Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®

Journal of Algorithms 49 (2003) 13–41

Journal of Algorithms

www.elsevier.com/locate/jalgor

Finding an optimal path without growing the tree

Danny Z. Chen,[a,1] Ovidiu Daescu,[b,2] Xiaobo (Sharon) Hu,[a,3] and Jinhui Xu [c,4]

[a] *Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556, USA*
[b] *Department of Computer Science, University of Texas at Dallas, Richardson, TX 75083, USA*
[c] *Department of Computer Science and Engineering, State University of New York at Buffalo, 201 Bell Hall, Buffalo, NY 14260, USA*

Received 1 December 1998

Abstract

For problems on computing an optimal path as well as its length in a certain setting, the "standard" approach for finding an actual optimal path is by building (or "growing") a single-source optimal path tree. 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 new 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

E-mail addresses: chen@cse.nd.edu (D.Z. Chen), daescu@utdallas.edu (O. Daescu), shu@cse.nd.edu (X.S. Hu), jinhui@cse.buffalo.edu (J. Xu).

[1] The research of this author was supported in part by the National Science Foundation under Grants CCR-9623585 and CCR-9988468.

[2] The research of this author was partially done at the CSE Department, University of Notre Dame, and supported in part by a fellowship from the Center for Applied Mathematics, University of Notre Dame, Notre Dame, Indiana, USA, and by the NSF under grant CCR-9623585.

[3] The research of this author was supported in part by the National Science Foundation under Grants MIP-9701416 and CCR-9988468, and by HP Labs, Bristol, England, under an external research program grant.

[4] The research of this author was supported in part by a faculty start-up fund from the CSE Department, SUNY at Buffalo, and an IBM faculty partnership award. This work was partially done at the CSE Department, University of Notre Dame, and supported in part by the NSF under grant CCR-9623585.

0196-6774/$ – see front matter
doi:10.1016/S0196-6774(03)00080-4

known algorithms, and settle some open problems. Our techniques are likely to be applicable to other problems.

Keywords: Computational geometry; Optimal paths; Arrangements; Dynamic programming; Space-efficient algorithms

1. Introduction

For combinatorial problems on computing an optimal path as well as its length in a certain setting, the "standard" approach for finding an actual optimal path is by building (or "growing") a single-source optimal path tree. This is normally done by maintaining some *predecessor* information as the path lengths from the source are being computed (e.g., see [11]). 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* bound than that for computing single-source optimal paths. However, this need not be the case for many problems as far as the *space* complexity is concerned. 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 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 [14] 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 [14] 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 [10,20,31] (this problem can be reduced to an optimal path problem). Hirschberg [20] 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 [14] use a recursive back-up method, and the one in [20] 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 [20], 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 [5,17], 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 [22]. We present an $O(n^2)$ time, $O(n)$ space algorithm.

Computing a longest monotone convex/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 [14]. We present an $O(n^2)$ time, $O(n)$ space algorithm. Our solution is an improvement on those of [14], and settles the corresponding open problem in [14].

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 $O(n^2/\epsilon)$ time, $O(n^{1+\epsilon}/\epsilon)$ space algorithm were given in [14]. We present an $O(n^2 \log n/\log(h+1))$ time, $O(nh)$ space algorithm, where h is any integer such that $1 \leqslant h \leqslant n^\epsilon$ 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 $O(n^2/\epsilon)$ time and $O(n^{1+\epsilon})$ space (unlike [14], our space bound does not depend on the $1/\epsilon$ factor). Our solution is an improvement on those of [14], and provides an answer to the corresponding open problem in [14].

Computing a longest monotone path in the arrangement of n planes in the 3D *space*. An $O(n^3)$ time, $O(n^2)$ space algorithm was given by Anagnostou et al. [1] for computing the *length* of such a path. If the techniques in [14] 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 $O(n^3 \log n/\log(h+1))$ time, $O(n^2 h)$ space algorithm, where h is any integer such that $1 \leqslant h \leqslant n^\epsilon$ for any positive constant $\epsilon < 1$. For $h = O(1)$, we use $O(n^3 \log n)$ time and $O(n^2)$ space, improving the space complexity by an $O(\log n)$ factor.

Computing a minimum-weight, k-link path in a graph. Let $G = (V, E)$ be a weighted graph of $n = |V|$ vertices and $m = |E|$ edges with nonnegative edge weights. A minimum-weight, k-link path in G between two vertices is a path that uses at most k edges and whose total sum of edge weights is minimized. If the standard tree-growing approach is used for computing such an actual optimal path, then it would use $O(k(n+m))$ time and $O(kn)$ working space [23,25]. We present an $O(k(n+m)\log k/\log(h+1))$ time, $O(nh)$ working space algorithm, where h is any integer such that $1 \leqslant h \leqslant k^\epsilon$ 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 $O((1/\epsilon)k(n+m))$ time and

$O(nk^{\epsilon})$ working space (the constant of the working space bound does not depend on $1/\epsilon$). 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*. Given a positive integer B and n items, with the ith item having a positive integer size k_i and an arbitrary weight value w_i, the problem is to select a subset of the items such that the sum of sizes of the selected items is no bigger than B and the total weight of the selected items is maximized. This problem is NP-complete and has often been solved by dynamic programming [11,25,26,30] or by reducing the problem to computing an optimal path in a directed acyclic graph of $O(nB)$ vertices and edges [23,27]. If the standard tree-growing approach is used for computing an actual solution, then it would use $O(nB)$ time and space [23,25–27,30] (it was also shown in [26] how to use a bit representation to reduce the space bound to $O(n+nB/\log(n+B))$). 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 [4]. 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 are arbitrary [6]. Psaraftis et al. [29] 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 [6] 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 end times. We present $O(n^2)$ time, $O(n)$ space algorithms for both these cases.

Actually, several of our algorithms can be further generalized. For example, for the optimal path problems on the arrangement of n planar lines, we can confine the paths to the portion of the arrangement in a specified convex region of m vertices. Our algorithms for these cases have better time bounds that depend on n, m, and the size of the arrangement portion in the convex region.

The structure of the paper is as follows. Section 2 discusses the clipped tree data structure. Section 3 gives an overview of our general algorithmic paradigm. Section 4 recalls several approaches for sweeping arrangements that are needed by our algorithms. We then illustrate various aspects of our paradigm with examples on different optimal path problems on arrangements (Sections 5 to 7). We finally characterize a class of dynamic programming problems to which our paradigm is applicable, and solve several combinatorial optimization problems of this class (Section 8).

2. Clipped trees

A key ingredient of our general paradigm is a new data structure called *clipped trees* that we introduce in this section. Clipped trees are important to our paradigm because they

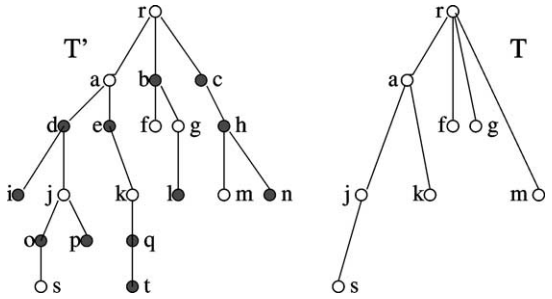


Fig. 1. T is a clipped tree of T' based on the sample nodes (unfilled circles).

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 of 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. An example of a clipped tree is given in Fig. 1.

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 from the given 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 a 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 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 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 often beneficial to associate with the nodes of T certain 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 [20] computes an actual optimal path using both the source-to-destination and destination-to-source orders. Although the problem in [20] 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., [6,29]), 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 [14], 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. Overview of the paradigm

In this section, we give a general overview of our algorithmic paradigm. Note that this paradigm, when applied to a specific problem, may be incorporated with other techniques and special structures of the problem to achieve an efficient algorithm.

The outline of our paradigm for finding an actual optimal path $OP(s,t)$ between two vertices s and t in a given setting is as follows. Suppose an algorithm for computing the *lengths* of such optimal paths from a vertex is already known.

1. Run the algorithm for computing the lengths of single-source optimal paths, starting at vertex s. Assume that this algorithm visits the vertices of the given setting in an order of growing a single-source optimal path tree *SST* rooted at s.
2. The tree *SST* is not explicitly stored. Instead, a sample set S of the nodes of *SST* (s and t are in S) is maintained by a clipped tree T, such that S contains some vertices of the sought optimal path $OP(s,t)$ in addition to s and t. (Note that the criterion for determining the sample set S is problem-specific.)
3. Identify the vertices of $S \cap OP(s,t)$ from T. Let these vertices be $v_1, v_2, \ldots, v_g$ along $OP(s,t)$, in this order, with $v_1 = s$, $v_g = t$, and $g \geqslant 3$.
4. Recursively find an optimal path from v_i to v_{i+1}, for every $i = 1, 2, \ldots, g-1$.

