has depth $t-1$. Without loss of generality, assume that the point p_w west of p has depth $t-1$. We consider two subcases. In case (a), there is a path of length 2 from p_w to p_s that goes south one step from p_w, and then goes east to p_s. In case (b), there is no such path.

Case (a): If there is such a path, the vertex directly south of p_w exists, and by the inductive hypothesis has depth $t-2$ (since there is a canonical shortest path from p_w to s of length $t-1$, the vertex directly to the south of p_w has depth $t-2$). Then p_s, which is directly east of this point, has depth at most $t-1$ and thus there is a canonical path from p to s which goes south whenever possible.

Case (b): Note that the only way there does not exist a path of length 2 from p_w to p_s (other than the obvious one through p) is if p is a vertex on the northeast corner of an obstacle which is bigger than 1×1. Suppose the obstacle is $k_1 \times k_2$, where k_1 is the length of the north (and south) side of the obstacle, and k_2 is the length of the east (and west) side of the obstacle. We know by the inductive hypothesis that the canonical shortest path from p_w goes either east or west along the north side of this obstacle, and since the vertex p has depth t we know that the canonical shortest path goes west. After having reached the corner, the canonical shortest path from p_w to s proceeds south. Thus, the vertex which is on the southwest corner of this obstacle has depth $l = t-1-(k_1-1)-k_2$. If we go from this vertex to p_s along the south side of the obstacle and then along the east side of the obstacle, then the depth of point p_s is at most $l + k_1 + (k_2 - 1) = t-1$. Thus, in this case there is also a canonical path from p to s which goes south whenever possible. □

Lemma 3 *Consider adjacent vertices v and w in the grid graph where v is north of w. In the northern region, without loss of generality, $d[v] = d[w]+1$.*

Proof: The proof follows immediately from Lemma 2. □

Lemma 4 *Consider adjacent vertices v and w in the grid graph where v is west of w. In the northern region, without loss of generality, $d[v] = d[w] \pm 1$.*

Proof: We prove the lemma by induction on the y-coordinate of the vertices in the northern region. If v and w have the same y-coordinate as s, then we know that $d[v] = d[w]+1$ if s is east of w and $d[v] = d[w]-1$ if s is west of v. Assume that the claim is true for vertices v and w with y-coordinate k. In the following we show that it is also true for vertices v and w with y-coordinate $k+1$. We distinguish the case that there is no obstacle directly south of v and w from the case that there is an obstacle directly south of v or w.

If there is no obstacle directly south of v and w the claim follows by Lemma 3 and the induction assumption.

Now we consider the case that there is an obstacle directly south of v or w. We assume without loss of generality that both v and w are on the boundary of the north side of the obstacle. (Note that v or w may, however, be at a corner of the obstacle.)

If our claim did not hold it would mean that $d[v] = d[w]$ for two adjacent vertices v and w (because, in any graph, the d values for adjacent vertices can differ by at most one). This would also mean that all shortest paths from v to s must go through vertex v_w at the north-west corner of the obstacle and all shortest paths from w to s must go through vertex v_e at the north-east corner of the obstacle. However, we next show that there is a grid point m on the boundary of the north side of the obstacle that has shortest paths through both v_e and v_w. The claim of Lemma 4 follows directly.

The distance x between m and v_w can be obtained by solving the following equation: $x + d[v_w] = (k-x) + d[v_e]$ where k is the length of the north side of the obstacle. The distance x is $(k + d[v_e] - d[v_w])/2$. This distance is integral and therefore, m exists because by inductive assumption the following holds: If k is even then $|d[v_e] - d[v_w]|$ is even, and if k is odd then $|d[v_e] - d[v_w]|$ is odd. □

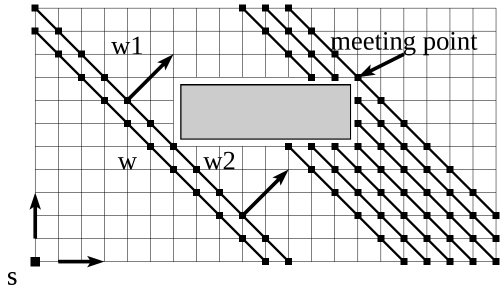

Figure 6: Splitting and merging of wavefronts along a corner of an obstacle. Illustration of meeting point and sibling wavefronts.

5 The wavefront algorithm

In this section we first develop some preliminary concepts and results based on an analysis of breadth-first search. We then present the wavefront algorithm, prove it to be correct, and show that it runs in linear time.

