An Efficient Algorithm for Shortest Paths in Vertical and Horizontal Segments

David Eppstein *   David W. Hart *

Abstract. Suppose one has a line segment arrangement consisting entirely of vertical and horizontal segments, and one wants to find the shortest path from one point to another along these segments. Using known algorithms one can solve this in $O(n^2)$ time and in $O(n^2)$ space. We show that it is possible to find a shortest path in time $O(n^{1.5} \log n)$ and space $O(n^{1.5})$. Furthermore, if only one path endpoint is known in advance, it is possible to preprocess the arrangement in the same time and space and then find shortest paths for query points in time $O(\log n)$.

1 Introduction

Consider the problem of finding the shortest path from one location to another in a large city. Many cities have two sets of parallel streets, where the streets of one set are perpendicular to the streets of the other. These streets may have sections closed or there may be points where the streets do not go through.

Geometrically, we can represent the city streets as line segments, with endpoints where the streets do not go through. What we have is an arrangement of horizontal and vertical line segments. We would like to find a shortest path from one point on some segment to another point on another segment.

There are many other routing problems that fit this model, such as routing pipes and wiring in a building, routing utility lines in a city, and routing connections on a circuit board.

One field in particular where this problem has application is VLSI design. In VLSI design, one has a set of cells, which represent logic elements in the circuit and which have fixed locations. In between cells, one has vertical and horizontal channels where one can route connections. The cost of the connection is the path length, so one wants to minimize path lengths in this arrangement of vertical and horizontal segments.

De Rezende et al. [8] study a related problem of finding a rectilinear shortest path from one point to another in a region where there are obstacles consisting of disjoint, axis parallel rectangles. The constraint is that the path must not pass through the rectangles. For this problem, the authors give a solution that takes $O(n \log n)$ preprocessing time, given the path starting point, and $O(\log n)$ time per query for each path ending point.

Their problem is less general, requiring that the obstacles be disjoint rectangles. For the problem presented here of an arrangement of segments, we make

* Department of Information and Computer Science, University of California, Irvine, CA 92697-3425. Email to eppstein@ics.uci.edu and dhart@ics.uci.edu.

no assumptions about what obstacles or other constraints end each segment; we allow any set of vertical and horizontal segments.

The problem discussed here is also related to other problems in computational geometry. The first is, given an arrangement consisting of line segments with O(1) different orientations, find a shortest path between two points. We have not found an easy way to extend the methods of this paper to solve this problem. The second, more general problem is, given any arrangement of lines, find a shortest path between two points in less than $O(n^2)$ time and space. Both of these problems remain open. Bose et al. [2] describe a method for approximating the shortest path in a line arrangement.

Returning to the problem discussed in this paper: one can solve this using known algorithms as follows. Given n line segments, we can compute the arrangement for these segments in time $O(n^2)$. The arrangement has $O(n^2)$ vertices. An algorithm by Klein, et al. [5] computes shortest paths in a planar graph in linear time. Using this gives a time of $O(n^2)$ to find a shortest path.

We would like to improve this time, but of perhaps greater concern is that this approach also takes $O(n^2)$ space. We show here that one can compute shortest paths faster and in less space. We describe an algorithm that computes a path in time $O(n^{1.5} \log n)$ and space $O(n^{1.5})$.

We also look at the case where only one path endpoint is known in advance. We show that one can preprocess the segments in the same time and space bounds, and one can then find a shortest path for query points in time $O(\log n)$.

2 The Basic Concept

The intuition for what our algorithm does is easy to describe. Consider first where the undesirable $O(n^2)$ bound arises. We have only n lines, but these produce as many as $n^2/4$ vertices at intersection points. For the number of intersection points to be large, each of the n lines must intersect most of the other n lines. If this were not the case, then there would be few intersection points, and we could solve the problem quickly.

