Finding an Optimal Path without Growing the Tree*

Danny Z. Chen, Ovidiu Daescu, Xiaobo (Sharon) Hu, and Jinhui Xu

Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556, USA
{chen,odaescu,shu,jxu}@cse.nd.edu

Abstract. In this paper, we study a class of optimal path problems with the following phenomenon: The *space* complexity of the algorithms for reporting the *lengths* of single-source optimal paths for these problems is asymptotically smaller than the space complexity of the "standard" tree-growing algorithms for finding actual optimal paths. We present a general and efficient algorithmic paradigm for finding an actual optimal path for such problems without having to grow a single-source optimal path tree. Our paradigm is based on the "marriage-before-conquer" strategy, the prune-and-search technique, and a data structure called *clipped trees*. The paradigm enables us to compute an actual path for a number of optimal path problems and dynamic programming problems in computational geometry, graph theory, and combinatorial optimization. Our algorithmic solutions improve the space bounds (in certain cases, the time bounds as well) of the previously best known algorithms, and settle some open problems. Our techniques are likely to be applicable to other problems.

1 Introduction

For combinatorial problems on computing an optimal path as well as its length, the "standard" approach for finding an actual optimal path is by building (or "growing") a single-source optimal path tree. This tree-growing approach is effective for finding actual single-source optimal paths, especially as the *time* complexity is concerned. In fact, it is well-known that no general algorithms are known that compute an optimal path between *one pair* of locations with a faster *time* complexity than that for computing single-source optimal paths. In this paper, we study a class of optimal path problems with the following interesting yet less-exploited phenomenon: The *space* complexity of the algorithms for reporting the *lengths* of single-source optimal paths for these problems is asymptotically smaller than the space complexity of the "standard" tree-growing algorithms for finding actual optimal paths. Our goal is to show that for such problems, it is

* The work of the first, second, and fourth authors was supported in part by the National Science Foundation under Grant CCR-9623585. The work of the third author was supported in part by the National Science Foundation under Grant MIP-9701416 and by HP Labs, Bristol, England under an external research program grant.

possible to find an actual optimal path without having to grow a single-source optimal path tree, thus achieving asymptotically better space bounds for finding one actual optimal path than those for single-source optimal paths.

It should be mentioned that the phenomenon that the space bound for finding an actual optimal path can be smaller than that for single-source optimal paths has been observed and exploited in some scattered situations. For example, Edelsbrunner and Guibas [9] showed that for computing a longest monotone path or a longest monotone concave path on the arrangement of size $O(n^2)$ formed by n lines on the plane, it is possible to report the *length* of such a path in $O(n^2)$ time and $O(n)$ space. To output an actual longest monotone path, they used $O(n^2 \log n)$ time and $O(n \log n)$ space, and to output an actual longest monotone concave path, they used $O(n^2 \log n)$ time and $O(n \log n)$ space (or alternatively, $O(n^3)$ time and $O(n)$ space). It was posed as open problems in [9] whether these extra time and space bounds for reporting an actual longest monotone path or longest monotone concave path could be partially or completely avoided. Another example is the problem of computing a longest common subsequence of two strings of size n [6, 14, 19] (this problem can be reduced to an optimal path problem). Hirschberg [14] used dynamic programming to find an actual longest common subsequence and its length in $O(n^2)$ time and $O(n)$ space without growing a single-source tree. The actual optimal path algorithms in [9] use a recursive back-up method, and the one in [14] is based on a special divide-and-conquer strategy called "marriage-before-conquer".

We study in a systematic manner the phenomenon that the space bound for finding an actual optimal path can be smaller than that for single-source optimal paths. We develop a general algorithmic paradigm for reporting an actual optimal path without using the tree-growing approach, and characterize a class of optimal path and dynamic programming problems to which our paradigm is applicable. This paradigm not only considerably generalizes the marriage-before-conquer strategy used in [14], but also brings forward additional interesting techniques such as prune-and-search and a new data structure called *clipped trees*. Furthermore, the paradigm makes it possible to exploit useful structures of some of the problems we consider. Our techniques enable us to compute efficiently an actual optimal solution for a number of optimal path and dynamic programming problems in computational geometry, graph theory, and combinatorial optimization, improving the space bounds (in certain cases, the time bounds as well) of the previously best known algorithms. Below is a summary of our main results on computing an actual optimal solution.

Computing a shortest path in the arrangement of n lines on the plane. As mentioned in [4, 12], it is easy to reduce this problem to a shortest path problem on a planar graph of size $O(n^2)$ that represents the arrangement, and then solve it in $O(n^2)$ time and space by using the optimal shortest path algorithm for planar graphs [15]. We present an $O(n^2)$ time, $O(n)$ space algorithm.