Clearly, the above paradigm gives rise to an algorithm of the $(g-1)$-way marriage-before-conquer nature. There are two keys to a successful application of this paradigm to solving a problem:

(1) the availability of the *length* version of the presumed single-source optimal path algorithm, and
(2) the (problem-specific) criterion and method for determining the sample set S.

For all the problems we consider, a single-source optimal path length algorithm is available. Hence, the main difficulty is on the second key which we further discuss below.

In the above paradigm, determining an appropriate sample set S ($s, t \in S$) in an efficient fashion is critical to the overall efficiency of the desired algorithm for computing an actual optimal path $OP(s,t)$. Some of the particularly useful properties of S are as follows:

1. S should not be very large (otherwise, S itself could become the optimal path tree SST).
2. S should contain some vertices of the optimal path $OP(s,t)$ in addition to s and t.
3. The vertices in $S \cap OP(s,t)$ should induce "nice" subproblems for the recursion:
   (3.1) The sizes of the subproblems should somewhat be "balanced."
   (3.2) It would be very helpful if the sum of the sizes of the $g-1$ subproblems is smaller than the size of the original problem (say, a constant fraction of the original size).

Remark. Note that in property (3.2), when subproblem sizes sum to a constant fraction of the original problem, balancing condition (3.1) becomes unnecessary.

If the sample set S is "nice" (i.e., having all the above properties), then it is possible to compute an optimal path $OP(s,t)$ in the same time bound as that for computing the lengths of optimal paths from vertex s. In fact, for the majority of the problems we consider, the sample sets we use have these properties. However, there are still a few problems for which we cannot find sample sets satisfying property (3.2). As a consequence, our algorithms for computing $OP(s,t)$ for such problems have an additional logarithmic factor in the time bound in comparison with the corresponding length versions of these algorithms. Thus, depending on whether the sample sets we use satisfy property (3.2), we classify the problems into two types: type A satisfying property (3.2) and type B not satisfying property (3.2).

One of our main efforts is therefore spent on constructing nice sample sets, and it is usually for determining a nice sample set that other techniques and specific problem structures are brought into the picture. Sections 5 to 8 present examples showing various ways of obtaining a nice sample set S for different optimal path problems.

4. Topological sweep, topological walk, and topological peeling

Arrangements are a fundamental structure in combinatorial and computational geometry [13], and a great deal of work has been devoted to studying various arrangements and

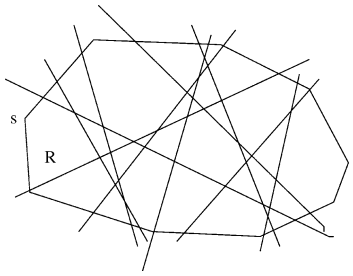


Fig. 2. The arrangement H in a convex region R.

their properties. We briefly discuss in this section several arrangement sweeping algorithms that will be needed in Section 5 to Section 7: Topological sweep [1,14], topological walk [2,3], and topological peeling [8,9].

Let $H = \{l_1, l_2, \ldots, l_n\}$ be a set of n straight lines on a plane. The lines in H divide the plane into a subdivision called the *arrangement* $A(H)$ of H. $A(H)$ consists of a set of convex regions (called *cells*), each bounded by some edges (i.e., segments of the lines in H) and vertices (i.e., intersection points between the lines in H). The interior of each cell of $A(H)$ intersects no lines of H. In general, $A(H)$ has $O(n^2)$ cells, edges, and vertices. One may also consider the portion A_R of $A(H)$ within a convex polygonal region R, i.e., $A_R = A(H) \cap R$ (see Fig. 2).

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 and no line is vertical (the general case can be handled by using the techniques in [15]).

If one is interested only in constructing and reporting (but not storing) $A(H)$, then this can be done by a relatively simple algorithm that sweeps the plane by a vertical line, in $O(n^2 \log n)$ time and $O(n)$ space [16]. Edelsbrunner and Guibas [14] gave 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 et al. [2] developed another interesting 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 by preferring left branches [2,3]. Topological walk can also be extended to traversing the portion A_R of $A(H)$ inside a convex polygonal region R, in $O(K + (n + |R|)\log(n + |R|))$ time and $O(n + |R|)$ space, where K is the number of vertices in A_R and $|R|$ is the number of vertices of R.

Let $P = \{p_1, p_2, \ldots, p_n\}$ be a set of n planes in the 3D space. The arrangement $A(P)$ in the 3D space is the subdivision of convex polytopes dissected by the n planes in P. Each such convex polytope is bounded by a set of 2D faces, each of which is a cell of the arrangement on a plane $p_i \in P$ defined by the $n-1$ lines that are the intersections of p_i with all $p_j \in P - \{p_i\}$. In general, there are $O(n^2)$ such lines on each plane of P. Hence $A(P)$ has $O(n^3)$ vertices. Anagnostou, Guibas, and Polimenis [1] extended the planar topological sweep technique in [14] to 3D, sweeping the n 2D arrangements (one on each plane p_i) simultaneously with a special unbounded monotone surface. This sweeping algorithm takes $O(n^3)$ time and $O(n^2)$ space [1].

However, there are problems on arrangements to which topological sweep and topological walk may not be appropriate. Computing shortest paths in $A(H)$ is such a problem. It seems to be difficult for topological walk to compute shortest paths in $A(H)$ because of the depth-first nature of its searching strategy. Topological sweep may also not work well, since a shortest path can cross its y-monotone sweeping curve multiple times.

A new arrangement sweeping approach, called *topological peeling*, was introduced by Chen and Xu [8,9]. Starting at a vertex s of $A(H)$ in a convex region R on a plane, this approach expands the sweeping of A_R in a manner as if a wave is propagated on A_R from s. Let $B(R)$ be the boundary of R and P be a simple curve on A_R such that P starts and ends on $B(R)$. We say P is a *convex curve* to s if the planar region containing s and enclosed by P and $B(R)$ is convex. A topological peeling advances its traversal of A_R by "propagating" from one convex curve to the next convex curve on A_R. It traverses A_R in $O(K + (n + |R|)\log(n + |R|))$ time and $O(n + |R|)$ space. Due to its "wave-propagation" nature, topological peeling is suitable for several problems such as computing shortest paths in arrangements of lines on a plane [8,9].

5. Shortest paths in an arrangement

In this section, we illustrate our paradigm with an algorithm for finding an actual shortest path between two points in the arrangement of lines on the plane. The problem can be stated as follows: Given a set H of n lines and two points s and t on some lines of H on the plane, find an s-to-t path of the shortest Euclidean distance that is restricted to lie on the lines of H. As mentioned in [5,17,18], to solve this geometric shortest path problem, one can first construct a planar graph of size $O(n^2)$ that represents the arrangement $A(H)$ of H and then apply the optimal algorithm for computing a shortest path in a planar graph [22]. 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. By using the topological peeling approach, Chen and Xu [8,9] was able to come up with an $O(n \log n + K)$ time, $O(n)$ space algorithm for computing the lengths of single-source shortest paths in A_R. Here, K is the number of vertices of A_R for a special convex polygonal region R that contains a shortest s-to-t path in $A(H)$ ($K = O(n^2)$ in the worst case).

Although we are not yet able to improve the asymptotic time bound of the previously known actual shortest path algorithm [5,17], we show how to reduce its space bound by a factor of n. Our algorithm finds an actual shortest s-to-t path in $O(n)$ space and $O(n \log^2 n \log(K/n) + \min\{n^2, K \log n\})$ time. Hence our algorithm in the worst case takes $O(n)$ space and $O(n^2)$ time. Our solution is based on topological peeling [8,9], as well as topological sweep [14] and topological walk [2,3]. It also makes use of additional techniques such as marriage-before-conquer, prune-and-search, and the clipped tree data structure, and exploits a number of interesting observations on this particular problem.

5.1. 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. Of course, $\overline{st}$ need not be on any line of H. 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 [5], 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 A_R of $A(H)$ contained in R (by computing the common intersection of the half-planes in $HP(H - H_c(\overline{st}))$ [28] and identifying the crossing lines of $\overline{st}$). Henceforth, we still let n denote the number of lines of H intersecting the convex region R.

A topological peeling, starting at s, is used in [8,9] to report the length of a shortest s-to-t path in A_R. The following is a simple yet useful lemma.