5.1 BFS and wavefronts

In city-block graphs 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 4 illustrates the wavefronts that can arise.

A *wavefront* w can then be defined as an ordered list of explored vertices $\langle v_1, v_2, \ldots, v_m \rangle$, $m \geq 1$, such that $d[v_i] = d[v_1]$ for all i, and such that $\delta(v_i, v_{i+1}) \leq 2$ for all i. (As we shall prove, the distance between adjacent points in a wavefront is always exactly equal to 2.) We call $d[w] = d[v_1]$ the distance of the wavefront.

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 are *sibling* wavefronts if they each have exactly one endpoint on the same obstacle and if the waves to which they belong merge on the far side of that obstacle. The point on an obstacle where the waves first meet is called the *meeting point* m of the obstacle. 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 Figures 6 and 7.

Lemma 5 *A wavefront can only consist of diagonal segments.*

Proof: By definition a wavefront is a sequence of vertices at the same distance to s for which the distance between adjacent vertices is at most 2. It follows from Lemma 3 and 4 that neighboring points in the grid cannot be in the same wavefront. Therefore, the distance between adjacent vertices is exactly 2. Thus, the wavefront can only consist of diagonal segments. $\square$

We call the points that connect diagonal segments (of different orientation) of a wavefront *peaks* or *valleys*. A peak is a vertex on the wavefront that has a larger y-coordinate than the y-coordinates of its adjacent vertices in the wavefront, and a valley is a vertex on the wavefront that has a smaller y-coordinate than the y-coordinates of its adjacent vertices as illustrated in Figure 7.

The initial wavefront is just a list containing the start point s. Until a successor of the initial wavefront hits an obstacle, the successor wavefronts consist of two diagonal segments connected by a peak. This peak is at the same x-coordinate for these successive wavefronts. Therefore, we say that the *shape* of the wavefronts does not change. In the northern region a wavefront can only have descendants that have a different shape if a descendant curls around the northern corners of an obstacle, or when it merges with another wavefront, or splits into other wavefronts. These descendants may have more complicated shapes.

A wavefront w splits whenever its hits an obstacle. That is, if a vertex v_i in the wavefront is on the boundary of an obstacle, w splits into wavefronts $w_1 = \langle v_1, v_2, \ldots, v_i \rangle$ and $w_2 = \langle v_i, v_{i+1}, \ldots, v_m \rangle$. Wavefront w_1 propagates around the obstacle in one direction, and wavefront w_2 propagates around in the other direction. Eventually, some descendant wavefront of w_1 and some descendant wavefront of w_2 will have a common point on the boundary of the obstacle - the meeting point. The position of the meeting point is determined by the shape of the wave approaching the obstacle. (In the proof of Lemma 4 vertex m is a meeting point and we showed how to calculate its postion once the length k of the north side of the obstacle and the shortest path distances of the vertices v_e and v_w at the north-east and north-west corners of the obstacle are known: The distance from v_w to the meeting point m is $(k + d[v_w] - d[v_e])/2$.)

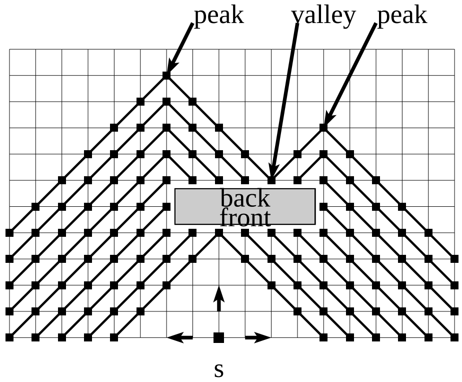

Figure 7: Shapes of wavefronts. Illustration of peaks and valleys, and front and back of an obstacle. The meeting point is the lowest point in the valley.

In the northern region, the *front* of an obstacle is its south side, the *back* of an obstacle is its north 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 (but not at a corner), this peak will not be part of the shape of the wave after it "passes" the obstacle. Instead, the merged wavefront may have one or two new peaks which have the same x-coordinates as the sides of the obstacle (see Figure 7). The merged wavefront has a valley at the meeting point on the boundary of the obstacle.

5.2 Description of 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 $w = \langle v_1, v_2, \ldots, v_l \rangle$ 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 $d[v] = d[w] + 1$. The learner starts with vertex v_1 and moves to all of its unexplored adjacent vertices. The learner then moves to the next vertex in the wavefront and explores its adjacent unexplored vertices. It proceeds this way down the vertices of the wavefront.