In an arrangement of vertical and horizontal segments, if one has most vertical lines intersecting most horizontal lines, the arrangement contains many regions like the one shown in Figure 1, where all vertical lines intersect all horizontal lines. Suppose there is a path starting point at one side of this region and a path destination point on another side. The algorithms described above look at paths going through all n^2 vertices and edges. But clearly there are very many different paths in this grid that are all equally good. An algorithm that checks all these paths is doing a tremendous amount of unnecessary work.

2.1 Algorithm Overview

It is possible to avoid looking at many of the vertices and edges in the arrangement. Consider a grid of lines and the rectangular shape consisting of the four outermost segments. If the path starts at one side of this rectangle and leaves

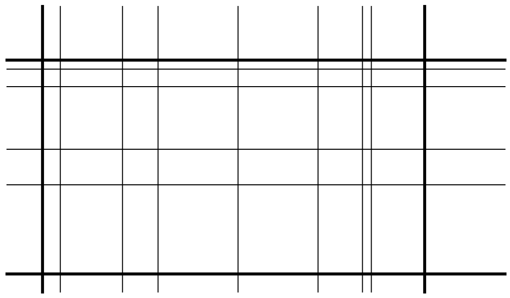

Fig. 1. A region where all vertical segments intersect all horizontal segments, defined here to be a *complete grid*. The bold lines are *boundary segments* and all other lines are *interior segments*.

from a side with different orientation, then a path through the interior of the rectangle has corresponding to it an equally short path traveling only on the outermost lines. If a path starts at one side of this rectangle and leaves from the opposite side, it may be necessary to travel through the interior of the rectangle, but all of this travel can be on a single segment.

Thus, the algorithm described here constructs a graph that eliminates most of the intersection points and the corresponding, numerous edges, but the graph produced has a shortest path equal to the shortest path in the original arrangement.

The algorithm finds regions where all vertical lines intersect all horizontal lines. It adds the outermost segments—the boundary of a rectangle—to the graph, including vertices at all intersections of these lines with other lines. It adds lines crossing the rectangle but omits vertices at the intersections of these lines with each other; this eliminates many of the intersection points found in the arrangement.

2.2 Definitions

Suppose one can identify a rectangular region where every vertical line intersects every horizontal line. This region is what we term a *complete grid* (see Figure 1). A paper by Chan and Chin [3] uses a similar concept of complete grids. When our algorithm finds a complete grid, it adds all the line segments to the graph but leaves out every intersection point in the interior of the rectangle.

More precisely, the algorithm determines which four line segments are outermost. These are what we term *boundary lines*. Every intersection of a line with a boundary line is included as a vertex of the graph. Every intersection point cuts line segments into pieces; these are edges of the graph.

Lines in the complete grid other than boundary lines are what we term *interior lines*. The algorithm does not include, as vertices of the graph, the intersection points of interior lines with other interior lines. Each interior line becomes three edges: the two fragments outside the rectangle and the one subsegment that crosses the interior of the rectangle (see Figure 1).

A special case occurs if a path endpoint is located on a line segment of the complete grid. In this case, the algorithm treats this segment just like a boundary segment, including as vertices in the graph all intersection points of this segment with any other segments.

One of the difficulties making this approach work is finding, in a complex arrangement of segments, regions that fit the description of a complete grid. The solution we describe uses a technique similar to that used in a paper by Bern, Dobkin, and Eppstein [1]. The algorithm finds and enlarges complete grids by looking in vertical strips of the arrangement that are $\sqrt{n}$ vertical lines wide.

It makes use of the fact that there are few ($2n$) segment endpoints in the arrangement. The only interruption of a complete grid that can occur is the start or end of a line segment. The algorithm enlarges grids until an endpoint interrupts its progress. The algorithm then adds the complete grid to the graph, and continues by finding the next complete grid.

This produces a graph representing the arrangement, but unlike the naively constructed graph, this is not a planar graph. The previously mentioned algorithm by Klein et al. [5] works only for planar graphs. Thus, we apply Dijkstra's shortest path algorithm to our graph.