Computing a longest monotone concave path in the arrangement of n lines on the plane. An $O(n^2 \log n)$ time, $O(n \log n)$ space algorithm and an $O(n^3)$ time, $O(n)$ space algorithm were given by Edelsbrunner and Guibas [9]. We present an $O(n^2)$ time, $O(n)$ space algorithm. Our solution is an improvement on those of [9], and settles the corresponding open problem in [9].

Computing a longest monotone path in the arrangement of n lines on the plane. An $O(n^2 \log n)$ time, $O(n \log n)$ space algorithm and an $O(\frac{n^2}{\epsilon})$ time, $O(\frac{n^{1+\epsilon}}{\epsilon})$ space algorithm were given by Edelsbrunner and Guibas [9]. We present an $O(\frac{n^2 \log n}{\log(h+1)})$ time, $O(nh)$ space algorithm, where h is any integer such that $1 \leq h \leq n^{\epsilon}$ for any constant ϵ with $0 < \epsilon < 1$. Note that for $h = O(1)$, our algorithm uses $O(n^2 \log n)$ time and $O(n)$ space, and for $h = n^{\epsilon}$, our algorithm uses $O(\frac{n^2}{\epsilon})$ time and $O(n^{1+\epsilon})$ space (unlike [9], our space bound does not depend on the $\frac{1}{\epsilon}$ factor). Our solution is an improvement on those of [9], and provides an answer to the corresponding open problem in [9].

Computing a longest monotone path in the arrangement of n planes in the 3-D space. An $O(n^3)$ time, $O(n^2)$ space algorithm was given by Anagnostou, Guibas, and Polimenis [1] for computing the *length* of such a path. If the techniques in [9] are used, then an actual path would be computed in $O(n^3 \log n)$ time and $O(n^2 \log n)$ space. We present an $O(\frac{n^3 \log n}{\log(h+1)})$ time, $O(n^2 h)$ space algorithm, where h is any integer such that $1 \leq h \leq n^{\epsilon}$ for any positive constant $\epsilon < 1$.

Computing a minimum-weight, k-link path in a graph. A standard tree-growing approach uses $O(k(n+m))$ time and $O(kn)$ working space to compute a minimum-weight, k-link path in an edge-weighted graph of n vertices and m edges. We present an $O(\frac{k(n+m) \log k}{\log(h+1)})$ time, $O(nh)$ working space algorithm, where h is any integer such that $1 \leq h \leq k^{\epsilon}$ for any constant ϵ with $0 < \epsilon < 1$. Note that for $h = O(1)$, our algorithm uses $O(k(n+m) \log k)$ time and $O(n)$ working space, and for $h = k^{\epsilon}$, our algorithm uses $O(\frac{1}{\epsilon} k(n+m))$ time and $O(nk^{\epsilon})$ working space (the constant of the working space bound does not depend on $\frac{1}{\epsilon}$). Furthermore, if G is a directed acyclic graph, then our algorithm uses $O(k(n+m))$ time and $O(n)$ working space.

0–1 knapsack with integer item sizes. The 0–1 knapsack problem is NP-complete and has often been solved by dynamic programming [17] or by reducing the problem to computing an optimal path in a directed acyclic graph of $O(nB)$ vertices and edges. If a standard tree-growing approach is used for computing an actual solution, then it would use $O(nB)$ time and space [17] (it was also shown in [17] how to use a bit representation to reduce the space bound to $O(\frac{nB}{\log(n+B)})$). We present an $O(nB)$ time, $O(n+B)$ space algorithm.

Single-vehicle scheduling. The general problem is to schedule a route for a vehicle to visit n given sites each of which has a time window during which the vehicle is allowed to visit that site. The goal is to minimize a certain objective function of the route (e.g., time or distance), if such a route is possible. This problem is clearly a generalization of the Traveling Salesperson Problem and is NP-hard even for some very special cases. For example, it is NP-hard for the case in which a vehicle is to visit n sites on a straight line (equivalently, a ship is to visit n harbors on a convex shoreline) with time windows whose start times and end times (i.e., *deadlines*) are arbitrary [5]. Psaraftis *et al.* [18] gave an $O(n^2)$ time and space dynamic programming algorithm for the case with n sites on a straight line whose time windows have only (possibly different) start times. Chan and Young [5] gave an $O(n^2)$ time and space dynamic programming algorithm for the case with n sites on a straight line whose time windows have the same start time but various deadlines. We present $O(n^2)$ time, $O(n)$ space algorithms for both these cases.

