A Geometric Shortest Path Problem, with Application to Computing a Longest Common Subsequence in Run-Length Encoded Strings

Joseph S. B. Mitchell*

1 Introduction

Let R denote an axis-aligned rectangle in the plane. Without loss of generality, we assume that R is the region $\{(x,y) : 0 \leq x \leq M, 0 \leq y \leq N\}$. Within R there are n pairwise-disjoint axis-aligned rectangles, $R_1, R_2, \ldots, R_n \subset R$. We refer to the region $B = \cup_i R_i$ as "blue", and the complement, $R \setminus B$, as "red". Our goal is to travel from the upper left corner $s = (0, N)$ to the lower right corner $t = (M, 0)$, while minimizing the total length of travel in the red region. We are constrained to travel as follows:

(a) In the red region, we must travel horizontally to the right, or vertically downwards.

(b) In the blue regions, we must travel along a line of slope -1.

In this paper, we give an $O(n \log n)$ time algorithm for this problem, based on methods we developed earlier for rectilinear and "fixed orientation" shortest paths in the plane. In particular, we base our algorithm on the "continuous Dijkstra" paradigm, as we developed it in Mitchell [4, 5]. See [3] for an extensive survey of shortest path results in geometry.

Our study is motivated by the problem of computing a longest common subsequence (LCS) of run length encoded strings, $X = X_1 X_2 \ldots X_l$ and $Y = Y_1 Y_2 \ldots Y_k$, where each X_i or Y_j represents a character, with its *run length*. For example, we write $X_1 = a^8$ if the first 8 characters of string X are the letter "a", and the ninth character is not an "a". If $X = aaaaaaaabbfffgddddaaabbbbb$, then we write $X = a^8 b^2 f^3 g^1 d^5 a^3 b^5$ as the run length encoded string. We let M (resp., N) denote the total number of characters in string X (resp., Y). If X_i is a run of the same characters as is Y_j, then we create a rectangular ("blue") block corresponding to (i, j), whose x-interval (resp., y-interval) is given by the indices of the characters in X_i (resp., Y_j). Then, the problem of finding a longest common subsequence between X and Y is readily seen to be equivalent to the optimal path problem mentioned above: we must move from upper left corner to lower right corner of R, while minimizing the number of mismatches (i.e., the number of steps taken in the red (non-matching) region). By sorting the set of characters, it is easy to obtain the set of n blue blocks $(R_1, \ldots, R_n)$ that correspond the the shortest path instance, in time $O(n + (k+l)\log(k+l))$.

Related Work. In the time since our results were first obtained, an independent effort by Apostolico, Landau, and Skiena [1] has resulted in some similar results. In particular, they obtain a time bound of $O(kl \log(kl))$. Since $n \leq kl$, our result $(O((n+k+l)\log(n+k+l)))$ compares favorably to theirs, improving the worst-case time bound, particularly when n is much smaller than its worst-case upper bound kl.

*jsbm@ams.sunysb.edu; http://www.ams.sunysb.edu/~jsbm/. Department of Applied Mathematics and Statistics, State University of New York, Stony Brook, NY 11794-3600. Partially supported by NSF grant CCR-9504192, and by grants from Boeing Computer Services, Bridgeport Machines, Hughes Aircraft, and Sun Microsystems.

2 The Algorithm

We follow the continuous Dijkstra paradigm: We want to track a "front", parameterized by d, that corresponds to points within R that are known to be within distance d of the source point s, where "distance" is measured according to the length only of travel in the red region, and the path is constrained to travel according to (a) and (b) above.

At any given stage of the algorithm, we have a set of *dragged segments* that represent pieces of the front. Each is inclined at an angle of 45 degrees with respect to the positive x-axis, and is advancing down and to the right. As in [4, 5], the endpoints of a dragged segment slide along *track rays*, which are axis-parallel, pointing downwards or rightwards. Each dragged segment has a *root* point, which is labeled with the distance from s. All points on a dragged segment are below and to the right of the root, and are at the same (rectilinear) distance from the root.