The following lemma shows that a wavefront of l ver-

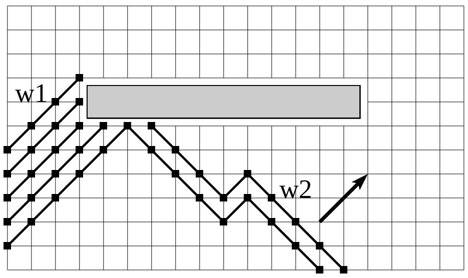

Figure 8: Blockage of w_1 by w_2. Wavefront w_1 has finished covering one side of the obstacle and the meeting point is not set yet.

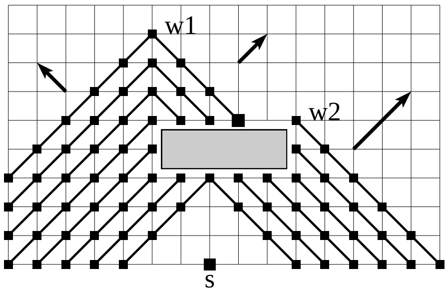

Figure 9: Blockage of w_1 by w_2. Wavefront w_1 has reached the meeting point on the obstacle, but the sibling wavefront w_2 has not.

tices can be expanded in time $O(l)$.

Lemma 6 *A learner can expand a wavefront* $w = \langle v_1, v_2, \ldots, v_l \rangle$ *by traversing at most* $2(l-1) + 2\lceil l/2 \rceil + 4$ *edges.*

Proof: To expand a wavefront $w = \langle v_1, v_2, \ldots, v_l \rangle$ the learner needs to move along each vertex in the wavefront and find all of its unexplored neighbors. This can be done efficiently by moving along pairs of unexplored edges between vertices in w. These unexplored edges connect l of the vertices in the successor wavefront. This results in at most $2(l-1)$ edge traversals, since neighboring vertices are at most 2 apart. The successor wavefront might have $l+2$ vertices, and thus at the beginning and the end of the expansion (i.e., at vertices v_1 and v_l), the learner may have to traverse an edge twice. In addition, at any vertex which is a peak, the learner may have to traverse an edge twice. Note that a wavefront has at most $\lceil l/2 \rceil$ peaks. Thus, the total number of edge traversals is at most $2(l-1) + 2\lceil l/2 \rceil + 4$. □

Since our algorithm computes exactly the same set of wavefronts as BFS, but persistently pushes one wave along, it is important to make sure the wavefronts are expanded correctly. 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 8), and (b) it cannot expand a wavefront that touches a meeting point until its sibling has arrived there as well (see Figure 9). 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*.

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. 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." We say that a wavefront expires if it consists of just one vertex with no unexplored neighbors.

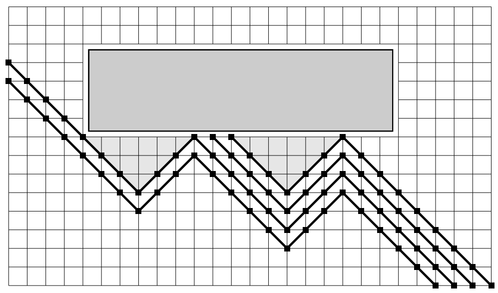

Figure 10: Triangular areas (shaded) delineated by two expiring wavefronts.

Procedure Wavefront-algorithm is an efficient optimally interruptible 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)
CREATE-MONOTONE-PATHS
for region = north, south, east, and west
    initialize current wavefront w ← ⟨s⟩
    EXPLORE-AREA(w, region)
    Take a shortest path to s
```

For each region we keep an ordered list L of all the wavefronts to be expanded. In the northern region, the wavefronts are ordered by the x-coordinate of their west-most point. *Neighboring* wavefronts are wavefronts that are adjacent in the ordered list L of wavefronts. Note that for each pair of neighboring wavefronts there is an obstacle on which both wavefronts have an endpoint.

Initially, we expand each wavefront in the northern region from its west-most endpoint to its east-most endpoint (i.e., we are expanding wavefronts in a "clockwise" manner). The direction of expansion changes for the first time in the northern region when a wavefront is blocked by a wavefront to its west (the direction of expansion then becomes "counter-clockwise"). In fact, the