Due to the space limit, quite a few of our algorithms must be left to the full paper. Also, the proofs of our lemmas are left to the full paper.

2 Clipped Trees

A key ingredient of our general paradigm is the data structure called *clipped trees* that we introduce in this section. Clipped trees are important to our paradigm because they contain information needed for carrying out techniques such as marriage-before-conquer and prune-and-search.

In a nutshell, a clipped tree T is a "compressed" version of a corresponding single-source optimal path tree *SST*, such that T consists of a (usually sparse) sample set of the nodes of *SST* and maintains certain topological structures of *SST*. The sample nodes are selected from *SST* based on a certain criterion (e.g., geometric or graphical) that depends on the specific problem.

Let T' be a rooted tree with root node r. Let S be a set of sample nodes from T' with $r \in S$. A clipped tree T of T' based on the sample set S is defined as follows:

- The nodes of the clipped tree T are precisely those in S.
- For every node $v \in S - \{r\}$, the parent of v in T is the nearest proper ancestor w of v in T' such that $w \in S$.

Clearly, the size of T is $O(|S|)$. If S consists of all the nodes of T', then T is simply T' itself. The clipped tree T of T' can be obtained by the following simple procedure:

- Make the root r of T' the root of T, and pass down to all children of r in T' a pointer to r.
- For every node v of T' that receives from its parent in T' a pointer to a proper ancestor node w of v in T' (inductively, w is already a node of T), do the following: If $v \in S$, then add v to T, make w the parent of v in T, and pass down to all children of v in T' (if any) a pointer to v; otherwise, pass down to all children of v in T' (if any) the pointer to w.

It is easy to see that it takes $O(|T'|)$ time to construct the clipped tree T from T' and the sample set S, and $O(|S|)$ space to store T. Also, observe that the above procedure need not have the tree T' explicitly stored. In fact, as long as the nodes of T' are produced in any parent-to-children order, T can be constructed. Note that this is precisely the order in which a single-source optimal path tree grows, and this growing process takes place in the same time as the lengths of optimal paths are being computed. Further, observe that one need not have the sample set S explicitly available in order to construct T. As long as a criterion is available for deciding (preferably in $O(1)$ time) whether any given node v of T' belongs to the sample set S, the above procedure is applicable.

Consequently, one can use an algorithm for computing the lengths of single-source optimal paths and a criterion for determining the membership for a sample set S of the nodes of the single-source optimal path tree *SST* to construct a clipped tree T based on *SST* and S, without having to store *SST*. Actually, when T is being constructed, it is beneficial to associate with the nodes of T some information about the corresponding optimal paths to which these nodes belong. Once the process of computing the lengths of single-source optimal paths terminates, the clipped tree T, together with useful optimal path information stored in its nodes, is obtained.

Perhaps we should point out a seemingly minor but probably subtle aspect: The above procedure for building a clipped tree depends only on the ability to generate a single-source optimal path tree in a parent-to-children (or source-to-destination) order. This is crucial for the applicability of our general paradigm. In contrast, the marriage-before-conquer algorithm in [14] computes an actual optimal path using both the source-to-destination and destination-to-source orders. Although the problem in [14] is symmetric with respect to these two orders, it need not be the case with many other optimal path problems. For example, for some dynamic programming problems that are solvable by following a source-to-destination order (e.g., [5, 18]), it may be quite difficult or even impossible to use the destination-to-source order. This aspect of clipped trees also enables us to avoid using the recursive back-up method of [9], since it may be difficult to use this back-up method to significantly reduce the sizes of the subproblems in a marriage-before-conquer algorithm.

3 Shortest Paths in an Arrangement

In this section, we illustrate our algorithmic paradigm with algorithms for finding a shortest path and its length between two points in the arrangement of lines on the plane. The problem can be stated as follows: Given a set H of n planar lines and two points s and t on some lines of H, find an s-to-t path of the shortest Euclidean distance that is restricted to lie on the lines of H. As mentioned in [4, 12], to solve this geometric shortest path problem, one can construct a planar graph of size $O(n^2)$ that represents the arrangement of H and then apply the optimal algorithm for computing a shortest path in a planar graph [15]. Such an algorithm (even for the path *length*) uses $O(n^2)$ time and space, and it has been an open problem to improve these bounds. Although we are not yet able to improve the asymptotic time bound, we show how to reduce the space bound by a factor of n. Our first algorithm reports the length of the shortest s-to-t path in $O(n)$ space and $O(n \log n + K)$ time. Our second algorithm finds an actual shortest path in $O(n)$ space and $O(n \log^2 n \log(K/n) + \min\{n^2, K \log n\})$ time. Here, K is the size of the part of the arrangement of H that is inside a special convex polygonal region R (R will be defined below), and $K = O(n^2)$ in the worst case. Hence both our algorithms in the worst case take $O(n)$ space and $O(n^2)$ time.