Lemma 1. *For any line $l \in H$ and any vertex v of $A(H)$ on l, the intersection of a shortest s-to-v path with l is either the vertex v or a line segment on l that has v as an endpoint. Consequently, 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) and l cannot contribute two disjoint line segments to a shortest s-to-v path in $A(H)$.*

Proof. Obviously, the vertex v is on the shortest s-to-v path. Assume that the shortest s-to-v path crosses l and let u be a vertex at which the path crosses l. Then, by replacing the portion of the path between u and v by the line segment $\overline{uv}$ on l, a shorter path is obtained, a contradiction. Similar arguments hold when assuming that l contributes two disjoint line segments, or one line segment that does not have v as an endpoint, to the shortest s-to-v path. □

We consider a generalization of Lemma 1. Recall that a simple curve P on A_R with both its endpoints on the boundary $B(R)$ of R is convex to s if the planar region containing s and enclosed by P and $B(R)$ is convex. The following lemma is important to the algorithm for computing single-source shortest path lengths in A_R.

Lemma 2. *Let P be a convex curve to s on A_R whose endpoints are both on $B(R)$, and v be a vertex of A_R inside the region enclosed by P and $B(R)$. Then no shortest s-to-v path in A_R can cross P.*

Proof. This follows easily from the convexity of P. □

Based on Lemma 2 and by incorporating the computation of shortest path lengths with the construction and traversal of A_R by topological peeling, an efficient algorithm for computing the lengths of single-source shortest paths in A_R from the source s was given in [8,9]. In particular, this algorithm computes the shortest path lengths in the parent-to-children order in the single-source shortest path tree rooted at s. The following result has been given in [8,9].

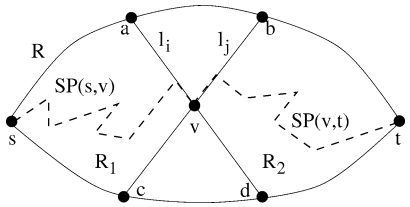


Fig. 3. The two regions R_1 and R_2 in R defined by two lines l_i and l_j of H.

Lemma 3. *The length of a 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.*

5.2. Computing an actual shortest path

In this subsection, we present the $O(n \log^2 n \log(K/n) + \min\{n^2, K \log n\})$ 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.

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 marriage-before-conquer based algorithm.

Lemma 4. *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 (see Fig. 3). 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.*

Proof. The first part of the lemma follows from Lemma 1. For the second part, observe that R_1 and R_2 share l_i and l_j. Let a and b (respectively, c and d) be the intersection points of l_i and l_j with the upper (respectively, lower) boundary of R. Let $l(\overline{ab})$ be the line (possible not in H) containing $\overline{ab}$. If $l(\overline{ab}) \in H$ (respectively, $l(\overline{cd}) \in H$), then $l(\overline{ab})$ (respectively, $l(\overline{cd})$) is shared by R_1 and R_2. Since R is a convex region, no other upper (respectively, lower) boundary line of H can be shared by R_1 and R_2. □

Lemma 5. *The lines in H that $SP(s,t)$ crosses are exactly the crossing lines of $\overline{st}$ (i.e., $H_c(\overline{st})$).*

Proof. Simple and omitted. □

Lemma 6. *Let l_i and l_j be defined as in Lemma 4. 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 (respectively, H_2) intersects $SP(v,t)$ (respectively, $SP(s,v)$). Moreover, H_1 (respectively, H_2) consists of all the lines in $H_c(\overline{st}) - \{l_i, l_j\}$ that intersect the interior of the line segment $\overline{sv}$ (respectively, $\overline{vt}$), i.e., $H_1 = H_c(\overline{sv})$ (respectively, $H_2 = H_c(\overline{vt})$).*

Proof. Assume that there is a line $l \in H_c(\overline{st}) - \{l_i, l_j\}$ such that l intersects the interior of both $SP(s,v)$ and $SP(v,t)$. Since $SP(s,t) = SP(s,v) \cup SP(v,t)$, it implies that either l contributes to $SP(s,t)$ with at least two disjoint line segments, which contradicts Lemma 1, or l contributes to $SP(s,t)$ with exactly one line segment, which means that l contains v and hence $l \in \{l_i, l_j\}$, also a contradiction. To prove the second part of the lemma, let $l \in H_c(\overline{st}) - \{l_i, l_j\}$ and observe that l cannot intersect both $\overline{sv}$ and $\overline{vt}$ (otherwise, s and t would be on the same side of l, and thus $l \notin H_c(\overline{st})$). Let $P = \overline{sv} \cup \overline{vt}$. Since P is a continuous curve inside R, l must cross P. Hence l crosses either $\overline{sv}$ or $\overline{vt}$ (but not both). Assume l crosses $\overline{sv}$. By Lemma 5, $SP(s,v)$ crosses l and thus l cannot intersect $SP(v,t)$. □

Lemma 6 implies that if we are to compute $SP(s,v)$ (respectively, $SP(v,t)$), the lines in $H_c(\overline{vt})$ (respectively, $H_c(\overline{sv})$) 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 4). 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 in H.

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

Lemma 7. *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 line $l(u)$ (respectively, $l(w)$) of H intersect l at u (respectively, w). Then $H_c(\overline{su})$ differs from $H_c(\overline{sw})$ on at most two elements. Furthermore, these different elements are in $\{l(u), l(w)\}$.*

Proof. Since u and w are neighboring vertices of $A(H)$, it is easy to see that $H_c(\overline{su}) - \{l_u, l_w\} = H_c(\overline{sw}) - \{l_u, l_w\}$. The possible difference between $H_c(\overline{su})$ and $H_c(\overline{sw})$ can only be caused by l_u and l_w, and there are four possible cases:

(1) $l_u \in H_c(\overline{sw})$ and $l_w \in H_c(\overline{su})$;
(2) $l_u \in H_c(\overline{sw})$ and $l_w \notin H_c(\overline{su})$;
(3) $l_u \notin H_c(\overline{sw})$ and $l_w \in H_c(\overline{su})$;
(4) $l_u \notin H_c(\overline{sw})$ and $l_w \notin H_c(\overline{su})$.

To identify which case holds, one only needs to check the intersections of l_u, l_w with $\overline{sw}, \overline{su}$, which can be easily done in constant time. □

Based on Lemma 7, 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 peeling 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 ($k = \lceil t_c/2 \rceil$ can be obtained by running the shortest path length algorithm on A_R once, as a preprocessing step). The following is done.

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 $\overline{st}$ 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 (by Lemma 4).

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 kth level of the arrangement of n planar lines. The best known lower bound for the kth level size of such an arrangement is $O(n\log(k+1))$ [19,24], and the best known upper bound is $O(nk^{1/3})$ [12]. 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 8. *For any two vertices u and w of $A(H)$ such that u is on $SP(s,w)$, $w_c \geqslant u_c$.*

Proof. An immediate consequence of Lemma 1. □

Lemma 9. *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 $c_1 + c_2 = 1$.*

Proof. Use a topological walk to compute a vertex u of A_R such that there are at least $O(K/4)$ vertices of A_R on each side of the vertical line L passing through u. This can be done as follows. WLOG, assume K is super-linear.

1. Find the median of the x-coordinates of the first K/n vertices of A_R encountered by the topological walk, then find the median of the next K/n vertices, and so on, until all the vertices of A_R are encountered.
2. Find the median x_u of the n medians computed in Step 1. Let L pass through x_u.

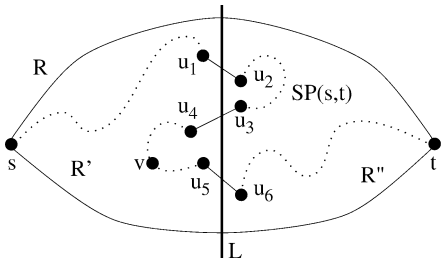


Fig. 4. Searching for a desired vertex v on $SP(s,t)$ by prune-and-search.

Note that the algorithm above finds a median among the medians of n lists. The fact that there are at least $O(K/4)$ vertices of A_R on each side of the vertical line L follows from the standard analysis for the linear time selection algorithm [11]. Since the median of a list can be computed in linear time and space, u is found by a topological walk in $O(n \log n + K)$ time and $O(n)$ space. □

Lemma 8 provides a structure on $SP(s,t)$ for searching, and Lemma 9 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 9. Let L partition the region R^* into two convex subregions R' and R'' (Fig. 4). 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^*} from s^* to t^*, 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 both 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'' (Fig. 4).
6. Let R' be the subregion containing v. Search for v on $SP(u_i, u_{i+1})$ recursively in R'.