Events correspond to

(Type I) A dragged segment hitting a vertex of some blue rectangle R_i, at a point interior to the dragged segment; or

(Type II) An endpoint of a dragged segment hitting the boundary of a blue rectangle R_i, or the outer boundary of R.

We can determine events of either type in time $O(\log n)$, by the same methods as in [4, 5]. (If the track rays are parallel to each other, then we use segment-dragging queries by the methods of Chazelle [2]; if the tracks are orthogonal to each other, then we use point location queries in the subdivision described in [4, 5].)

The algorithm is simply this: Process events in order of increasing distance from s, until t is reached by some dragged segment.

To process an event of Type I, we distinguish two cases, depending on whether or not the hit vertex, v, has already been hit previously. If it has, then we *clip* the dragged segment, according to the track ray of the dragged segments that have previously hit it, as in [4, 5]. If it has not been hit previously, then we label it with its newly discovered distance from s. (The Dijkstra property of the algorithm assures us that the label we give it is correct.) We also instantiate a new dragged segment rooted at v', the point on the boundary of R_i where a ray fired southeast from v exits R_i. Finally, we clip the dragged segment on both the top of R_i and the left side of R_i, resulting in two dragged segments still rooted at the same point that the original one was.

To process an event of Type II, we create a labeled point at the point p where the endpoint of the dragged segment hits R_i, and we instantiate a new dragged segment, rooted at p', the point on the boundary of R_i where a ray fired southeast from p exits R_i.

The proof of correctness is based on observing that we are conservative in our clipping of dragged segments: we only clip the propagation when we have cause to do so.

The proof of complexity involves bounding the total number of events to be $O(n)$, and noting that each event is processed in time $O(\log n)$. The crucial thing to examine is the number of times that a dragged segment *re*-hits a vertex, causing clipping. (The first time a vertex of some R_i is hit, we charge the event to that vertex.) However, each time a dragged segment, rooted at r, hits a vertex v that has already been hit, we can charge the event to a vertex of the shortest path map, which is a planar subdivision of size $O(n)$.

Theorem 1 *An optimal path, minimizing travel in the red region, can be computed in time $O(n \log n)$, using $O(n)$ space. In fact, within these same bounds, the shortest path map with respect to source s can be constructed.*

Corollary 2 *The LCS problem on run length encoded strings can be solved in time $O((n+k+l)\log(n+k+l))$.*

Extensions. It is easy to extend our results to more general settings. In particular, the blue regions need not be axis-aligned rectangles; if they are disjoint polygons, having a total of n vertices, the same results apply. Further, if the travel in the blue regions is not at slope -1, but at any other negative slope, the same results apply. This, in particular, allows the LCS problem to be solved in a more general setting, allowing for more general edit distances.

In fact, for any decomposition of the plane into polygonal regions of two (or more) colors, and with an convex distance function defined within each region, our methods should yield an $O(n \log n)$ time algorithm. Are there applications for this more general class of shortest path problems?

Acknowledgements

I thank Tim Chen for pointing out this problem to me, and for helpful discussions in the formulation of the shortest path version of the LCS problem.

References

[1] A. Apostolico, G. M. Landau, and S. S. Skiena. Matching for run length encoded strings. Manuscript, Stony Brook, 1997.

[2] B. Chazelle. An algorithm for segment-dragging and its implementation. *Algorithmica*, 3:205–221, 1988.

[3] J. Mitchell. Shortest paths and networks. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 24, pages 445–466. CRC Press LLC, Boca Raton, FL, 1997.

[4] J. S. B. Mitchell. An optimal algorithm for shortest rectilinear paths among obstacles. In *Abstracts 1st Canad. Conf. Comput. Geom.*, page 22, 1989.

[5] J. S. B. Mitchell. L_1 shortest paths among polygonal obstacles in the plane. *Algorithmica*, 8:55–88, 1992.