An $O(n^{5/2} \log n)$ Algorithm for the Rectilinear Minimum Link-Distance Problem in Three Dimensions (Extended Abstract) (Dartmouth Computer Science Technical Report TR2005-538)

Robert Scot Drysdale * Clifford Stein † David P. Wagner * ‡

Abstract

In this paper we consider the Rectilinear Minimum Link-Distance Problem in Three Dimensions. The problem is well studied in two dimensions, but is relatively unexplored in higher dimensions. We solve the problem in $O(\beta n \log n)$ time, where n is the number of corners among all obstacles, and β is the size of a BSP decomposition of the space containing the obstacles. It has been shown that in the worst case $\beta = \Theta(n^{3/2})$, giving us an overall worst case time of $O(n^{5/2} \log n)$. Previously known algorithms have had worst-case running times of $\Omega(n^3)$.

1 Introduction

The Minimum Link-Distance Problem (sometimes referred to as the Minimum Bends Path Problem) attempts to find a path between two points which has a minimum number of turns. Equivalently, one can minimize the number of straight line segments, or "links", in the path. An entire chapter in [8] is devoted to the subject of link-distance problems. Rectinlinear versions of this problem received considerable attention during the early and mid 1990's, notably including [9, 5, 11, 7, 2]. Most of these treatments have been confined to just two dimensions, since much of the interest in this problem has been motivated by applications in VLSI[11].

A notable exception comes from deBerg, et al. who have considered rectilinear variants of this problem in higher dimensions [2]. They describe an $O(n^d \log n)$ method to solve a combined metric problem in d dimensions, where the objective is to minimize the total distance travelled, plus some constant C times the total number of bends in the path. Setting C to zero or to a sufficiently high number solves the Euclidean Shortest Path Problem, and the Minimum Link-Distance Problem, respectively.

*Department of Computer Science, Dartmouth College, {scot,dwagn}@cs.dartmouth.edu

†Department of Industrial Engineering and Operations Research, Columbia University, cliff@ieor.columbia.edu

‡Portions of this were written at Korea Advanced Institute for Science and Technology (KAIST)

Here we focus on the Minimum Link-Distance problem with rectilinear (axis-parallel) paths among rectilinear obstacles (with axis-perpendicular faces) in three dimensions. A motivation for our consideration of this problem has been its application to Homogeneous Modular Robots, as described by Fitch, et al. in [3].

Our algorithm improves on the previously best known bound of $O(n^3)$ (see [3] and Section 2.1), running in in $O(\beta n \log n)$ time. It has been shown that in the worst case $\beta = \Theta(n^{3/2})$ [4, 10], however in many practical circumstances, $\beta \approx \Theta(n)$.

For a more detailed version of this paper visit:

http://www.cs.dartmouth.edu/~dwagn/minlinkpath

2 Preliminaries

Other authors have discussed how to efficiently divide rectinlinear polygons with n corners into $O(n)$ rectangles [6, 1], and so we assume that our input obstacle faces have already been thus partitioned. We refer to one such rectangle as a "Subface".

2.1 An $O(n^3)$ algorithm

There exists a straightforward algorithm solving this problem in $O(n^3)$ time. Given a set of rectangular obstacle subfaces, define its grid to be the arrangement of all planes which contain a subface. This grid divides space into $O(n^3)$ cells, each of which lies either entirely within an obstacle or entirely outside of all obstacles [2].

The algorithm then is similar to a breadth-first search through those cells which lie outside of the obstacles. Two important differences are that repeating a step to another cell in the same direction does not increment the distance to that cell, and we do not immediately eliminate a cell from consideration via an entry direction, if it has only been reached by other entry directions.

2.2 Overview of our Algorithm

Our algorithm follows a similar approach to the $O(n^3)$ algorithm, but saves time by using a Binary Space Partition decomposition of space[4, 10]. The leaves of the tree representing this decomposition all designate subspaces

which are either entirely within an obstacle or entirlely outside obstacles. We refer to the subspace represented by such a leaf as a "Block". We call the blocks outside of obstacles "Empty Blocks" and the blocks which are within an obstacle "Obstacle Blocks". The problem then reduces to searching for an optimal path through the set of empty blocks.

Our algorithm finds all points reachable by paths with zero bends, then by paths with one bend, then by paths with two bends, and so on, until the finish point is found.