It is easy to see that the procedure above takes $O(n)$ space. Its time complexity (dominated by the computation in the second and third steps) is given by the following recurrence:

$$T(K) \leqslant T\left(\frac{3K}{4}\right) + O(K + n\log n), \quad \text{if } K > n,$$
$$T(K) = O(n), \quad \text{if } K \leqslant n,$$

whose solution is $T(K) = O(K + n\log n\log(K/n))$. Therefore, the above procedure takes $O(K + n\log n\log(K/n))$ time and $O(n)$ space, where $K = |A_R|$.

With 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. Our algorithm for reporting $SP(s,t)$ proceeds as follows (initially, $R^* = R$, $s^* = s$, $t^* = t$, $t_c^* = t_c$, and $t_c = O(n)$):

1. Let R^* be a convex subregion of R in which an actual path $SP(s^*,t^*)$, for two vertices s^* and t^* on $SP(s,t)$, is to be reported. Let K^* be the size of $A_{R^*} = A_R \cap R^*$. If $K^* = O(n)$, then report $SP(s^*,t^*)$ in A_{R^*} by a tree-growing approach. Otherwise, continue.
2. Find a vertex v on $SP(s^*,t^*)$ such that $v_c = t_c^*/2$ and v is on a crossing line of $\overline{s^*t^*}$.
3. Compute the line sets H_1 and H_2, each of size v_c, that are respectively associated with the convex regions R_1 and R_2 defined by the lines intersecting at v (Lemmas 4 and 6).
4. Recursively report $SP(s^*,v)$ and $SP(v,t^*)$ on R_1 and R_2, respectively.

Let $T(R)$ be the recursion tree of the algorithm above, whose height is $O(\log n)$. Then each node u of $T(R)$ is associated with a convex subregion R_u. Observe that our algorithm need not store the whole tree $T(R)$. Instead, at any given moment, it only needs to store a leaf-to-root path in $T(R)$ (as well as the associated information of the $O(\log n)$ nodes on that path). For a node u on such a path in $T(R)$, the algorithm mainly maintains two vertices between which a shortest path is to be reported, and the crossing lines associated with R_u. Since the number of crossing lines at each node is a constant fraction of that at its parent node, the information of each leaf-to-root path in $T(R)$ uses only $O(n)$ space to store. Hence the overall algorithm also uses $O(n)$ space.

The time complexity of the algorithm is given by the following recurrence:

$$T(m,K) \leqslant T\left(\frac{m}{2}, K_1\right) + T\left(\frac{m}{2}, K_2\right) + O\big(K + m\log m\log(K/n)\big), \quad \text{if } K > n,$$
$$T(m,K) = O(n), \quad \text{if } K \leqslant n,$$

where $K \leqslant m^2$, $K_1 \leqslant (m/2)^2$, $K_2 \leqslant (m/2)^2$, and $K_1 + K_2 \leqslant K$. The solution of this recurrence is $T(n,K) = O(n\log^2 n\log(K/n) + \min\{n^2, K\log n\})$. Therefore, the algorithm takes altogether $O(n\log^2 n\log(K/n) + \min\{n^2, K\log n\})$ time and $O(n)$ space, where $K = |A_R|$.

Theorem 1. *A shortest s-to-t path in the arrangement $A(H)$ of a set H of n planar lines can be reported in $O(n\log^2 n\log(K/n) + \min\{n^2, K\log n\})$ time and $O(n)$ space, where K is the number of vertices of A_R and R is the convex region associated with s and t on the plane.*

Proof. It follows from the above discussion and analysis. □

Note that it is easy to extend our algorithm to computing a shortest s-to-t path confined to the portion of $A(H)$ in any given convex region R' on the plane, with an additive time factor $|R'|\log|R'|$ and an additive space factor $|R'|$.

6. Longest monotone concave/convex paths in an arrangement

In this section, we present an $O(n^2)$ time, $O(n)$ space algorithm for reporting a longest monotone concave (or convex) path in $A(H)$, improving the time and space bounds of the algorithms by Edelsbrunner and Guibas [14]. This algorithm in spirit is similar to the one for reporting a shortest path in Section 5, yet is less sophisticated than the shortest path algorithm. Hence we will mainly describe the differences between these two algorithms. WLOG, assume that we are to compute a longest monotone concave path.

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. A monotone path π is *concave* (respectively, *convex*) if the curve keeps making left (respectively, right) turns as one traces it from left to right. The length of π is defined as the number of its turns plus one.

Let *LMC* be a longest monotone concave path in $A(H)$ and $LMC(u, v)$ be the subpath of *LMC* between two vertices u and v of *LMC*. WLOG, we assume that the vertices along *LMC* are in increasing order of their x-coordinates. Let $slope(l)$ denote the slope of a line (or line segment) l, and $slope_m(H')$ denote the median slope of the lines in a subset H' of H. Clearly, the slopes of the segments along *LMC* are in increasing order. For a line segment e, the supporting line $l(e)$ of e is the line containing e. The next lemma is useful.

Lemma 10. *Let s and t be two vertices of LMC and e_s and e_t be the first and last segments of $LMC(s, t)$, respectively. Let w be the intersection of the two supporting lines $l(e_s)$ and $l(e_t)$. Then each line in H that contributes a segment to $LMC(s, t)$ intersects both the segments $\overline{sw}$ and $\overline{wt}$.*

Proof. It follows from the fact that the slopes of the segments along the monotone concave path $LMC(s, t)$ are in increasing order (see Fig. 5). □

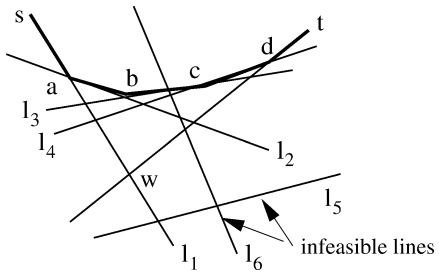


Fig. 5. Illustrating Lemma 10.

If a line $l \in H$ intersects both $\overline{sw}$ and $\overline{wt}$, then we say that, to $LMC(s,t)$, l is *feasible*, otherwise *infeasible*. Examples are given in Fig. 5, where l_1, l_2, l_3, l_4 are feasible and l_5, l_6 are infeasible.

Our algorithm for reporting *LMC* proceeds as follows:

1. Find two consecutive segments $\overline{uz}$ and $\overline{zw}$ on *LMC* such that $slope_m(H)$ is between $slope(\overline{uz})$ and $slope(\overline{zw})$ (i.e., $slope(\overline{uz}) \leqslant slope_m(H) \leqslant slope(\overline{zw})$).
2. Partition H into two subsets H_1 and H_2, each having at most $n/2$ lines, such that all lines in $H_1 \cup H_2$ are feasible, and such that the slopes of all lines in H_1 are less than $slope(\overline{uz})$ and the slopes of all lines in H_2 are greater than $slope(\overline{zw})$.
3. Recursively report the portion of *LMC* before $\overline{uz}$ and the portion of *LMC* after $\overline{zw}$.

Clearly, the key to the algorithm above is Step 1. For this step, we use the following prune-and-search procedure to locate the consecutive segments $\overline{uz}$ and $\overline{zw}$ of *LMC*.

1. Let R be a convex region in which the subpath $LMC(s,t)$ of *LMC*, for two vertices s and t of *LMC* in A_R, contains segments $\overline{uz}$ and $\overline{zw}$. (Initially, R is the whole plane, and s and t are the "vertices" of *LMC* at $-\infty$ and $+\infty$.) Let $K = |A_R|$. If $K = O(n)$, then find $LMC(s,t)$ in A_R (as well as segments $\overline{uz}$ and $\overline{zw}$) by a tree-growing approach. Otherwise, continue.
2. Compute a vertical line L as specified in Lemma 9. Observe that any monotone concave path can intersect L at most once. Let the sample node set S_L contain s, t and the right vertices of the edges e of all longest monotone concave paths in A_R from s such that L intersects e.
3. Build a clipped tree T based on S_L by running the length version of the longest monotone concave path algorithm in [2]. This algorithm computes the length of $LMC(s,t)$ in A_R. Let e_v be the edge of $LMC(s,t)$ whose right vertex v is a node of T.
4. Let L partition R into two convex subregions R_1 and R_2, each having $K/2$ vertices of R. By comparing $slope(e_v)$ and $slope_m(H)$, decide on which side of L the desired vertex z of *LMC* lies. Let say $slope_m(H) < slope(e_v)$ (and hence $z \in R_1$).
5. Recursively search for $\overline{uz}$ and $\overline{zw}$ (on path $LMC(s,v)$) in the convex region R_1', where R_1' is the common intersection of R_1 and the upper half-plane bounded by $l(e_v)$.

The procedure for Step 1 takes $O(n^2)$ time and $O(n)$ space, since the problem size in each recursive call is reduced by a constant fraction.