Our path length algorithm is based on the topological sweep [9] and topological walk [2, 3] techniques on planar arrangements. Our actual path algorithm makes use of additional techniques such as marriage-before-conquer, prune-and-search, and the clipped tree data structure. Our solutions also exploit a number of interesting observations on this particular problem.

3.1 Topological Sweep and Topological Walk

Arrangements are a fundamental structure in combinatorial and computational geometry [8], and a great deal of work has been devoted to studying various arrangements and their properties. Topological sweep [1, 9] and topological walk [2, 3] are two powerful space-efficient techniques for computing and traversing arrangements in a 2-D or 3-D space.

Let $H = \{l_1, l_2, \ldots, l_n\}$ be a set of n straight lines on a plane. The lines in H partition the plane into a subdivision called the *arrangement* $A(H)$ of H. $A(H)$ consists of a set of convex regions (*cells*), each bounded by some edges (i.e., segments of the lines in H) and vertices (i.e., intersection points between the lines). In general, $A(H)$ consists of $O(n^2)$ cells, edges, and vertices. Without loss of generality (WLOG), we assume that the lines in H are in general position, i.e., no three lines meet at the same point (the general case can be handled by using the techniques in [10]).

If one is interested only in constructing and reporting (but not storing) $A(H)$, then this can be done by a relatively easy algorithm that sweeps the plane with a vertical line, in $O(n^2 \log n)$ time and $O(n)$ space [11]. Edelsbrunner and Guibas [9] discovered the novel *topological sweep* approach for constructing and reporting $A(H)$ in $O(n^2)$ time and $O(n)$ space. The topological sweep approach sweeps the plane with an unbounded simple curve that is monotone to the y-axis and that intersects each line of H exactly once. Asano, Guibas, and Tokuyama [2] developed another approach, called *topological walk*, for constructing and reporting $A(H)$ in $O(n^2)$ time and $O(n)$ space. Essentially, a topological walk traverses $A(H)$ in a depth-first search fashion [2, 3]. This approach can be extended to traversing a portion of $A(H)$ that is inside a convex polygonal region R on the plane in $O(K + (n + |R|) \log(n + |R|))$ time and $O(n + |R|)$ space, where K is the size of the portion of $A(H)$ in R and $|R|$ is the number of vertices of R.

3.2 Computing Shortest Path Lengths

We begin with some preliminaries. Let s and t be the source and destination points on the arrangement $A(H)$ for the sought shortest path. Let $\overline{st}$ be the line segment connecting s and t. WLOG, assume $\overline{st}$ is horizontal with s as the left end vertex. Let $H_c(\overline{st})$ be the set of lines in H that intersect the interior of $\overline{st}$, called the *crossing lines* of $\overline{st}$. Let $HP(H - H_c(\overline{st}))$ be the set of half-planes each of which is bounded by a line in $H - H_c(\overline{st})$ and contains $\overline{st}$. As observed in [4], since no shortest path in $A(H)$ can cross a line in H twice, one can restrict the search of a shortest s-to-t path to the (possibly unbounded) convex polygonal region R that is the common intersection of the half-planes in $HP(H - H_c(\overline{st}))$. Hence, the problem of finding a shortest s-to-t path in $A(H)$ can be reduced in $O(n \log n)$ time to that of finding a shortest s-to-t path in the portion of $A(H)$ contained in R (by computing the common intersection of the half-planes in $HP(H - H_c(\overline{st}))$ and identifying the crossing lines of $\overline{st}$). Henceforth, we let n denote the number of lines of H intersecting the convex region R and let A_R denote the portion $A(H) \cap R$ of $A(H)$.

We use topological walk, starting at s, to report the length of the shortest s-to-t path in A_R. In fact, our algorithm correctly reports the length of the shortest path from s to every vertex in A_R. The following is a simple yet useful lemma to our algorithm.

Lemma 1. *For any line $l \in H$ and any vertex v of $A(H)$ on l, no shortest s-to-v path in $A(H)$ can cross l (i.e., intersecting the interior of both the half-planes bounded by l).*

We use Lemma 1 in conjunction with the fact that, for a line $l \in H_c(\overline{st})$ and a vertex v of A_R on l, the topological walk visits v by following paths in A_R that stay on the left half-plane of l before visiting v on any path that crosses l (this follows from the definition of the topological walk [2, 3]). By incorporating the computation of shortest path lengths with the construction and traversing of A_R by the topological walk, we obtain an algorithm for computing the lengths of the single-source shortest paths in A_R from the source s. Furthermore, the shortest path lengths are computed in the parent-to-children order in the single-source shortest path tree rooted at s (the details of the algorithm are a little tedious and left to the full paper). Hence we have the following lemma.