2.3 Partitioning the Rectangle

Before discussing the algorithm in detail, we discuss an abstraction that will aid precise mathematical analysis. Consider a rectangular region R of the plane defined by the extreme x and y values of all segment endpoints (and thus enclosing all segments). Imagine that all vertical line segments extend the full length of this rectangle. Choose vertical lines at intervals of $\sqrt{n}$, including the left and right boundaries of the rectangle. These lines partition the rectangle into vertical strips. The vertical line segments on the boundary of each strip could be considered to belong to the strips on either side; we assign them to the strip on their left.

Now imagine that, for each vertical strip, we sweep a horizontal line down the strip. Every time the sweep line intersects the endpoint of a vertical line, we add to the partition of R a horizontal segment across the strip. If the sweep line crosses a horizontal line extending all the way across the strip, we do nothing. If the line crosses a horizontal line segment with an endpoint in the strip, we add another partitioning horizontal segment.

Doing this for each strip of the rectangle gives a partition of R into subrectangles, where the interiors of the subrectangles are disjoint. The subrectangles will correspond to complete grids. The rectangles include and intersect at their boundary lines. Taking the intersection of each subrectangle with the set of line segments gives a set of subsegments. Since the union of the subrectangles equals rectangle R, the union of all subsegments in the subrectangles equals the set of line segments.

3 Algorithm

The first step in the algorithm is to create an events list. We combine the lists of horizontal and vertical lines and sort them by y coordinate, where both endpoints of the vertical line segments count as events.

We choose vertical line segments that are separated by $\sqrt{n}$ vertical line segments. The vertical lines corresponding to these segments partition the rectangle into vertical strips. We will start with the leftmost vertical strip. We include any vertical line segment that is on a strip boundary as part of the vertical strip for which it is the right hand boundary. The algorithm processes all vertical strips going from left to right.

The computations we do for a vertical strip are as follows. We have a horizontal sweep line travel down the vertical strip, stopping for each event on the event list. We will build up a set of line segments corresponding to a subrectangle in our previously described partition of rectangle R, and the line segments will satisfy the definition given earlier of a complete grid. The set of segments is initially empty at the top of each vertical strip.

One event type is a horizontal line. If the horizontal line does not intersect the vertical strip, the algorithm does nothing. If the horizontal line segment extends the full width of the vertical strip, we add the line segment to a list of horizontal line segments in the current complete grid. If the horizontal line segment has one or both endpoints in the interior of the vertical strip, we define this segment to be a *free segment*, which is not part of the complete grid. This ends the complete grid, and we call the graph construction routine, providing the set of lines in the complete grid and the free segment as the parameters.

The event encountered by the sweep line may be an endpoint of a vertical line segment. If so, we call the graph construction routine with the set of lines in the complete grid as the parameter. In this case, there is no free segment.

After adding a complete grid, we reset the current list of lines. Doing this means setting the list of horizontal lines to null, and for the vertical lines, finding all line segments in the vertical strip that both intersect the horizontal sweep line and extend below it. This becomes the current set of lines for the next complete grid.

We continue moving the sweep line until the next event is reached or we reach the end of the vertical strip. At the end of the strip, we call our routine for adding the current complete grid to the graph. We then start the sweep down the next vertical strip.

3.1 Adding a Complete Grid to the Graph

The part of the algorithm described above finds sets of lines that fit our definition of a complete grid. In this section we describe how the algorithm constructs a part of the graph containing all line segments in the complete grid.

We use an adjacency list representation for the graph, and for each vertex we record its Cartesian coordinates. To add a complete grid, the algorithm constructs the graph by adding subsegments from the grid going from left to right and top to bottom.

At any time in the construction of the graph, line segments may have subsegments that are represented in the graph and other subsegments that are to be added to the graph. Adding a complete grid to the graph may add subsegments from a segment already represented in the graph, or it may add entirely new subsegments. Here we describe low level operations that are used in constructing the graph.