Hence, the time bound of the overall *LMC* algorithm is given by the recurrence $T(n) \leqslant 2T(n/2) + O(n^2)$, whose solution is $T(n) = O(n^2)$. The space bound is only $O(n)$, since each recursive call uses $O(n)$ space for bookkeeping and for the topological walk.

Theorem 2. *A longest monotone concave* (*or convex*) *path in the arrangement of n lines on the plane can be reported in $O(n^2)$ time and $O(n)$ space.*

Proof. It follows from the above discussion. □

Our algorithm can also be used to solve, in the same time and space bounds, the dual problem, which is stated as follows [14]: Given a set S of n planar points (S is dual to H so that the slopes of the lines determine the x-coordinates of the points), find a longest concave/convex chain of S.

7. Longest monotone paths in an arrangement

In this section, we apply our algorithmic paradigm to the problems of computing a longest monotone path in the arrangement of n lines in 2D or n planes in 3D, yielding two efficient algorithms. These algorithms make use of topological sweep [1,14] and our clipped tree data structure. Our algorithm for the planar case takes $O(n^2 \log n/\log(h+1))$ time and $O(nh)$ space, with $1 \leqslant h \leqslant n^\epsilon$ and $0 < \epsilon < 1$. For $h = O(1)$, our algorithm uses $O(n^2 \log n)$ time and $O(n)$ space, improving the $O(n^2 \log n)$ time, $O(n \log n)$ space solution in [14]. For $h = n^\epsilon$, our algorithm uses $O(n^2/\epsilon)$ time and $O(n^{1+\epsilon})$ space (unlike the $O(n^2/\epsilon)$ time, $O(n^{1+\epsilon}/\epsilon)$ space solution in [14], our space bound does not depend on the $1/\epsilon$ factor). Our algorithm for the 3D case takes $O(n^3 \log n/\log(h+1))$ time and $O(n^2 h)$ space, improving the $O(n^3 \log n)$ time, $O(n^2 \log n)$ space solution that would result if the techniques in [14] are applied.

7.1. Longest monotone path in a 2D arrangement

Let $A(H)$ be the arrangement formed by a set H of n lines on the plane. We denote by $LMP(A(H))$ a longest monotone path in $A(H)$. Monotone paths in $A(H)$ and their lengths are defined in the same manner as in Section 6.

Edelsbrunner and Guibas [14] used topological sweep to compute the length of $LMP(A(H))$, in $O(n^2)$ time and $O(n)$ space. To find an 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 initial state again. As it turns out, the algorithm for reporting $LMP(A(H))$ in [14] 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 [14]. We use a marriage-before-conquer approach and a clipped tree. Our algorithm first performs a topological sweep on $A(H)$ and makes h snapshots $S_1, S_2, \ldots, S_h$ of the sweeping process. Each snapshot S_i determines a cut C_i (i.e., a sequence of n special edges of $A(H)$) and a corresponding y-monotone sweeping curve SC_i, as in [14]. The h sweeping curves partition the $O(n^2)$ vertices of $A(H)$ into $h+1$ subsets of (roughly) equal sizes of $O(n^2/(h+1))$. Using the $O(nh)$ right vertices of the edges of the h cuts $C_1, C_2, \ldots, C_h$ as the sample nodes for the clipped tree, we can identify for each C_i the vertex v_i on $LMP(A(H)) \cap C_i$. After the h vertices $v_1, v_2, \ldots, v_h$ of $LMP(A(H))$ are identified, the problem is reduced to solving $h+1$ subproblems of equal sizes, in the left to right order.

The ith subproblem is to find a longest monotone subpath of $LMP(A(H))$ between v_i and v_{i+1} in the region bounded by the sweeping curves SC_i and SC_{i+1} (initially, let

v_0 and v_{h+1} be on the vertical lines $x = -\infty$ and $x = +\infty$, respectively). We solve the subproblem on each such region recursively, until the region for the 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 this algorithm, once $v_1, v_2, \ldots, v_h$ are identified, we associate with each v_i the number of vertices of $A(H)$, denoted by num_i, to the left of the sweeping curve SC_i, and release the space occupied by snapshots $S_1, S_2, \ldots, S_h$. Starting at the initial state (or snapshot) S_0 of the sweeping algorithm, we recursively solve the first subproblem. In each step of the recursion, we always maintain exactly one snapshot S_c for the left boundary of the currently considered subproblem. It is important to observe that for any such snapshot S_c, the subpath of $LMP(A(H))$ to the left of the sweeping curve SC_c has been reported. Once the ith subproblem is solved, we resume the sweeping from S_c, which is the snapshot for the left boundary of the just solved subsubproblem, to restore snapshot S_{i+1}, and let S_{i+1} be the new S_c. Restoring S_{i+1} from S_c is done by counting the number of $A(H)$ vertices visited by the sweeping until the num_{i+1}th vertex is met.

Note that, unlike our algorithm for the actual *shortest* path problem on $A(H)$, it is not clear to us how the total size 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.

We summarize our algorithm as follow. Let R be the region bounded by two sweeping curves SC_l and SC_r, and let K be the number of vertices of $A(H)$ in R. The algorithm computes in R the subpath $LMP_R(A(H))$ of $LMP(A(H))$ from one vertex s on C_l to another vertex t on C_r, where C_l and C_r are the corresponding cuts of SC_l and SC_r. Initially, the region R is bounded by two vertical lines at $-\infty$ and $+\infty$. The following steps are carried out.

1. If R contains only $O(nh)$ vertices of $A(H)$, then perform a topological sweep in R and use the tree-growing approach to report the subpath $LMP_R(A(H))$ from s to t. Otherwise, continue.
2. Starting at the snapshot S_c (initially, S_c is the initial state of the sweeping algorithm), restore the snapshot S_l, let S_l be the new S_c, and sweep region R from S_l. During the sweeping, make a snapshot S_i at every $(K/(h+1))$th vertex encountered, and build a clipped tree T using the right vertices of the edges of the cuts $C_i, i = 1, 2, \ldots, h$, as the sample nodes.
3. Find the h ancestors $v_1, v_2, \ldots, v_h$ of t in the clipped tree T with v_i on C_i, determine for v_i the number num_i of $A(H)$ vertices to the left of SC_i, and release the space occupied by $S_1, S_2, \ldots, S_h$ and the clipped tree T.
4. Recursively report the subpath of $LMP_R(A(H))$ from s to v_1, from v_1 to $v_2, \ldots$, and from v_h to t, in this order.

The correctness of this algorithm follows from the correctness of the solution in [14]. Its time bound is given by the following recurrence:

$$T(K) = (h+1)T\left(\frac{K}{h+1}\right) + O(K), \quad \text{if } K > nh,$$

$$T(K) = O(nh), \quad \text{if } K \leqslant nh,$$

whose solution is $T(K) = O(K \log K / \log(h+1))$, with K being the size of the portion of $A(H)$ in a region R. For $A(H)$, $K = O(n^2)$, and hence the time bound for computing $LMP(A(H))$ is $O(n^2 \log n / \log(h+1))$.

For the space bound, note that the topological sweep uses $O(n)$ space. The h snapshots and the clipped tree both use $O(nh)$ space. After determining the vertices in $(\bigcup_{i=1}^{h} C_i) \cap LMP(A(H))$, we release the space occupied by the h snapshots and the clipped tree. Thus, for each subproblem (except the currently considered one), we only need to maintain $O(1)$ information. Hence the total space used in any step is $O(nh)$.

The following result follows from the above discussion.

Theorem 3. *An actual longest monotone path $LMP(A(H))$ in a* 2D *arrangement $A(H)$ of n lines can be computed in $O(n^2 \log n / h + 1)$ time and $O(nh)$ space, where h is an integer between* 1 *and n^ϵ, for any constant ϵ with $0 < \epsilon < 1$.*

7.2. Longest monotone path in a 3D arrangement

In 3D, a monotone path π in the arrangement $A(H)$ of n planes in H is a connected curve of edges and vertices of $A(H)$ such that any plane perpendicular to the x-axis cuts π at exactly one point.

To find a longest monotone path in $A(H)$ in 3D, we use an algorithm similar to the 2D version. The main differences are as follows.

First of all, instead of using the 2D topological sweep algorithm [14], we use the generalized version, the 3D topological sweep [1], to sweep $A(H)$. Second, for each region R bounded by two sweeping surfaces, if the number of vertices in R is $O(n^2 h)$, then a tree-growing approach is used to report the subpath $LMP_R(A(H))$ of $LMP(A(H))$ in R. Otherwise, we perform a 3D topological sweep in R to make h snapshots $S_1, S_2, \ldots, S_h$ and build a clipped tree T from the $O(n^2 h)$ right vertices of the edges of the cuts C_i, $i = 1, 2, \ldots, h$. After determining the h vertices of $(\bigcup_{i=1}^{h} C_i) \cap LMP(A(H))$, the space for the clipped tree and $S_1, S_2, \ldots, S_h$ is released, and the algorithm orderly reports the subpaths of $LMP_R(A(H))$ in the equal-size subregions of R. Third, we make use of the length version of the 3D longest monotone path algorithm [1].