Lemma 2. *The length of the shortest path in A_R from s to every vertex of A_R can be computed in $O(n \log n + K)$ time and $O(n)$ space, where K is the number of vertices of A_R.*

3.3 Computing an Actual Shortest Path

In this subsection, we present our $O(n \log^2 n \log(K/n) + \min\{n^2, K \log n\})$ time, $O(n)$ space algorithm for reporting an actual shortest s-to-t path in A_R, where K is the size of A_R. In contrast, this algorithm is more interesting yet more sophisticated than the shortest path length algorithm.

Let v be a vertex on a shortest s-to-t path in A_R such that v is the intersection of two lines l_i and l_j of H and such that at least one of l_i and l_j is a crossing line of $\overline{st}$. Let $SP(s,t)$ denote the shortest s-to-t path in A_R. Then $SP(s,t) = SP(s,v) \cup SP(v,t)$. The following lemmas are a key to our algorithm.

Lemma 3. *The two lines l_i and l_j of H define two interior-disjoint convex subregions R_1 and R_2 in R such that $SP(s,v)$ stays within R_1 and $SP(v,t)$ stays within R_2. Further, at most four lines of H (two of them are l_i and l_j) can appear on the boundaries of both R_1 and R_2.*

Lemma 4. *The lines in H that $SP(s,t)$ crosses are exactly the crossing lines of $\overline{st}$ (i.e., $H_c(\overline{st})$).*

Lemma 5. *Let l_i and l_j be defined as in Lemma 3. The crossing lines of $\overline{st}$ in $H_c(\overline{st}) - \{l_i, l_j\}$ can be partitioned into two subsets H_1 and H_2, such that no line in H_1 (resp., H_2) intersects $SP(v,t)$ (resp., $SP(s,v)$). Moreover, H_1 (resp., H_2) consists of all the lines in $H_c(\overline{st}) - \{l_i, l_j\}$ that intersect the interior of the line segment $\overline{sv}$ (resp., $\overline{vt}$), i.e., $H_1 = H_c(\overline{sv})$ (resp., $H_2 = H_c(\overline{vt})$).*

Lemma 5 implies that if we are to compute $SP(s,v)$ (resp., $SP(v,t)$), the lines in $H_c(\overline{vt})$ (resp., $H_c(\overline{sv})$) need not be considered. Let v_c denote the number of lines in H crossed by $SP(s,v)$ (i.e., $v_c = |H_c(\overline{sv})|$), called the *crossing number* of v. If we could somehow find a vertex v on $SP(s,t)$ such that its crossing number v_c is (roughly) half the crossing number t_c of t, then we would have an efficient marriage-before-conquer algorithm for reporting $SP(s,t)$. This is because we would be able to recursively report the subpaths $SP(s,v)$ and $SP(v,t)$ in R_1 and R_2, respectively (by Lemma 3). Moreover, when computing $SP(s,v)$ and $SP(v,t)$, we would not have to consider the intersections between lines from the two line sets $H_c(\overline{sv})$ and $H_c(\overline{vt})$, thus eliminating from further consideration a constant fraction of the total $O(n^2)$ intersections of $A(H)$ among the n lines.

The next lemma makes it possible for an incremental method to compute the crossing numbers.

Lemma 6. *Let u and w be two neighboring vertices of $A(H)$ on a line $l \in H$ (i.e., $\overline{uw}$ is an edge of $A(H)$ on l). Let the line $l(u)$ (resp., $l(w)$) of H intersect l at u (resp., w). Then $H_c(\overline{su})$ differs from $H_c(\overline{sw})$ on at most two elements. Furthermore, these different elements are in $\{l(u), l(w)\}$.*

Based on Lemma 6, if the crossing number u_c of a vertex u of $A(H)$ is already known, then it is easy to compute w_c for a neighboring vertex w of u in $A(H)$. This immediately implies that the crossing numbers of the vertices of A_R can be computed by a topological walk starting from the source vertex s (with $s_c = 0$). In particular, our shortest path length algorithm can be easily modified to report (but not store) the crossing numbers of the vertices of A_R.

At this point, it might be tempting to try to compute an actual path $SP(s,t)$ with the algorithm below. Let k be half the crossing number t_c of t ($k = \frac{t_c}{2}$ can be obtained by running the shortest path length algorithm on A_R once, as a preprocessing step). Then do the following.