The low level operations depend on what is already in the graph. Vertices in the graph may be one of two types. The first type is a vertex at the intersection of segments. These vertices, once placed in the graph, are never changed except to add edges. The second type of vertex is a subsegment endpoint; it is not at an intersection. If a colinear subsegment is to be added to the graph (that is, both subsegments are part of the same segment), the algorithm always moves the existing endpoint by changing the coordinates associated with it. The vertex becomes the endpoint of the new, longer subsegment.

It may happen that a new vertex is at an intersection of two segments, and both segments have subsegments already in the graph. If both subsegments in the graph end in vertices that are not at intersections, one has a choice of which existing vertex to move. The algorithm picks one vertex. Since the other subsegment should be lengthened to end at this intersection, the vertex that is its endpoint is deleted and replaced by the newly moved vertex.

Still considering a new vertex at an intersection, if exactly one of the subsegments ends at an intersection point, the other vertex is moved. If both subsegments end at intersection points, then a new vertex is added at the new intersection point.

In summary, the low level operations for placing a vertex in the graph are to add, move, or move and delete vertices.

4 Algorithm Running Time

Here we analyze the running time of the algorithm. This has three components: the time for finding complete grids, the time for constructing a graph from the complete grids, and the time for applying Dijkstra's shortest paths algorithm to the graph. To determine how long Dijkstra's algorithm takes, we must bound the number of vertices and edges in the constructed graph.

4.1 Time to Find Complete Grids

The time for finding complete grids includes the time for sorting the events list and the time for passing a sweep line down $\sqrt{n}$ vertical strips processing events. Each event is simply to check whether endpoints are in the vertical strip, so each takes $O(1)$ time. There are $O(n)$ events processed in each vertical strip, giving $O(n^{1.5})$ time spent looking at events.

For certain events, the algorithm adds or removes lines from the two linked lists of lines representing the complete grid. Horizontal lines are encountered in sorted order, so adding them to a sorted linked list takes constant time. This happens at most n times in a vertical strip, giving a bound of $O(n^{1.5})$ total time for these operations.

Encountering vertical segment endpoints requires adding or removing vertical lines from the sorted linked list of vertical segments. The list may be as long as $\sqrt{n}$, and these events happen at most $2n$ times total in the algorithm. The total time for these operations is thus $O(n^{1.5})$.

We conclude that the time for finding complete grids is $O(n^{1.5})$.

4.2 Graph Construction Time

Here we look at the time it takes to construct the graph. We have the following lemma.

Lemma 1. *The time for constructing the graph is $O(n^{1.5} + |V|)$.*

Proof sketch. In every step the algorithm adds subsegments to the graph, and to do this it adds new vertices, moves existing vertices, and deletes existing vertices. The addition, deletion, or move each take constant time once one has accessed the nearest vertices. The algorithm accesses vertices in the graph in order from left to right and top to bottom, so the algorithm takes constant time to access vertices.

We now bound the total time of constructing the graph by charging the time of the low level operations (except moves) to vertices of the graph.

If we add a vertex, we charge this constant time to the vertex. For deletions: any time we delete a vertex, it is because we have placed a vertex at the intersection of two subsegments. This vertex is never itself deleted or moved. We thus charge the cost of deleting a vertex to the vertex that stays in the graph, and we also transfer the cost accrued to the deleted vertex (accrued when it was added to the graph) to the vertex that stays in the graph.

We next look at moving a vertex. Instead of charging this cost to vertices, we count the total number of moves possible. One can show that the total number of move operations on all vertices is $O(n^{1.5})$.

We have shown that the time for finding the complete grids is $O(n^{1.5})$, the time for moving vertices as we construct the graph is $O(n^{1.5})$, and the time for adding and deleting vertices is $O(|V|)$, proving the lemma. □

4.3 A Bound on the Number of Vertices and Edges

We bound the total number of vertices and edges in the graph by determining how many intersection points the algorithm adds for all complete grids and free lines. The only other vertices in the graph are $O(n)$ segment endpoints.