```
Explore-area(w, region)
 1  initialize list of wavefronts L ← ⟨w⟩
 2  initialize direction dir ← clockwise
 3  repeat Expand current wavefront w to successor wavefront w_s
 4         Relocate (w_s, dir)
 5         current wavefront w := w_s
 6         if w is a single vertex with no unexplored neighboring vertices
 7           then remove w from ordered list L of wavefronts
 8                if L is not empty
 9                  then w := neighboring wavefront of w in direction dir
10                       Relocate (w, dir)
11           else replace w by w_s in ordered list L of wavefronts
12                if the second back corner of any obstacle(s) has just been explored
13                  then set meeting points for those obstacle(s)
14                if w can be merged with adjacent wavefront(s)
15                  then Merge (w, L, region, dir)
16                if w hits obstacle(s)
17                  then Split (w, L, region, dir)
18         if L not empty
19           then if w is blocked by neighboring wavefront w' in direction
                       D ∈ { clockwise, counter-clockwise }
20                  then dir := D
21                       while w is blocked by neighboring wavefront w'
22                          do w := w'
23                             Relocate (w, dir)
24   until L is empty
```

direction of expansion changes each time a wavefront is blocked by a wavefront that is in the direction opposite of expansion.

We treat the boundaries as large obstacles. The north region has been fully explored when the list L of wavefronts is 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 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 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. Note that more than one of these cases may apply.

Procedures Merge and Split (see the following page) 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 (that is, assignments to these variables within procedures Merge and Split affect their values in procedure Explore-area). Each time procedure Relocate(w, dir) is called, the learner moves from its current location to the appropriate end point of w: in the northern region, if the direction is "clockwise" the learner moves to the west-most vertex of w, and if the direction is "counter-clockwise," the learner moves to the east-most vertex of w.

Procedure Relocate(w, dir) can be implemented so that when it is called, the learner simply moves from its current location to the appropriate endpoint of w via a shortest path in the explored area of the graph. However, for analysis purposes, we assume that when Relocate(w, dir) is called the learner moves from its current location to the appropriate end point of w as follows.

- When procedure Relocate(w_s, dir) is called in line 4 of Explore-area, the learner traverses edges between the vertices in wavefront w_s to get back to the appropriate end point of the newly expanded wavefront.
- When procedure Relocate(w_s, dir) is called in line 10 of Explore-area, the learner traverses edges along the boundary of an obstacle.
- When procedure Relocate(w_s, dir) is called in line 8 of Merge, the learner traverses edges between vertices in wavefront w to get to the appropriate end point of the newly merged wavefront.
- When procedure Relocate(w_s, dir) is called in line 23 of Explore-area, the learner traverses edges as follows. Suppose the learner is in the northern region and at the west-most vertex of wavefront w_0, and assume that w is to the east of w_0. Note that both w_0 and w are in the current ordered list of wavefronts L. Thus there is a path between the learner's current location and wave-

front w which "follows the chain" of wavefronts between w_0 and w. That is, the learner moves from w_0 to w as follows. Let $w_1, w_2, \ldots, w_k$ be the wavefronts in the ordered list of wavefronts between w_0 and and w, and let $b_0, b_1, \ldots b_{k+1}$ be the obstacles separating wavefronts $w_0, w_1, \ldots, w_k, w$ (i.e., obstacle b_0 is between w_0 and w_1, obstacle b_1 is between w_1 and w_2, and so on). Then to relocate from w_0 to w, the learner traverses the edges between vertices of wavefront w_0 to get to the east-most vertex of w_0 which is on obstacle b_0. Then the learner traverses the edges of the obstacle b_0 to get to the west-point vertex of w_1, and then the learner traverses the edges between vertices in wavefront w_1 to get to the east-most vertex of w_1 which is on obstacle b_1. The learner continues traversing edges in this manner (alternating between traversing wavefronts and traversing obstacles) until it is at the appropriate end vertex of wavefront w.

```
MERGE(w, L, region, dir)
  remove w from list L of wavefronts
  while there is a neighboring wavefront w' with
            which w can merge
       do remove w' from list L of wavefronts
            merge w and w' into wavefront w''
            w ← w''
            put w in ordered list L of wavefronts
  if w is not blocked
     then RELOCATE (w, dir)
```

Wavefronts are merged when exploration continues around an obstacle. A wavefront can be merged with two wavefronts, one on each end.