These paths are found during a series of sweep plane operations. We begin with six sweeps, each of which follows the zero-bend path from the starting point in a particular cardinal direction. We then perform six more sweeps, each of which follows the one-bend paths whose last segment is in a particular direction, and so on.

The sweep operation for a bend distance b and direction d keeps track of all points on the sweep plane that are reachable in exactly b bends, with the last segment travelling in direction d. As the plane sweeps across the various blocks, it is updated by adding or removing regions of reachable points. When the plane encounters an obstacle we must remove any points that lie in the region corresponding to the obstacle face. When the plane leaves an empty block which has been encountered before, we must add any previously saved outgoing paths of bend distance b to the sweep plane.

Whenever the sweep plane encounters an unvisited or only recently visited empty block we determine the best paths from the entry face to every point on all six exit faces. We store this path information as events for future sweep operations.

3 Preprocessing

3.1 Binary Space Partitioning

Given the set of obstacle subfaces, we first compute a Binary Space Partition. A result of Paterson and Yao describes how to find a BSP decomposition of n orthogonal rectangles in three dimensional space resulting in a BSP which contains $O(n^{3/2})$ fragments of the original rectangles. This gives us a tree with $O(n^{3/2})$ leaves [10].

Note, the starting and ending points are considered zero-dimensional obstacles, and so get their own leaves.

Theorem 1 *Space can be subdivided into $\beta = O(n^{3/2})$ blocks, including both obstacle blocks and empty blocks, by a Binary Space Partition[10].*

3.2 Neighbors in a BSP Tree

Our algorithm needs to know which blocks are adjacent to each other in space. We will call such pairs neighbors. The running time of preprocessing is dominated by the amount of time expended building a neighbor graph. The time spent in sweep plane operations also depends in part on the size of the neighbor graph.

Lemma 2 *An axis-perpendicular plane can intersect at most $O(n)$ blocks of the BSP space decomposition defined in [10].*

Proof. This can be derived from [10]. □

Theorem 3 *There are at most $O(\beta n)$ neighbor relationships between pairs of blocks, and between empty blocks and obstacle subfaces.*

Proof. By Lemma 2 each face of a block and each obstacle subface can have at most $O(n)$ neighbors. There are $O(n)$ obstacle subfaces, and so the total number of blocks given in Theorem 1 then limits the total number of neighbor relationships to $O(\beta n)$. □

4 Paths through Blocks

Within a single block, there can be several variations in the optimal bend distance to different places within the block. We would like to examine the kinds of optimal paths which can travel through an empty block.

An examination of a block then must answer the following question: "Given the set of points on a single face of a block through which optimal paths can enter with a particular bend distance, what configuration of exit points could be generated by optimal paths, and what are the optimal bend distances to those points?"

4.1 Classification of Paths through a Block

We define three kinds of paths which could travel through a block, based on their exit face:

Definition 4 (Through Paths) *Paths exiting the block through the face opposite to the one through which they entered.*

Definition 5 (Right Angle Paths) *Paths exiting the block through one of the four faces which are not parallel to the entry face (We use "Right Angle" here to indicate the angle between the entry and exit faces).*

Definition 6 (U-Turn Paths) *Paths exiting the block through the same face through which they entered.*

We can also classify sets of paths into three configurations of points reachable on an exit face. We define those classes with respect to a single entry point:

Definition 7 (Class A Path) *Paths which can exit anywhere through the exit face.*

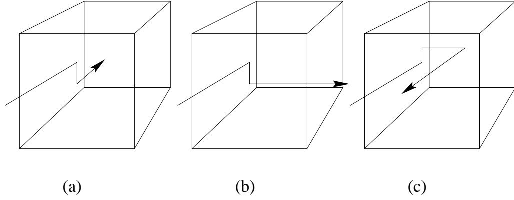

Figure 1: Three kinds of paths, (a) a Through Path, (b) a Right Angle Path, and (c) a U-Turn Path

Definition 8 (Class B Path) *Paths which can exit through a stripe across the exit face.*

Definition 9 (Class C Path) *Paths which can exit through a single point on the exit face.*

The following table describes all relations between the exit face of a path, the number of bends the path takes within a block, and the configuration of exit points:

# of bends	0	1	2	3
Through Paths	C	-	B	A
Right Angle Paths	-	B	A	-
U-Turn Paths	-	-	B	A

4.2 Generating Exit Paths

Given an entry face with a set of entry points, we would like to generate the appropriate sets of exit points resulting from the paths described above. This configuration depends on the class of path being considered: A, B, or C. All paths fall into one of these classes, so it is sufficient to only consider these paths.