In a vertical strip, a complete grid (and possibly a free line) is added to the graph whenever the algorithm encounters an endpoint of a line. Complete grids are also added when the algorithm reaches the bottom of a vertical strip. There are only $2n$ line endpoints and $\sqrt{n}$ vertical strips, so the algorithm adds complete grids at most $2n + \sqrt{n}$ times.

A complete grid has (at most) 4 boundary lines. The horizontal boundary lines intersect no more than $\sqrt{n}$ vertical lines, so the complete grid horizontal boundary lines add at most this many intersection points. The vertical boundary lines can intersect up to n horizontal lines, but in fact the sum of intersections for all vertical boundary lines for all complete grids in the vertical strip is at most $2n$. This is because at any point in the strip there are only two vertical boundary lines; except for the left/right pair, vertical boundary lines in the same strip do not overlap.

Since there are 2 path endpoints, they add at most 2 segments for which all intersection points are added to the graph. Free lines (always horizontal) can intersect up to $\sqrt{n}$ vertical lines each.

All intersection points of segments with other segments are accounted for in these computations since there are no interior to interior line intersection points.

We add at most one free line each time we add a complete grid. The number of vertices added for horizontal segments for each complete grid is at most $3\sqrt{n}$.

Let n_i be the number of complete grids in vertical strip i. Then the total number of intersection points in strip i is bounded by $3\sqrt{n}\ n_i + 2n$. The total number of intersection points in all $\sqrt{n}$ strips (ignoring those added by segments containing path endpoints) is bounded by

$$\sum_{1 \leq i \leq \sqrt{n}} (3\sqrt{n}\, n_i + 2n)$$

We stated above that $\sum_i n_i \leq 2n + \sqrt{n}$. This gives us a bound on intersection points of

$$3\sqrt{n}(2n + \sqrt{n}) + 2n\sqrt{n}$$

Adding the intersections produced by boundary lines containing the path endpoints, we get

$$\begin{aligned} |V| &\leq 3\sqrt{n}(2n + \sqrt{n}) + 2n\sqrt{n} + 2n \\ &= O(n^{1.5}) \end{aligned}$$

This lets us finish the work of bounding the time for constructing the graph with the following theorem.

Theorem 2. *This algorithm takes $O(n^{1.5})$ time to construct a graph.*

The number of edges in the constructed graph is bounded by 4 times the number of intersection points plus (in case there are no intersection points) the number of line segments; this gives $O(n^{1.5})$ edges. Applying Dijkstra's single source shortest paths algorithm to this graph, we compute a shortest path in time

$$\begin{aligned} & O(|V|\log|V| + |E|) \\ &= O(n^{1.5}\log(n^{1.5}) + n^{1.5}) \\ &= O(n^{1.5}\log n) \end{aligned}$$

We state this as a theorem.

Theorem 3. *The constructed graph requires $O(n^{1.5})$ space, and a shortest path in it can be computed in time $O(n^{1.5}\log n)$.*

5 Paths in the Constructed Graph

Here we prove that the shortest path in the original arrangement G has an equally short path in the constructed graph G'.

Lemma 4. *The constructed graph G' contains edges representing all line segments in the the arrangement G and vertices for all intersection points except interior to interior line intersections.*

This follows from the manner of constructing G', and the proof is omitted.

Theorem 5. *Suppose one is given an arrangement of vertical and horizontal segments G and points s and t in it. If G' is the graph constructed by the algorithm described above, then G' contains a path P' between s and t with weight equal to the shortest path P in G.*

Proof. We first need to explain some terminology. We say that a path *crosses through* a vertex v if it goes from one line segment to a different line segment through intersection point v. If the path goes through a vertex but continues on what, in the arrangement, is the same line segment, then we say that the path *does not cross through* the vertex.

Let P be a shortest path in the arrangement G. By our lemma, G' contains all segments found in G. If P crosses through only vertices found in G', then the same path exists in G', so we have nothing to prove.

Suppose instead that P crosses through some vertex (or vertices) not found in G'. Then, since G' does not contain a vertex at this intersection point, a path cannot go from one line segment to another, so the same path cannot exist in G'. Let S be the set of vertices that P crosses through that are not in G'. Choose one vertex v from S.