The correctness of our 3D algorithm can be argued as for the 2D version. The time bound is given by the following recurrence:

$$T(K) = (h+1)T\left(\frac{K}{h+1}\right) + O(K), \quad \text{if } K > n^2 h,$$
$$T(K) = O\left(n^2 h\right), \quad \text{if } K \leqslant n^2 h,$$

whose solution is $T(K) = O(K \log K / \log(h+1))$. For $K = O(n^3)$, the time bound is $O(n^3 \log n / \log(h+1))$. Clearly, the space bound is $O(n^2 h)$. Thus, we have the following result.

Theorem 4. *A longest monotone path in a* 3D *arrangement of n planes can be reported in $O(n^3 \log n / \log(h+1))$ time and $O(n^2 h)$ space, where h is an integer between* 1 *and n^ϵ,*

for any constant ϵ with $0 < \epsilon < 1$. In particular, if $h = O(1)$, then the time and space bounds are $O(n^3 \log n)$ and $O(n^2)$, respectively.

8. Dynamic programming problems

Our paradigm is also applicable to a number of other problems, such as computing a minimum-weight, k-link path in a graph [23,25], 0–1 knapsack with integer item sizes [11, 23,25–27,30], and single-vehicle scheduling for sites on a straight line with special time window constraints [6,29]. We first discuss the common properties of the class of such problems, and then show how our paradigm is applied to several such problems.

8.1. General characterization

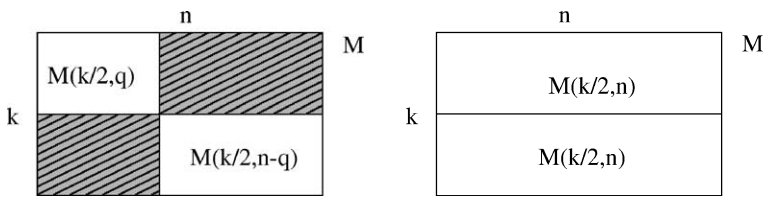
Generally speaking, our paradigm applies to problems with the following two properties:

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

Let the value table M have n columns and k rows, where n is the size of the input and k is an input integer value that may or may not be related to n. Using our clipped-tree based paradigm, we can report an actual optimal structure by first finding an element of that structure 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 $k/2$.

As mentioned in Section 3, depending on its particular properties, a problem solvable by our paradigm may fall into one of two categories: type A and type B. In fact, the path problems in Sections 5 and 6 are of type A, and those in Section 7 are of type B. We now further discuss these two types of problems in the framework of dynamic programming algorithms. Figure 6 illustrates these two types of problems.

For a type A problem, the original problem of size n can be reduced to solving two independent subproblems of sizes q and $n - q$, resulting in that a constant fraction of the



(a) Reduction for type A (b) Reduction for type B

Fig. 6. Reductions for the two different types of dynamic programming problems.

entries of M is eliminated from consideration when solving these subproblems. Hence our algorithms for finding an actual optimal structure for type A problems have the same time and space bounds as those for computing an optimal value. Examples of such problems include computing a minimum-weight, k-link path in a directed acyclic graph [23,25], 0–1 knapsack with integer item sizes [11,23,25–27,30], and single-vehicle scheduling for sites on a straight line with special time window constraints [6,29].

However, there exist some problems (type B) for which it is not clear how to reduce the original problem of size n to two independent subproblems of sizes q and $n-q$ (i.e., we only know how to reduce it to two subproblems of size n each). Thus for a type B problem, one needs to involve virtually the whole table M when solving the subproblems. Our algorithms for finding an actual optimal structure for type B problems have the same space bound as that for computing an optimal value, and a time bound with an extra $\log k$ factor. An example of such problems is computing a minimum-weight, k-link path in a general graph of n vertices and m edges [23,25]. Actually, by sampling h rows of the table M, we obtain an $O(k(n+m)\log k/\log(h+1))$ time, $O(nh)$ working space algorithm for the general minimum-weight, k-link path problem, where h is any integer such that $1 \leqslant h \leqslant k^{\epsilon}$ for any constant ϵ with $0 < \epsilon < 1$.

A key to determining whether a problem is of type A or type B is the dependencies among the entries in the corresponding dynamic programming table. In the following subsections, we apply our general paradigm to solving several problems in the framework of dynamic programming.

8.2. 0–1 knapsack problem

The 0–1 knapsack problem with integer item sizes is a well studied special case of the knapsack problem [23]. Given a positive integer B (for the knapsack size) and n items, with the ith item having a positive integer size k_i and an arbitrary weight value w_i, the problem is to select a subset of items such that the sum of the sizes of the selected items is no bigger than B and such that the total weight of the selected items is maximized. This problem is NP-complete and has often been solved by dynamic programming [11, 25,26,30] or by reducing the problem to computing an optimal path in a directed acyclic graph of $O(nB)$ vertices and edges [23,27]. When the standard tree-growing approach is used for computing an actual solution, it takes $O(nB)$ time and space [23,25–27,30]. It was also shown in [26] how to use a bit representation to reduce the space bound to $O(n + nB/\log(n+B))$. We present an $O(nB)$ time, $O(n+B)$ space algorithm.

Let $W_{i,j}$ be the optimal weight value of the knapsack problem with a knapsack size j and using as the candidates of selection the first i items. Then $W_{i,j}$ can be computed as follows [23]:

$$W_{i,j} = \max\{W_{i-1,j}, W_{i-1,j-k_i} + w_i\}.$$

A dynamic programming table M of size $n \times B$ for the weight values $W_{i,j}$ is depicted in Fig. 7 (where the dashed lines indicate the partitions of the subproblems). The rows of M correspond to the n given items, and the columns correspond to knapsack sizes of $0, 1, 2, \ldots, B$. The two arrows into entry $M(i, j)$ indicate the dependency of $W_{i,j}$ on $W_{i-1,j}$ and $W_{i-1,j-k_i}$. Let s be a dummy source node that has an arrow going into each

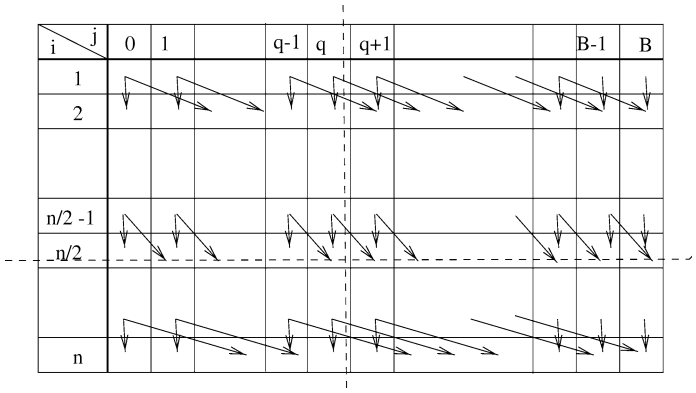


Fig. 7. Dynamic programming table for the 0–1 knapsack problem.

entry of the first row and the first column of M. It is well known that a subset of items which form an actual optimal solution for the original knapsack problem corresponds to a path from s to entry $M(n, B)$ in M. A commonly-used approach for finding such a path is by maintaining a single-source optimal path tree *SST* rooted at s.

From Fig. 7, one can immediately see that the 0–1 knapsack problem has the two properties specified in Section 8.1, and thus is able to make use of our paradigm. For this problem, a natural choice for a sample set for building a clipped tree T is the entries of row $n/2$ of table M plus the dummy source node s and $t = M(n, B)$. The clipped tree T is produced by an algorithm for computing the optimal weight $W_{n,B}$, which only needs to store $O(1)$ rows of M. Suppose the parent node of t in T is $M(n/2, q)$. Then the original problem $P(n, B)$ is reduced to two subproblems, with knapsack sizes q and $B - q$, respectively, and one having the first $n/2$ items while the other having the second $n/2$ items. That is, the two subproblems are $P(n/2, q)$ and $P(n/2, B - q)$. Clearly, the 0–1 knapsack problem is of type A and can be solved in $O(nB)$ time and $O(n + B)$ space.