`Class A` paths can exit anywhere on the exit face. Thus, from any entry point or points, the complete rectangular outgoing face of the block is the data generated.

`Class B` paths have the property that they must remain in one of the two axis parallel planes perpendicular to the entry face which contains the entry point. Such a path can exit at any point in one of the segments formed at the intersection of a plane containing the entry point and an exit face.

The data generated by all Class B paths coming from all points in the entry face and exiting through a specific exit face can be described by projecting the entry data onto one of its axes, and then striping the outgoing face according to the resulting set of intervals along the axis.

All six faces have Class B paths exiting them. The entry face, and the face opposite it, each have two sets of Class B paths exiting through them, one set striped in each direction. The remaining faces can only be striped in a single direction.

`Class C` paths generate outgoing points which are identical in configuration to the set of incoming points. Unfortunately, this data could be complex, so copying it in its entirety at every block would be prohibitively time consuming. We therefore will employ a sweep plane, described in section 5, which will carry the same data forward from block to block without copying.

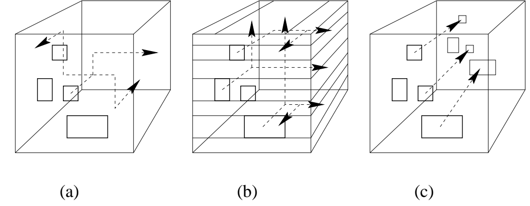

Figure 2: Some examples of the three classes of paths, (a) Class A, (b) Class B, and (c) Class C.

5 Sweep Plane

As outlined in Section 2.2, we will maintain a sweep plane, during each of a series of sweep operations. The regions stored in the plane contain points which are reachable by paths of a particular bend distance, and whose last segment is in the direction of the sweep. We store this data as a two-dimensional segment tree.

5.1 Two-Dimensional Segment Trees

We will need a data structure to describe the sweep plane, as well as the sets of paths which can enter and exit an empty block. For these we employ a two-dimensional segment tree. This data structure is designed to store rectangles, and to support efficient queries. The advantage of using this data structure is that a set of rectangles can be stored efficiently, even if the rectangles are criss-crossing and overlapping.

Unlike traditional segment trees, our two-dimensional segment trees do not distinguish between rectangles after they have been inserted. Rather the union of all inserted regions is kept.

We will use the following segment tree operations :

- `InsertRect` inserts a rectangle into the tree.
- `InsertStripedRect` inserts a set of $O(n)$ stripes which are contained within, and extend the length or width of a given rectangle. The stripes vary in the other dimension according to the set of intervals defined in a given one-dimensional segment tree.
- `QueryRect` queries a rectangle in the segment tree, returning `TRUE` if it overlaps with previously inserted data.
- `ClearRect` clears all data within the specified rectangle (Note this may fragment previously inserted rectangles).
- `ProjectRect` projects all data within the given rectangle onto an axis, returning a one-dimensional segment tree containing the resulting intervals.

Some auxilliary data is maintained in the nodes of the two-dimensional segment tree in order to support efficient projections and queries. Each of the above operations then runs in amortized $O(n \log n)$ time or better.

5.2 Events

There are three types of events that the sweep may encounter. We define a function `SweepPlaneEvent` which processes these events based on the event type:

- `OutgoingPath`: A set of paths leaving a block in the direction and bend distance of the sweep are encountered. Add them to the sweep plane, by calling `InsertRect` or `InsertStripedRect`, for Class A and B paths respectively. Neighboring empty blocks and subfaces are inserted into the queue.
- `ObstacleFace`: A subface is encountered, and so all paths which intersect that face must be deleted from the sweep plane, using `ClearRect`.
- `EmptyBlock`: An empty block is encountered. If it was discovered more than three bends ago, treat it like an obstacle, since all optimal paths through the block have already been generated. If it was discovered less than three bends ago, query the sweep plane, via a call to `QueryRect`, to see if any paths enter that block. If so, then outgoing paths are generated, using `ProjectRect`, and neighboring blocks and obstacle subfaces are inserted into the queue.