By the lemma above, G' contains every vertex contained in G except vertices at the intersection of interior lines with interior lines. Since the vertex v is not in G', it is at an intersection of interior lines. Note also that the omitted vertices are always at intersections of interior line segments from the same complete grid. Thus we can say that v belongs to a complete grid.

In the complete grid, the outermost line segments are designated as boundary segments. The four (at most) boundary segments define a rectangle; we denote this rectangle, including its interior, by r. (If there are fewer than four boundary segments, there is no enclosed region and so no interior.)

In the remainder of the proof we consider three cases, depending upon whether the path endpoints s and t are inside the rectangle r bounded by boundary segments. These are (1) both path endpoints s and t are exterior to or on the boundary of r, (2) one of s or t is in r, and (3) both path endpoints are in r.

Case 1 Suppose that both path endpoints s and t are exterior to or on the boundary of rectangle r. Consider vertex v from set S: by the definitions of boundary lines and interior lines, v is inside r.

Since neither s nor t is inside rectangle r, there must be some first point p_1 (starting from endpoint s) where P intersects the boundary of r and some last point p_2 where P ends travel on the boundary of r, never again intersecting r. (These points may be s or t.)

The first possibility is that one of these crossing points (say WLOG p_1) is on a horizontal boundary line h_1 and the other point p_2 is on a vertical boundary line v_2. Then there is no shorter path from p_1 to p_2 than $\overline{p_1 h_1 v_1 p_2}$. (We describe paths by listing both points and line segments on the path.) Thus we can replace the part of P from p_1 to p_2 with this subpath and we have at least as good a path as P.

This new path does not travel through the interior of r, and in particular it does not cross through vertex v. The number of interior to interior intersection points crossed through (the vertices of S) has decreased by at least one.

The second possibility is that both of the intersection points are on vertical boundary lines v_1 and v_2. The intersection at vertex v involves some horizontal line h_1. No path from p_1 to p_2 that travels through any point of h_1 is shorter than $\overline{p_1 v_1 h_1 v_2 p_2}$.

Thus we can replace the subpath of P from p_1 to p_2 with this path and have a new path that is at least as good. The new path reduces the number of cross-through vertices by at least one.

Case 2 Suppose that exactly one path endpoint, say s, is in the interior of rectangle r. Since t is exterior to r, starting at s the path must cross the rectangle boundary at least once. Let p_1 be the last point (this may be t) where P intersects the boundary of rectangle r before reaching t.

Suppose WLOG that p_1 is on a vertical boundary line v_1. If path endpoint s is on a horizontal segment h_1, then no path from p_1 to s is shorter than $\overline{p_1 v_1 h_1 s}$. Replacing the subpath of P going from p_1 to s with this subpath gives a new path that is at least as good as P. This new path reduces the size of the set of cross-through vertices S by at least one.

Still supposing that the boundary line containing p_1 is vertical line v_1, now suppose that path endpoint s is on a vertical segment v_2. Since the path endpoint is on this segment, the algorithm we described treats this segment as if it were a boundary segment; that is, all intersection vertices of segment v_2 with other lines are contained in the constructed graph G'.

The intersection point v is at an intersection involving some horizontal line h_1. There is no shorter path from p_1 to v than $\overline{p_1 v_1 h_1 v}$ and no shorter path from v to s than $\overline{v h_1 v_2 s}$. Replacing the subpaths $\overline{p_1 v}$ and $\overline{vs}$ in P with these new subpaths gives a path that is at least as good as P and does not cross through vertex v. Thus, the number of vertices in the set S of cross through vertices is decreased by at least one.

Case 3 Both path endpoints are in rectangle r. Remembering that any segment holding an endpoint has all intersection points with other segments included in G', this case is easy and is omitted.