1. If $k = O(\sqrt{n})$, then report $SP(s,t)$ by a tree-growing approach in A_R. Otherwise, continue.
2. Run the path length algorithm on A_R, and build a clipped tree T with sample nodes s, t, and all vertices u of A_R such that u is on a crossing line of $\overline{st}$ (by Lemma 3) and $u_c = k$.
3. From the clipped tree T, find a vertex v on $SP(s,t)$ such that $v_c = k$ (the parent node of t in T is such a vertex).
4. Using the vertex v, recursively report the subpaths $SP(s,v)$ and $SP(v,t)$ in R_1 and R_2.

The above algorithm, however, does not work well due to one difficulty: The size of the sample node set S for T is *super-linear*! The astute reader may have observed that the size of the sample set S is closely related to the well-known problem on the combinatorial complexity of the k-th level of the arrangement of n planar lines. The best known lower bound for the k-th level size of such an arrangement is $O(n\log(k+1))$ [13, 16], and the best known upper bound is $O(nk^{1/3})$ [7]. Hence the clipped tree T based on such a sample set S would use super-linear space, not the desired $O(n)$ space. To resolve this difficulty, we avoid using these vertices u as sample nodes for T such that u is on a crossing line of $\overline{st}$ and $u_c = k$. Instead, we use a prune-and-search approach to locate a vertex v on $SP(s,t)$ such that v is on a crossing line of $\overline{st}$ and $v_c = k$. Our prune-and-search procedure is based on some additional observations and (again) on the clipped tree data structure.

Lemma 7. *For any two vertices u and w of $A(H)$ such that u is on $SP(s,w)$, $w_c \geq u_c$.*

Lemma 8. *It is possible to find, in $O(K + n\log n)$ time and $O(n)$ space, a vertical line L such that L partitions the K vertices of A_R into two subsets of sizes c_1K and c_2K, where c_1 and c_2 are both positive constants and $c_1 + c_2 = 1$.*

Lemma 7 provides a structure on $SP(s,t)$ for searching, and Lemma 8 provides a means for pruning. The procedure below finds such a desired vertex v on $SP(s^*,t^*)$ in a convex subregion R^* of R that (possibly) is between two vertical lines (initially, $R^* = R$, $s^* = s$, and $t^* = t$).

1. Let K^* be the number of vertices of $A_{R^*} = A_R \cap R^*$. If $K^* = O(n)$, then find the desired vertex v on $SP(s^*,t^*)$ in A_{R^*} by a tree-growing approach. Otherwise, continue.
2. Compute a vertical line L as specified in Lemma 8. Let L partition the region R^* into two convex subregions R' and R''. Let S be the set of sample nodes that includes s^*, t^*, and all vertices u and w of A_{R^*} such that $\overline{uw}$ is an edge of A_{R^*} that intersects L. Note that $|S| = O(n)$ since L intersects each line of H once.
3. Run the path length algorithm on A_{R^*}, and build a clipped tree T based on the sample node set S. Associate with each node of S its crossing number.
4. Find all proper ancestors s^*, u_1, u_2, $\ldots$, u_r of t^* in T. If T contains no such nodes u_i, then $SP(s^*,t^*)$ does not touch the vertical line L and hence the search for v is reduced to the subregion (say) R' containing s^* and t^*; go to Step 6. Otherwise, go to Step 5.
5. T contains such nodes u_i, and hence $SP(s^*,t^*)$ touches L (possibly multiple times). Let $u_1, u_2, \ldots, u_r$ appear along $SP(s^*,t^*)$ in the s^*-to-t^* order. Then, either the desired vertex $v \in \{u_1, u_2, \ldots, u_r\}$, or v is an interior vertex on exactly one path $SP(u_i, u_{i+1})$, $i = 0, 1, \ldots, r$ (with $u_0 = s^*$ and $u_{r+1} = t^*$). Note that based on the definition of the sample set S, such a path $SP(u_i, u_{i+1})$ stays completely inside one of the subregions R' and R''.
6. Let R' be the subregion containing v. Search for v on $SP(u_i, u_{i+1})$ recursively in R'.

It is not hard to show that the above procedure takes $O(K+n\log n\log(K/n))$ time and $O(n)$ space, where $K=O(|A_R|)$. Making use of this procedure, we are able to report an actual path $SP(s,t)$ in $O(n\log^2 n\log(K/n)+\min\{n^2, K\log n\})$ time and $O(n)$ space (the complete details of the $SP(s,t)$ algorithm and its analysis are left to the full paper).