```
SPLIT(w, L, region, dir)
  split w into appropriate wavefronts w_0, ..., w_n
     in standard order
  remove w from ordered list L of wavefronts
  for i = 0 to n
            put w_i on ordered list L of wavefronts
            if dir = clockwise
               then w ← w_0
               else  w ← w_n
```

When procedure SPLIT is called on wavefront w, we note that the wavefront is either the result of calling procedure EXPAND in line 4 of EXPLORE-AREA or the result of calling procedure MERGE in line 15 of EXPLORE-AREA. Once wavefront w is split into $w_0, \ldots, w_n$, we update the ordered list L of wavefronts, and update the current wavefront.

5.3 Correctness of the piecemeal search algorithm

The following theorems establish the correctness of our algorithm.

Theorem 3 *The algorithm* EXPLORE-AREA *expands wavefronts so as to maintain optimal interruptability.*

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

First we claim that at any time our algorithm knows the shortest path from s to any explored vertex in the north region. We show this by induction on the number of stages in the algorithm. Each stage of the algorithm is an expansion of a wavefront.

The shortest path property is trivially true when the number of stages $k = 1$. There is initially only one wavefront, the start point. Now we assume all wavefronts that exist just after the k-th stage satisfy the shortest path property, and we want to show that all wavefronts that exist just after the $k+1$-st stage also satisfy the shortest path property.

Consider a wavefront w in the k-th stage which the algorithm has expanded in the $k+1$-st stage to w_s. We claim that all vertices in w_s have shortest path length $d[w]+1$. Note that any vertex in w_s which is directly north of a vertex in w definitely has shortest path length $d[w]+1$. This is because there is a shortest path from any vertex v to s which goes south whenever possible, but if it is not possible to go south because of an obstacle, it goes east or west around the obstacle.

The only time any vertex v in w_s is not directly north of a vertex in w is when w is expanded around the back of an obstacle. This can only occur for a vertex that is either the west-most or east-most vertex of a wavefront in the north region. Without loss of generality we assume that v is the west-most point on w_s and v is on the boundary of some obstacle b. Let p be the path that leads northwards from the front east corner v_c of obstacle b to the meeting point of b. We know that there exists a shortest path from s to any vertex v_p on p that goes from s to v_c and from v_c to v_p along path p. (The shortest path does not go through the front west corner because v_p is east of the meeting point.) Because the algorithm only expands any wavefront until it reaches the meeting point of an obstacle, vertex v is not to the west of the meeting point. It has a shortest path from s that goes through v_c and along the obstacle to v. Thus, the wavefront that includes vertex v is expanded correctly so as to maintain shortest path information. □

Theorem 4 *There is always a wavefront that is not blocked.*

Proof: We consider exploration in the north region. The key observations are that (1) neighboring wavefronts cannot simultaneously block each other and (2) the east-most wavefront in the north 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 learner can always "follow a chain" of wavefronts to either its east or west to find an unblocked wavefront.

A neighboring wavefront is either a sibling wavefront or an expiring wavefront. An expiring wavefront can never block neighboring wavefronts. In order to show that neighboring wavefronts cannot simultaneously block each other, it thus suffices to show next that sibling wavefronts cannot block each other. We use this to show that we can always find a wavefront $\hat{w}$ which is not blocked. The unblocked wavefront $\hat{w}$ nearest in the ordered list of wavefronts L can be found by "following the chain" of blocked wavefronts from w to $\hat{w}$. By following the chain of wavefronts between w and $\hat{w}$ we mean that the learner must traverse the edges that connect the vertices in each wavefront between w and $\hat{w}$ in L and also the edges on the boundaries of the obstacles between these wavefronts. Note that neighboring wavefronts in list L each have at least one endpoint that lies on the boundary of the same obstacle.

Before we show that sibling wavefronts cannot block each other we need the following. The first time an obstacle is discovered by some wavefront, we call the point that the wavefront hits the obstacle the *discovery* point. (Note that there may be more than one such point. We arbitrarily choose one of these points.) In the north region, we split up the wavefronts adjacent to each obstacle into an *east wave* and a *west wave*. We call the set of all these wavefronts which are between the discovery point and the meeting point of the obstacle in a clockwise manner the west wave. We define the east wave of an obstacle in the same way.