All cases For any path in G, we use the above arguments to find, for any path that crosses through an interior to interior intersection vertex v, an equally good path that eliminates this cross-through vertex using a subpath contained entirely in G'. We can do this iteratively on P, removing cross through vertices, until we have a path that is at least as good and does not cross through interior to interior vertices. This path can be represented in G', completing the proof. □

This gives us our main result:

Theorem 6. *The algorithm described here computes a shortest path in an arrangement of horizontal and vertical segments in time $O(n^{1.5} \log n)$ and space $O(n^{1.5})$.*

6 Query points

Suppose one has a shortest path problem as described above, but one only knows the location of one path endpoint in advance. We show here that it is possible to preprocess the arrangement in time $O(n^{1.5} \log n)$ and space $O(n^{1.5})$ and then compute shortest paths for query points in time $O(\log n)$.

The algorithm begins by constructing a graph as described above, where we have only one known path endpoint. It then computes the shortest paths from the known endpoint to all vertices in the graph using Dijkstra's single source shortest paths algorithm.

To find the shortest path for a query point t, the algorithm must find vertices in the constructed graph that are close to the query point. In the worst case, the algorithm must check paths from 6 selected vertices in the graph.

Suppose the query point is at the intersection of two segments. If the intersection point is not the intersection of two interior segments, then the intersection point is a vertex of the constructed graph, and the already computed shortest path to the vertex is correct (as discussed below). If the intersection point is the intersection of two interior lines, then the algorithm finds the four vertices where these two interior lines intersect the boundary lines. It computes the path length to t from each of these four points and chooses the best of these. This is a shortest path in the arrangement.

Suppose the query point is on a single segment and so is not at an intersection. If this point is not in the interior of the grid boundary rectangle, one finds the two vertices that are nearest to the point on the segment and chooses the best path from these vertices.

The most complex case occurs when the point is interior to a boundary rectangle and not at an intersection. First one needs to look at the two vertices where the segment containing t intersects the boundary. Next, one needs to find two segments perpendicular to this segment that are closest to t on each side of t. These two segments intersect the boundary at four points. This gives six vertices total. One finds the best paths from these six vertices to t (in $O(1)$ time each) and chooses the best of these. This gives a shortest path in the arrangement.

If t is located in the interior of the boundaries of a complete grid and s is also interior to this rectangle, then one treats it as a special case; one can compute the shortest path in $O(1)$ time once one has found "close" segments.

We must describe, then, how one quickly finds the "close" vertices described above. The algorithm builds a lookup table for vertical lines and lookup tables for each complete grid.

For the set of vertical lines we have the following. We have an array of vertical lines sorted by x coordinate. For each vertical line, we store a (variable sized) array containing a list, sorted by y coordinate, of all complete grids that intersect that line (even though the segment, as opposed to the line, may not pass through a complete grid). Then, given the coordinates of the query point, we can find the vertical line (or nearest vertical line) in time $O(\log n)$ and find the complete grid on that line in time $O(\log n)$.

In a vertical strip there are $\sqrt{n}$ vertical lines, and we define n_i to be the number of complete grids in the strip. As stated previously, $\sum_i n_i \leq 2n + \sqrt{n}$. Then the space for this data structure is

$$\begin{aligned} & \textstyle\sum_i \sqrt{n}\ n_i \\ = & \ O(n^{1.5}) \end{aligned}$$

For each complete grid, we maintain two sorted arrays: one for horizontal boundaries and one for vertical boundaries. These each contain coordinates paired with pointers to vertices in the graph that are located at that x or y coordinate. Thus, once one finds the complete grid containing the query point, one can find the "closest" vertices—the ones needed by the computations described above—in $O(\log n)$ time. The number of pointers maintained is less than the number of vertices in the graph, so the space is $O(n^{1.5})$.

To see that the computations described here actually find a shortest path, we need to prove the following.

Lemma 7. *The shortest paths computed by Dijkstra's algorithm for all vertices in the constructed graph are shortest paths to these vertices in the arrangement.*

This does require proof, since the previous proofs all assume that t is known in advance, and the segments containing t are treated specially. The proof is easy and is omitted.