8.3. Minimum weight, k-link paths

Let $G = (V, E)$ be a weighted graph with nonnegative edge weights, $n = |V|$, and $m = |E|$. Let w_{uv} denote the weight of edge $e(u, v)$ connecting vertices u and v. The minimum weight, k-link path problem is that of finding a minimum weight path $P_k(s, t)$ between two vertices s and t in G such that $P_k(s, t)$ uses no more than k edges ($k \leqslant n$) [23]. Some application problems can be formulated as the minimum weight, k-link path problem (e.g., [21]).

It was shown in [23] that the *weight* $W_k(s, t)$ of a minimum weight, k-link path $P_k(s, t)$ can be computed using the Bellman–Ford method. For initialization, do the following:

$$W_1(s, s) = 0,$$
$$W_1(s, v) = w_{sv}, \quad v \in V - \{s\}. \tag{1}$$

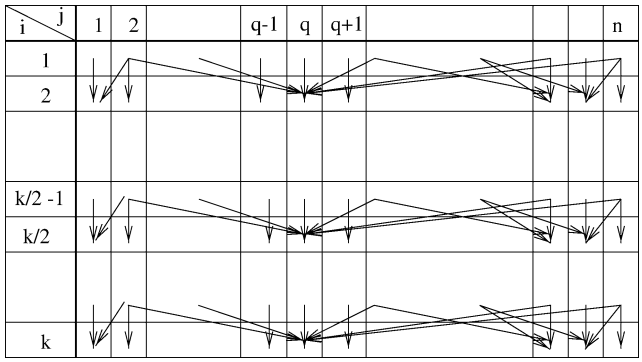


Fig. 8. Dynamic programming table for the minimum weight, k-link path problem.

Then the weight of an $(i+1)$-link path, $i \geqslant 1$, can be computed iteratively as follows:

$$W_{i+1}(s,v) = \min\Big\{W_i(s,v), \min_{e(u,v)\in E}\big\{W_i(s,u) + w_{uv}\big\}\Big\}.$$

This algorithm takes $O(k(n+m))$ time and $O(n)$ working space. To find an actual path $P_k(s,t)$, one can use the standard tree-growing approach, in $O(k(n+m))$ time and $O(kn)$ working space.

By *working space*, we refer to the space used by the algorithm (for bookkeeping, data structures, etc) that is *in addition to* the space needed for input data. The working space may dominate the overall space bound of a graph algorithm in two cases:

(1) the graph G is sparse (i.e., G has $o(kn)$ edges), and
(2) G, although not sparse, can somehow be represented in $o(kn)$ space.

In applications, it is sometimes possible to represent graphs of $O(n^2)$ edges with only $O(n)$ space (e.g., [7,21]).

It is not difficult to see that the above relation among weight values $W_{i+1}(s,v)$ can be captured by a dynamic programming table M of size $k \times n$, in which the rows correspond to the numbers of links, the columns correspond to the vertices of G, and $W_i(s,v)$ is at entry $M(i,v)$. The dependencies among the entries of M are that an entry on row $i+1$ depends only on a number of entries on row i. In particular, $M(i+1,v)$ must depend on $M(i,v)$, and it depends on $M(i,u)$ if and only if $e(u,v) \in E$. Note that the positions of such entries $M(i,u)$ on row i are quite arbitrary for a general graph G (i.e., they can be anywhere in row i). Such a table is depicted in Fig. 8.

From the structure of the table, one can see that our paradigm is applicable to the minimum weight, k-link path problem. As for the 0–1 knapsack problem, the sample set consists of the entries on row $k/2$ together with the source node $M(1,s)$ and $M(k,t)$ (let $M(1,s)$ have an arrow into every other entry of row one). The corresponding clipped tree T is built during the weight computation. The parent vertex of $M(k,t)$ in T, say $M(k/2,q)$, is used to reduce the original problem to two subproblems: Finding a minimum weight, $(k/2)$-link path $P_{k/2}(s,q)$ from s to q and $P_{k/2}(q,t)$ from q to t. To decide whether the problem is of type A or B, note that an entry in M may depend on entries from *any*

columns of the previous row rather than just the columns to its left as for the 0–1 knapsack problem. In this situation, it is not clear to us how to partition the original graph G into two disjoint subgraphs for the two subproblems. Consequently, each subproblem must consider all columns of M, implying that the subproblem uses G as graph when computing a desired optimal $(k/2)$-link path. Hence, finding $P_k(s,t)$ in a general graph is a type B problem, and can be solved in $O(k(n+m)\log k)$ time and $O(n)$ working space. If we sample h rows of M instead of one, we can obtain an $O(k(n+m)\log k/\log(h+1))$ time, $O(nh)$ working space algorithm, where h is any integer such that $1 \leqslant h \leqslant k^{\epsilon}$ for any constant ϵ with $0 < \epsilon < 1$.

It is interesting to note that if G is a directed acyclic graph, then the problem is of type A. In this case, one can first sort the vertices of G by a topological sort [11], and then arrange the columns of the table M according to this sorted vertex order. As a result, each entry of M depends only on entries on columns to its left. Therefore, path $P_{k/2}(s,q)$ (respectively, $P_{k/2}(q,t)$) only involves vertices in G that are between s and q (respectively, q and t) in the column order of M. It follows that we can partition G into two subgraphs, one induced by vertices between s and q (for computing $P_{k/2}(s,q)$) and the other induced by vertices between q and t (for computing $P_{k/2}(q,t)$). Our algorithm for reporting an actual path $P_k(s,t)$ samples the entries of row $k/2$ of M. The recurrence for the time bound of this algorithm is $T(k,N) \leqslant T(k/2,N_1) + T(k/2,N_2) + O(kN)$, where $N = n+m$ and $N_1 + N_2 \leqslant N$. The solution of this recurrence is $T(k,N) = O(kN)$. Therefore, the $P_k(s,t)$ problem on a directed acyclic graph can be solved in $O(k(n+m))$ time and $O(n)$ working space.

8.4. Single-vehicle scheduling problems

The single-vehicle scheduling problem (SVS) studies route scheduling for a vehicle to visit n given sites, each having 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. The problem is a generalization of the Traveling Salesperson Problem and is NP-hard even for some very special cases [4]. 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 (also called *ready times*) and end times (also called *deadlines*) are arbitrary [6]. We shall consider two special cases of this problem.

The first special case of the SVS problem considers the following [29]: A vehicle is to visit a set S of n sites on a straight line. Each site s_i has a ready time r_i but no deadline, and the Euclidean distance between two sites s_i and s_j is denoted by $D_{i,j}$. A vehicle with unit speed starts at site s_1 and wants to visit all the n sites. The goal is to find a feasible schedule for visiting the n sites which has the minimum completion time. (A schedule is feasible if each site is visited on or after its ready time.)

It was proved in [29] that at any time along an optimal schedule, the set of visited sites is the union of two disjoint sets S_1 and S_2, where S_1 includes all sites from s_1 to s_i $(1 \leqslant i \leqslant n)$ while S_2 includes all sites from s_j to s_n $(i < j \leqslant n+1$, with the convention that $S_2 = \phi$ if $j = n+1)$. Furthermore, only s_i or s_j can be the last visited site along the

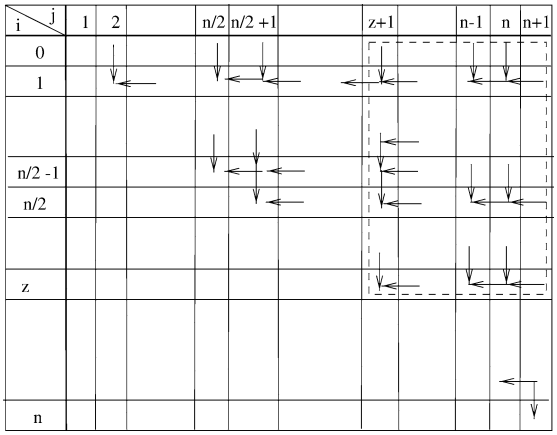


Fig. 9. Dynamic programming table for the first case of the SVS problem.

route. Based on this observation, the following dependencies can be used to compute the minimum completion time.

$$V(i,i,j) = \min\Big\{ \max\big\{r_i, V(i-1,i-1,j) + D_{i-1,i}\big\}, \max\big\{r_i, V(j,i-1,j) + D_{j,i}\big\}\Big\},$$
$$V(j,i,j) = \min\Big\{ \max\big\{r_j, V(j+1,i,j+1) + D_{j+1,j}\big\}, \max\big\{r_j, V(i,i,j+1) + D_{i,j}\big\}\Big\},$$
$$\text{for } 1 \leqslant i < j \leqslant n+1,$$

where $V(k,i,j)$ denotes the minimum completion time of a schedule for visiting sites $s_1, s_2, \ldots, s_i$ and $s_j, s_{j+1}, \ldots, s_n$ such that the last visited site is s_k ($k \in \{i,j\}$). For initialization, let $V(1,1,n+1) = r_1$ and $V(0,0,j) = V(n+1,i,n+1) = \infty$ for all $j > 1$ and $i \leqslant n$. Then, the minimum completion time for visiting all n sites is

$$C_{\min} = \min_{1 \leqslant i \leqslant n} V(i,i,i+1) = \min_{1 < j \leqslant n+1} V(j,j-1,j).$$

Based on the above characterization, the algorithm for computing the minimum completion time for visiting all n sites takes $O(n^2)$ time and $O(n)$ space [29]. However, to produce an actual optimal route for the visit, the algorithm in [29] uses $O(n^2)$ time and space.