The discovery point of an obstacle b is always at the front of b. The wavefront that hits at b is split into two wavefronts, one of which is in the east wave and one of which is in the west wave of the obstacle. We claim that a descendent wavefront w_1 in the west wave and a descendant wavefront w_2 in the east wave cannot simultaneously block each other. Assume that the algorithm is trying to expand w_1 but that wavefront w_2 blocks w_1. Wavefront w_2 can only block w_1 if one of the following two cases applies. In both cases, we show that w_1 cannot also block w_2.

In the first case, w_1 is about to expand to the back of obstacle b, but both of the back corners of obstacle b have not been explored, and thus the meeting point has not been determined. Wavefront w_2 can only be blocked by w_1 if w_2 is either already at the meeting point of the obstacle or about to expand to the back of the obstacle. Since none of the back corners of obstacle b have been explored, neither of these possibilities holds. Thus, wavefront w_1 does not block w_2.

In the second case, w_1 has reached the meeting point at the back of b. Therefore, both back corners of the obstacle have been explored and w_1 is not blocking w_2.

We have just shown that if w_2 blocks w_1 then w_1 cannot also block w_2. Thus, the algorithm tries to pick w_2 as the nearest unblocked wavefront to w_1. However, w_2 may be blocked by its sibling wavefront w_3 on a different obstacle b'. For this case, we have to show that this sibling wavefront w_3 is not blocked, or that its sibling wavefront w_4 on yet another obstacle b'' is not blocked and so forth. Without loss of generality, we assume that the wavefronts are blocked by wavefronts towards the east. Proceeding towards the east along the chain of wavefronts will eventually lead to a wavefront which is not blocked - the east-most wavefront in the northern region. The east-most wavefront is adjacent to the initial monotone east-north path. Therefore, it cannot be blocked by a wavefront towards the east. □

Theorem 5 *The wavefront algorithm is an optimally interruptible piecemeal search algorithm for city-block graphs.*

Proof: To show the correctness of a piecemeal algorithm that uses our wavefront algorithm for exploration with interruption, we show that the wavefront algorithm maintains the shortest path property and explores the entire environment.

Theorem 3 shows by induction on shortest path length that the wavefront algorithm mimics breadth-first search. Thus it is optimally interruptible.

Theorem 4 shows that the algorithm does not terminate until all vertices have been explored. Completeness follows. □

5.4 Efficiency of the wavefront algorithm

In this section we show the number of edges traversed by the piecemeal algorithm based on the wavefront algorithm is linear in the number of edges in the city-block graph.

We first analyze the number of edges traversed by the wavefront algorithm. Note that the learner traverses edges when procedures CREATE-MONOTONE-PATHS, EXPAND, and RELOCATE are called. In addition, it traverses edges to get back to s between calls to EXPLORE-AREA. These are the only times the learner traverses edges. Thus, we count the number of edges traversed for each of these cases. In Lemmas 7 to 10, we analyze the number of edges traversed by the learner due to calls of RELOCATE. Theorem 6 uses these lemmas and calculates the total number of edges traversed by the wavefront algorithm.

Lemma 7 *An edge is traversed at most once due to relocations after a wavefront has expired (line 14 of* EXPLORE-AREA*).*

Proof: Assume that the learner is in the northern region and expanding wavefronts in a clockwise direction. Suppose wavefront w has just expired onto obstacle b (i.e., it is a single vertex with all of its adjacent edges explored). The learner now must relocate along obstacle b to its neighboring wavefront w' to the east. Note hat w' is also adjacent to obstacle b, and therefore the learner is only traversing edges on the obstacle b.

Note that at this point of exploration, there is no wavefront west of w which will expire onto obstacle b. This is because expiring wavefronts are never blocked, and thus the direction of expansion cannot be changed due to an expiring wavefront. So, when a wavefront is split, the learner always chooses the west-most wavefront to expand first. Thus, the wavefronts which expire onto obstacle b are explored in a west to east manner. Thus relocations after wavefronts have expired on obstacle b continuously move east along the boundary of this obstacle. □

Lemma 8 *An edge is traversed at most once due to relocations after wavefronts have merged (line 10 of* MERGE*).*

Proof: Before a call to procedure MERGE, the learner is at the appropriate end vertex of wavefront w. Let's assume that the learner is in the northern region and expanding wavefronts in a clockwise direction. Thus the learner is at the west-most vertex of wavefront w. Note that wavefront w can be merged with at most two wavefronts, one at each end, but only merges with the wavefront to the west of w actually cause the learner to relocate. Suppose wavefront w is merged with wavefront w' to its west to form wavefront w''. Then, if the resulting wavefront w'' is unblocked, procedure RELOCATE is called and the learner must traverse w'' to its west-most vertex (i.e., also the west-most vertex of w'). However, since wavefront w'' is unblocked, w'' can immediately be expanded and is not traversed again. □