Remark: By using a prune-and-search procedure and a marriage-before-conquer algorithm of a similar nature as those above, we are able to find an actual longest monotone concave path in $A(H)$ on the plane in $O(n\log n\log(K/n)+\min\{n^2, K\log n\})$ time and $O(n)$ space, improving the $O(n^2\log n)$ time, $O(n\log n)$ space and $O(n^3)$ time, $O(n)$ space solutions in [9].

4 Longest Monotone Paths in an Arrangement

In this section, we illustrate our algorithmic paradigm with an algorithm for computing a longest monotone path in the planar arrangement $A(H)$. This algorithm makes use of topological sweep [9] and topological walk [2, 3] on arrangements, and of the clipped tree data structure. It takes $O(\frac{n^2\log n}{\log(h+1)})$ time and $O(nh)$ space, where h is any integer such that $1\le h\le n^\epsilon$ for any positive constant ϵ with $\epsilon<1$. This is an improvement over the $O(n^2\log n)$ time, $O(n\log n)$ space solution in [9].

A monotone path π in $A(H)$ is a continuous curve consisting of edges and vertices of $A(H)$, such that every vertical line intersects π in exactly one point. A vertex of π is a *turn* if the two incident edges are not collinear. The length of π is defined as the number of its turns plus one. The longest monotone path in $A(H)$ is denoted by $LMP(A(H))$.

Edelsbrunner and Guibas [9] used topological sweep to compute the length of $LMP(A(H))$, in $O(n^2)$ time and $O(n)$ space. To find the actual path $LMP(A(H))$, they used a recursive back-up method that maintains some "snapshots" which are states of their sweeping process. Storing each snapshot uses $O(n)$ space, which enables them to resume the sweeping process of their algorithm at the corresponding state, without having to start from the scratch. As it turns out, the algorithm for reporting $LMP(A(H))$ in [9] needs to maintain simultaneously $O(\log n)$ snapshots. Altogether, it takes $O(n^2\log n)$ time and $O(n\log n)$ space.

Our techniques are different from [9]. We use a marriage-before-conquer approach and a clipped tree whose sample node set is determined by h y-monotone curves $C_1, C_2, \ldots, C_h$, where each $C_i, i=1,\ldots,h$, is a snapshot of the y-monotone curve used in the topological sweep [9]. Those curves partition the $O(n^2)$ vertices of $A(H)$ into $h+1$ subsets of (roughly) equal sizes of $O(n^2/(h+1))$. With the clipped tree, we can identify for each C_i a vertex v_i on the path $LMP(A(H))$, such that v_i is adjacent to an edge $e(v_i)$ of $LMP(A(H))$ that intersects C_i. After the h vertices $v_1, v_2, \ldots, v_h$ are identified, the problem is reduced to $h+1$ subproblems. The i-th subproblem is to find a longest monotone subpath between v_i and v_{i+1} in the region delimited by C_i and C_{i+1} (initially, with v_0 and v_{h+1} being on the vertical lines $x=-\infty$ and $x=+\infty$, respectively). We then solve the subproblem on each such region recursively, until the region for each subproblem contains only $O(nh)$ vertices of $A(H)$ (at that point, we simply use a tree-growing approach to report the portion of $LMP(A(H))$ in that region). In the above algorithm, once $v_1, v_2, \ldots, v_h$ are identified, we associate v_i with the number of vertices of $A(H)$, denoted by num_i, between C_i and C_{i+1}. Therefore we can release the space occupied by the snapshots for $C_2, C_3, \ldots, C_h$. When the first subproblem is solved, we do a sweeping, starting from the last snapshot made in the first subproblem, in order to restore the snapshot of C_2. This is done by counting the number of $A(H)$ vertices until the num_1-th vertex is met by the sweeping. This computation continues with the other subproblems.

Note that, unlike our algorithm for the actual *shortest* path problem on $A(H)$, it is not clear to us how the sizes of the arrangement portions for the subproblems can be significantly reduced. One reason for this is that a longest monotone path in $A(H)$ can cross a line in H multiple times.

Our algorithm takes $O(\frac{n^2 \log n}{\log(h+1)})$ time and $O(nh)$ space. We leave the details of this algorithm, the correctness proof, and analysis to the full paper.

5 Dynamic Programming Problems

We briefly characterize the class of dynamic programming problems to which our general paradigm is applicable. Generally speaking, our paradigm applies to problems of the following nature:

- The problem seeks an optimal solution that consists of a value (e.g., an optimal path length) and a structure formed by a set of actual elements (e.g., an actual optimal path).
- The optimal value can be obtained by a dynamic programming approach by building a table M, such that each row of M is computed from $O(1)$ immediately preceding rows.