Theorem 8. *There is a shortest path from s to t that passes through one of the (at most) 6 vertices described above.*

Proof sketch. If t is not in the interior of the boundaries of a complete grid, then it is either at an intersection point, for which the shortest path has already been computed, or it is between two vertices, and the shortest path passes through one of these vertices.

Suppose that t is in the interior of the region bounded by grid boundary segments. Assume also that s is not in the interior. Then a shortest path from s to t must pass through the boundary. Let p_1 be the first point where the path from s intersects the boundary. Then traveling from p_1 to one of (whichever gives the best path) the six vertices described above (all on the boundary) and taking the obvious shortest path from that vertex to t gives a path that is as good as any other path traveling through p_1 to t.

If s is in the interior of the rectangle, one treats this as a special case and computes the shortest path in $O(1)$ time once t has been located. □

7 Concluding remarks

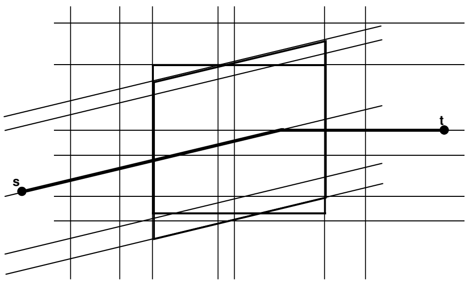

Fig. 2. An example with 3 line orientations where a shortest path (bold path in figure) does not have an equally short path in the constructed graph of overlapping, different-orientation grids.

It would be nice if the algorithm described here could be generalized to find shortest paths in an arrangement containing segments with $O(1)$ different orientations. However, a direct extension of the methods here to more line orientations does not appear to work.

The most direct way to extend the ideas of this paper (to 3 orientations for example) is as follows. Choose one line orientation and choose partitioning lines at $\sqrt{n}$ intervals. Find complete grids with respect to lines at a second orientation and build a graph from these. Then find complete grids with line segments of the third orientation and add these to the graph.

One must decide for this approach whether interior segments of grids from one orientation should have vertices at intersections with interior segments from different orientation, overlapping grids. The answer has to be that these intersections are not added as vertices; otherwise, the graph will have too many vertices. But as shown in Figure 2, there exist shortest paths in the arrangement for which no equally short path can be found in a graph constructed this way.

The factor $O(n^{1.5})$ in our results may be induced by the solution method rather than the problem. It is an open question whether one can find shortest paths in vertical and horizontal segments more quickly and in less space.

References

1. M. Bern, D. Dobkin, and D. Eppstein. Triangulating polygons without large angles. *International Journal of Computational Geometry & Applications* 5 (1995) 171-192.
2. P. Bose, W. Evans, D. Kirpatrick, M. McAllister, and J. Snoeyink. Approximating shortest paths in arrangements of lines. *Proceedings of the 8th Canadian Conference on Computational Geometry* (1996) 143-148.
3. W.-T. Chan and F. Y. L. Chin. Efficient algorithms for finding disjoint paths in grids. *Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms* (1996) 454-463.
4. J. Hershberger and S. Suri. Efficient computation of Euclidian shortest paths in the plane. *Proceedings of the 24th Annual Symposium on Foundations of Computer Science* (1993) 508-517.
5. P. Klein, S. Rao, M. Rauch, and S. Subramanian. Faster shortest-path algorithms for planar graphs. *Proceedings of the 26th Annual ACM Symposium on the Theory of Computing* (1994) 27-37.
6. D.T. Lee, C.D. Yang, and C.K. Wong. Rectilinear paths among rectilinear obstacles. *Discrete Applied Mathematics* 70 (1996) 185-215.
7. F. Preparata and M. Shamos. *Computational Geometry.* Springer-Verlag, 1985.
8. P.J. de Rezende, D.T. Lee, and Y.F. Wu. Rectilinear shortest paths in the presence of rectangular barriers. *Discrete & Computational Geometry* 4 (1989) 41-53.

This article was processed using the LaTeX macro package with LLNCS style