Lemma 9 *At most one wavefront from the east wave of an obstacle is blocked by one or more wavefronts in the west wave. At most one wavefront from the west wave is blocked by one or more wavefronts in the east wave.*

Proof: Consider the west wave of an obstacle. By the definition of blocking, there are only two possible wavefronts in the west wave that can be blocked. One wavefront is adjacent to the back corner of the obstacle. Call this wavefront w_1. The other wavefront is adjacent to the meeting point of the obstacle. Call this wavefront w_2.

We first show that if w_1 is blocked then w_2 will not be blocked also. Then we also know that if w_2 is blocked then w_1 must not have been blocked. Thus at most one wavefront in the west wave is blocked.

If w_1 is blocked by one or more wavefronts in the east wave then these wavefronts can be expanded to the meeting point of the obstacle without interference from w_1. That is, wavefront w_1 cannot block any wavefront in the east wave, and thus there will be no traversals around the boundary of the obstacle until the east wave has reached the meeting point. At this point, the west wave can be expanded to the meeting point without any wavefronts in the east wave blocking any wavefronts in the west wave.

Similarly, we know that at most one wavefront from the west wave is blocked by one or more wavefronts in the east wave. □

Lemma 10 *An edge is traversed at most three times due to relocation after blockage (line 33 of* EXPLORE-AREA*).*

Proof: Without loss of generality, we assume that the wavefronts are blocked by wavefronts towards the east. Proceeding towards the east along the chain of wavefronts will eventually lead to a wavefront which is not blocked, since the east-most wavefront is adjacent to the initial monotone east-north path.

First we show that any wavefront is traversed at most once due to blockage. Then we show that the boundary of any obstacle is traversed at most twice due to blockage. Note that pairs of edges connecting vertices in a wavefront may also be edges which are on the boundaries of obstacles. Thus any edge is traversed at most three times due to relocation after blockage.

We know from Theorem 4 that there is always a wavefront that is not blocked. Assume that the learner is at a wavefront w which is blocked by a wavefront to its east. Following the chain of wavefronts to the east leads to an unblocked wavefront w'. This results in one traversal of the wavefronts. Now this wavefront w' is expanded until it is blocked by some wavefront w''. Note that wavefront w'' cannot be to the west of w', since we know that the wavefront west of w' is blocked by w'. (We show in the proof of Theorem 4 that if w_1 blocks w_2 then w_2 does not block w_1.) The learner will not move to any wavefronts west of wavefront w' until a descendant of w' no longer blocks the wavefront immediately to its west. Once this is the case, then the west wavefront can immediately be expanded. Similarly, we go back through the chain of wavefronts, since - as the learner proceeds west - it expands each wavefront in the chain. Thus the learner never traverses any wavefront more than once due to blockage.

Now we consider the number of traversals, due to blockage, of edges on the boundary of obstacles. As wavefronts expand, their descendant wavefronts may still be adjacent to the same obstacles. Thus, we need to make sure that the edges on the boundaries of obstacles are not traversed too often due to relocation because of blockage. We show that any edge on the boundary of an obstacle is not traversed more than twice due to relocations because of blockage. That is, the learner does not move back and forth between wavefronts on different sides of an obstacle. Lemma 9 implies that each edge on the boundary of the obstacle is traversed at most twice due to blockage.

Thus, since the edges on the boundary of an obstacle may be part of the pairs of edges connecting vertices in a wavefront, the total number of times any edge can be traversed due to blockage is at most three. □

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 $14|E|$. Note that when the procedures CREATE-MONOTONE-PATHS, EXPAND, and RELOCATE are called, the learner traverses edges in the environment. In addition, the learner traverses edges in the environment to get back to s after exploration of each of the four regions. These are the only times the learner actually traverses edges in the environment. Thus, to calculate the total number of edge traversals, we count the edge traversals for each of these cases.

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 learner walks back to s four times after exploring each of the four regions. Thus the number of edges traversed here is at most $4|E|$. Lemma 6 implies that 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 calls to procedure RELOCATE.