Let the table M have n columns and k rows (where n is the size of the input). Using our clipped tree based paradigm, we can report an actual solution by first finding an element of the actual solution at row $k/2$ (if row $k/2$ contains such an element), and then recursively solving the subproblems on the two subtables of M (one above and the other below row $k/2$).

Depending on the particular structures of the problems, one of two possibilities may occur. One possible situation is that the original problem of size n can be reduced to solving two independent subproblems of size r and size $n-r$, resulting in that a constant fraction of the entries of M is eliminated from consideration when solving these subproblems. Our algorithms for finding an actual solution for problems of this type have the same time and space bounds as those for computing the optimal value. Another possible situation is that it is not clear how to reduce the original problem of size n to two independent subproblems of sizes r and $n-r$ (i.e., each subproblem is still of size n), forcing one to use virtually the whole table M when solving the subproblems. Our algorithms for finding an actual solution for problems of this type have the same space bound as that for computing the optimal value, and a time bound with an extra $\log k$ factor.

References

1. E.G. Anagnostou, L.J. Guibas, and V.G. Polimenis, "Topological sweeping in three dimensions," *Lecture Notes in Computer Science, Vol. 450, Proc. SIGAL International Symp. on Algorithms*, Springer-Verlag, 1990, pp. 310–317.
2. T. Asano, L.J. Guibas, and T. Tokuyama, "Walking in an arrangement topologically," *Int. J. of Computational Geometry & Applications*, Vol. 4, No. 2, 1994, pp. 123–151.
3. T. Asano and T. Tokuyama, "Topological walk revisited," *Proc. 6th Canadian Conf. on Computational Geometry*, 1994, pp. 1–6.
4. P. Bose, W. Evans, D. Kirkpatrick, M. McAllister, and J. Snoeyink, "Approximating shortest paths in arrangements of lines," *Proc. 8th Canadian Conf. on Computational Geometry*, 1996, pp. 143–148.
5. C. Chan and G.H. Young, "Single-vehicle scheduling problem on a straight line with time window constraints," *Proc. 1st Annual International Conf. on Computing and Combinatorics*, 1995, pp. 617–626.
6. V. Chvatal, D.A. Klarner, and D.E. Knuth, "Selected combinatorial research problems," STAN-CS-72-292, 26, Stanford University, June 1972.
7. T. Dey, "Improved bounds for k-sets and k-th levels," *Proc. 38th Annual IEEE Symp. on Foundations of Computer Science*, 1997.
8. H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, Berlin, 1987.
9. H. Edelsbrunner and L.J. Guibas, "Topologically sweeping an arrangement," *J. of Computer and System Sciences*, Vol. 38, 1989, pp. 165–194.
10. H. Edelsbrunner and E.P. Mücke, "Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms," *ACM Trans. on Graphics*, Vol. 9, 1990, pp. 66–104.
11. H. Edelsbrunner and E. Welzl, "Constructing belts in two-dimensional arrangements with applications," *SIAM J. Comput.*, Vol. 15, 1986, pp. 271–284.
12. D. Eppstein and D. Hart, "An efficient algorithm for shortest paths in vertical and horizontal segments," *Lecture Notes in Computer Science, Vol. 1272, Proc. 5th International Workshop on Algorithms and Data Structures*, Springer-Verlag, 1997, pp. 234-247.
13. P. Erdös, L. Lovász, A. Simmons, and E.G. Straus, "Dissection graphs of planar point sets," In J.N. Srivastava *et al.* (Eds.), *A Survey of Combinatorial Theory*, North Holland, 1973, pp. 139–149.
14. D.S. Hirschberg, "A linear-space algorithm for computing maximal common subsequences," *Comm. ACM*, Vol. 18, No. 6, 1975, pp. 341–343.
15. P.N. Klein, S. Rao, M.H. Rauch, and S. Subramanian, "Faster shortest-path algorithms for planar graphs," *Proc. 26th Annual ACM Symp. Theory of Computing*, 1994, pp. 27–37.
16. L. Lovász, "On the number of halving lines," *Ann. Univ. Sci. Budapest, Eötvös, Sec. Math.*, Vol. 14, 1971, pp. 107–108.
17. S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer Implementations*, Wiley, New York, 1990.
18. H.N. Psaraftis, M.M. Solomon, T.L. Magnanti and T.-U. Kim, "Routing and scheduling on a shoreline with release times," *Management Science*, Vol. 36, No. 2, 1990, pp. 212–223.
19. R.A. Wagner and M.J. Fischer, "The string-to-string correction problem," *J. ACM*, Vol. 21, No. 1, 1974, pp. 168–173.