6 Algorithm

```
Min-Link-Path(obsFaces, s, t)
 1) Q = new PriorityQueue
 2) sweepPlane = new SegTree2D
 3) blocks = Preprocess(obsFaces, s, t)
 4) InitializeQ(Q, s)
 5) benddist = 0; dir = 0
 6) While not EmptyPriorityQ(Q)
 7)     event = RemoveMinPriorityQ(Q)
 8)     If (event.benddist ≠ benddist) or
                (event.dir ≠ dir) then
 9)         ClearSegTree2D(sweepPlane)
 10)        benddist = event.benddist
 11)        dir = event.dir
 12)    SweepPlaneEvent(Q, sweepPlane, event,
            benddist, dir)
 13) Output t.benddist
```

6.1 Runtime Analysis

The running time of this algorithm depends on the running time of all invocations of `SweepPlaneEvent`.

Each empty block only generates outgoing paths during sweeps having three different bend distances. After that the block is treated as an obstacle. Therefore each block can only generate a constant number of sets of outgoing paths, and we have $O(\beta)$ `OutgoingPath` events.

An obstacle subface is both rectangular and completely uncovered. Therefore, its neighbors are empty blocks, and there exists a path having at most three bends between any two points within this set of blocks. This implies that the minimum bend distance to any two of these neighbors differs by at most a constant. Thus, each obstacle subface will be added to the queue during at most a constant number of sweeps, and there can only be $O(n)$ `ObstacleFace` events. Similarly, there can only be $O(\beta)$ `EmptyBlock` events.

Each of these events invokes a constant number of two-dimensional segment tree operations. Each operation has amortized $O(n \log n)$ running time, the total time taken by all `SweepPlaneEvent` calls is $O(\beta n \log n)$.

Theorem 10 *Our algorithm finds the Minimum Link-Distance Path in $O(\beta n \log n)$ time.*

Acknowledgements

We are grateful to Robert Fitch, Lisa Fleischer, Joseph S. B. Mitchell, Der-Tsai Lee, and Takeshi Tokuyama for their helpful discussions. We are also grateful to Kyung-Yong Chwa for the contribution of his lab space at Korea Advanced Institute for Science and Technology in which early versions of this paper were written.

References

[1] B. Chazelle. *Computational Geometry and Convexity.* PhD thesis, Yale University, New Haven, CT, 1980.

[2] M. de Berg, M. J. van Kreveld, B. J. Nilsson, and M. H. Overmars. Shortest path queries in rectilinear worlds. *International Journal of Computational Geometry and Applications*, 2(3):287–309, 1992.

[3] R. Fitch, Z. Butler, and D. Rus. 3D rectilinear motion planning with minimum bend paths. In *International Conference on Intelligent Robots and Systems*, 2001.

[4] J. Hershberger and S. Suri. Binary space partitions for 3d subdivisions. In *Symposium on Discrete Algorithms*, pages 100–108, 2003.

[5] D. Lee, C. Yang, and C. Wong. Rectilinear paths among rectilinear obstacles. *Discrete Applied Mathematics*, 70, 1996.

[6] C. Levcopoulos and A. Lingas. Bounds on the length of convex partitions of polygons. In *Foundations of Software Technology and Theoretical Computer Science*, pages 279–295, 1984.

[7] A. Lingas, A. Maheshwari, and J. Sack. Optimal parallel algorithms for rectilinear link-distance problems. *Algorithmica*, 14(3):261–289, 1995.

[8] A. Mahehswari and J. Sack. Link distance problems. In *Handbook of Computational Geometry*, chapter 12. Elsevier Science Pub Co, 2000.

[9] J. Mitchell, C. Piatko, and E. Arkin. Computing a shortest k-link path in a polygon. In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, pages 573–582, 1992.

[10] M. S. Paterson and F. F. Yao. Optimal binary space partitions for orthogonal objects. *Journal of Algorithms*, 13:99–113, 1992.

[11] C. Yang, D. Lee, and C. Wong. On bends and distance paths among obstacles in two-layer interconnection model. *IEEE Transactions on Computers*, 43:711–724, 1994.