Procedure RELOCATE is called four times within EXPLORE-AREA and MERGE. The four calls are due to expansion (line 6 of EXPLORE-AREA), expiring (line 14 of EXPLORE-AREA), merging (line 10 of MERGE) and blocking (line 33 of EXPLORE-AREA). Relocations after expanding a wavefront results in a total of $|E|$ edge traversals. Lemma 7 shows that edges are traversed at most twice due to expiring wavefronts. Lemma 8 shows that edges are traversed at most once due to relocations after merges. Finally, Lemma 10 shows that edges are traversed at most three times due to relocations after blockage. Thus the total number of edge traversals due to calls of procedure RELOCATE is at most $6|E|$.

Thus the total number edges traversed by the wavefront algorithm is at most $14|E|$. A more careful analysis of the wavefront algorithm can improve the constant factor. □

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 5 and Theorem 6. □

6 Ray algorithm

We now give another efficient optimally interruptible 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 may be less suitable for generalization, because it appears more specifically oriented towards city-block graphs.

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, with the following exceptions. Assume that it is operating in the northern region. The basic operation is to explore a northern-going "ray" as far as possible, and then to return to the start point of the ray. Along the way, side-excursions of one-step are made to ensure the traversal of east-west edges that touch the ray. Optimal interruptability will always be maintained: the ray algorithm will not traverse a ray until it knows a shortest path to s from the base of the ray (and thus a shortest path to s from any point on the ray, by Lemma 2).

The high-level operation of the ray algorithm is as follows. (See Figure 11.) From each point on the (horizontal segments of the) monotone paths bordering the northern region, a north-going ray is explored. On each such ray, exploration proceeds north until blocked by an obstacle or the boundary of the city-block graph. Then the learner backtracks to the beginning of the ray and starts exploring a neighboring ray. As described so far, each obstacle creates a "shadow region" of unexplored vertices to its north. These shadow regions are explored as follows. Once the two back corners of an obstacle are explored, the shortest paths to the vertices at the back border of an obstacle are then known; the "meeting point" is then determined. Once the meeting point for an obstacle is known, the shortest path from s to each vertex on the back border of the obstacle is known. The learner can then explore north-going rays starting at each vertex at the back border of the obstacle. There

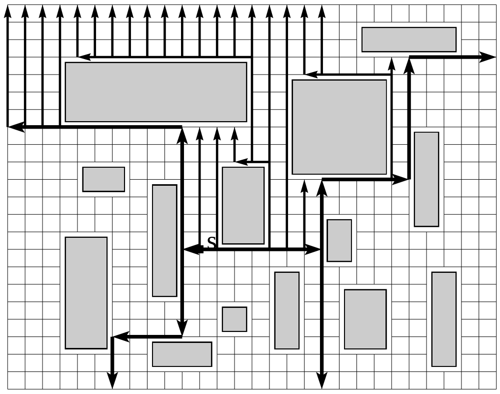

Figure 11: Operation of the ray algorithm.

may be further obstacles that were all or partially in the shadow regions; their shadow regions are handled in the same manner.

We 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. However, the ray algorithm is nonetheless an optimally interruptible search algorithm.

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

Proof: This follows from the properties of city-block graphs proved in Section 4, and the above discussion. In the ray algorithm each edge is traversed at most twice, with a careful attention to details. The linearity of the corresponding piecemeal search algorithm then follows from Theorem 2. □

7 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.

References

[1] Papadimitriou, Christos H. and M. Yanakakis. "Shortest paths without a map," Theoretical Computer Science, volume 84, 1991, pp. 127–150.

[2] Blum, Avrim, Prabhakar Raghavan and Baruch Schieber. "Navigating in Unfamiliar Geometric Terrain," Proceedings of Twenty-Third ACM Symposium on Theory of Computing, ACM, 1991, pp. 494–504.

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

[4] Rivest, Ronald L. and Robert E. Schapire. "Inference of Finite Automata using Homing Sequences," Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing, ACM, Seattle, Washington, May 1989, pp. 411–420.

[5] Deng, Xiaotie and Christos H. Papadimitriou. "Exploring an Unknown Graph,"Proceedings of the 31st Symposium on Foundations of Computer Science, 1990, pp. 355–361.

[6] Betke, Margrit. "Algorithms for Exploring an Unknown Graph". MIT Laboratory for Computer Science, Technical Report MIT/LCS/TR-536. March, 1992.

[7] Deng, Xiaotie, Tiko Kameda and Christos H. Papadimitriou. "How to learn an unknown environment," Proceedings of the 32nd Symposium on Foundations of Computer Science, 1991, IEEE, pp. 298–303.

[8] Bar-Eli, E., P. Berman, A. Fiat and P. Yan. "On-line Navigation in a Room,"Symposium on Discrete Algorithms, 1992, pp. 237–249.

[9] Edmonds, Jack and Ellis L. Johnson. "Matching, Euler Tours and the Chinese Postman", Mathematical Programming, 1973, volume 5, pp. 88–124.

[10] Papadimitriou, Christos H. "On the complexity of edge traversing" J. Assoc. Comp. Mach., 1976, volume 23, pp. 544–554.

[11] Rao, Nagewara S. V., Srikumar Kareti, Weimin Shi and S. Sitharama Iyengar. "Robot Navigation in Unknown Terrains: Introductory Survey of Non-Heuristic Algorithms"Oak Ridge National Laboratory, July 1993, ORNL/TM-12410.