Again, the dependencies defined in the above relation can be captured by a dynamic programming table M, where the row (respectively, column) indices correspond to i (respectively, j) in $V(k,i,j)$. Two values, $V(i,i,j)$ and $V(j,i,j)$, are stored in each table entry $M(i,j)$, $0 \leqslant i \leqslant n$ and $1 \leqslant j \leqslant n+1$. Figure 9 depicts such a table. (In fact, only the upper diagonal half of M is really needed.) Let s be the node representing entry $M(1,n+1)$ (i.e., the starting site s_1), and t be a dummy node into which an arrow comes from every entry $M(i,i+1)$ (representing $C_{\min} = \min_{1 \leqslant i \leqslant n} V(i,i,i+1)$). Then, an actual optimal site-visit sequence corresponds to a path from s to t in table M.

Each entry of M depends only on the entry immediately above and the one immediately to its right. Hence, this SVS problem has the two properties specified in Section 8.1 and

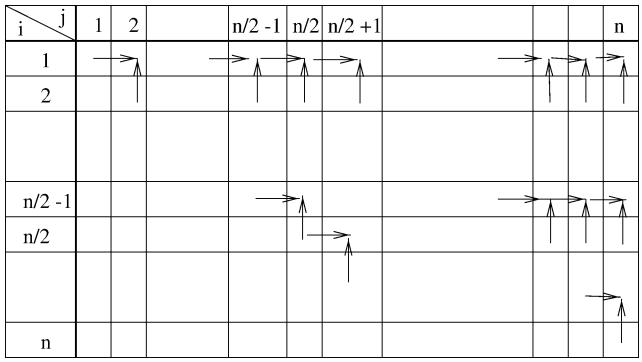


Fig. 10. Dynamic programming table for the second case of the SVS problem.

can make use of our paradigm. Although the shape of the original table M is triangular, we can easily reduce it to a rectangular subtable, as follows. Let s_z be the last visited site of the sought optimal schedule for the original problem (i.e., $C_{\min} = V(z, z, z+1)$). Note that $V(z, z, z+1)$ at entry $M(z, z+1)$ can be found by the algorithm for computing the minimum completion time for visiting all n sites. Then the problem is reduced to the subtable of M from row 0 to row z and from column $z+1$ to column $n+1$ (see Fig. 9). Once this is done, the rest of our algorithm for producing an actual optimal site-visit sequence proceeds in the same manner as the one in Section 8.2.

To summarize, this case of the SVS problem is of type A, and an actual schedule can be found in the same time ($O(n^2)$) and space ($O(n)$) bounds as for computing the minimum completion time.

The second special case of the SVS problem, which is a variation of the first case that we just discussed above, puts a deadline d_i to each site s_i, but lets all n sites have the same start time $r \geqslant 0$ [6]. A useful observation is given in [6] for solving this case: At any time along an optimal feasible schedule, the visited sites form a connected region on the straight line and the last visited site is either the leftmost or rightmost site of that region. Based on this observation, a dynamic programming algorithm, similar to the one for the first case, was given in [6] for computing the minimum completion time in $O(n^2)$ time and $O(n)$ space. However, the algorithm in [6] for producing an actual optimal route for visiting all the sites uses $O(n^2)$ time and space.

Let $V(k, i, j)$ denote the optimal time for visiting sites $s_i, s_{i+1}, \ldots, s_j$ such that the last visited site is s_k ($k \in \{i, j\}$). Initially, let $V(i, i, i) = \delta_i(\max\{r, D_{1,i}\})$, for each $i = 1, 2, \ldots, n$, where the function $\delta_i(x)$ is defined as follows:

$$\delta_i(x) = \begin{cases} x & \text{if } x \leqslant d_i, \\ \infty & \text{otherwise.} \end{cases}$$

The dependencies among optimal times $V(k, i, j)$ can be characterized by

$$V(i, i, j) = \min\{\delta_i(D_{i+1,i} + V(i+1, i+1, j)), \delta_i(D_{j,i} + V(j, i+1, j))\},$$
$$V(j, i, j) = \min\{\delta_j(D_{j-1,j} + V(j-1, i, j-1)), \delta_j(D_{i,j} + V(i, i, j-1))\},$$
$$\text{for } 1 \leqslant i < j \leqslant n.$$

The minimum completion time $C_{\min} = \min\{V(1,1,n), V(n,1,n)\}$. The dynamic programming table M for the above dependencies is shown in Fig. 10. Likewise, two values, $V(i,i,j)$ and $V(j,i,j)$, are stored in each table entry $M(i,j)$.

Comparing the dynamic programming table for this case (Fig. 10) with that for the first case (Fig. 9), one can immediately see the similarities between the two. Hence, it is easy to conclude that this case is of type A, and an actual optimal schedule for visiting all n sites can be obtained in $O(n^2)$ time and $O(n)$ space.

Our examples in this section have demonstrated that a number of problems solvable by a certain dynamic programming approach can utilize our general paradigm to reduce their space bounds or to achieve a trade-off between their time and space bounds.

References

[1] E.G. Anagnostou, L.J. Guibas, V.G. Polimenis, Topological sweeping in three dimensions, in: Proc. SIGAL International Symp. on Algorithms, in: Lecture Notes in Comput. Sci., Vol. 450, Springer-Verlag, 1990, pp. 310–317.
[2] T. Asano, L.J. Guibas, T. Tokuyama, Walking in an arrangement topologically, Internat. J. Comput. Geom. Appl. 4 (2) (1994) 123–151.
[3] T. Asano, T. Tokuyama, Topological walk revisited, in: Proc. 6th Canadian Conf. on Computational Geometry, 1994, pp. 1–6.
[4] L. Bodin, B. Golden, A. Assad, M. Ball, Routing and scheduling of vehicles and crews: The state of the art, Comput. Oper. Res. 10 (1983) 62–212.
[5] P. Bose, W. Evans, D. Kirkpatrick, M. McAllister, J. Snoeyink, Approximating shortest paths in arrangements of lines, in: Proc. 8th Canadian Conf. on Computational Geometry, 1996, pp. 143–148.
[6] C. Chan, G.H. Young, Single-vehicle scheduling problem on a straight line with time window constraints, in: Proc. 1st Annual International Conf. on Computing and Combinatorics, 1995, pp. 617–626.
[7] D.Z. Chen, X. Hu, P.J. Blatner, Efficient algorithms for orthogonal polygon approximation, in: Proc. IEEE International Symposium on Circuits and Systems, 1996, pp. 779–782.
[8] D.Z. Chen, J. Xu, Peeling an arrangement topologically, in: Proc. 4th CGC Workshop on Computational Geometry, John Hopkins University, Baltimore, MD, 1999.
[9] D.Z. Chen, S. Luan, J. Xu, Topological Peeling and Implementation, in: Proc. 12th Annual International Symposium on Algorithms And Computation (ISAAC), in: Lecture Notes in Comput. Sci., Vol. 2223, Springer-Verlag, 2001, pp. 454–466.
[10] V. Chvatal, D.A. Klarner, D.E. Knuth, Selected combinatorial research problems, in: STAN-CS-72-292, 26, Stanford University, June, 1972.
[11] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, McGraw–Hill, New York, 1990.
[12] T. Dey, Improved bounds on planar k-sets and k-levels, in: Proc. 38th Annual IEEE Symp. on Foundations of Computer Science, 1997, pp. 156–161.
[13] H. Edelsbrunner, Algorithms in Combinatorial Geometry, Springer-Verlag, Berlin, 1987.
[14] H. Edelsbrunner, L.J. Guibas, Topologically sweeping an arrangement, J. Comput. System Sci. 38 (1989) 165–194.
[15] H. Edelsbrunner, E.P. Mücke, Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms, ACM Trans. Graphics 9 (1990) 66–104.
[16] H. Edelsbrunner, E. Welzl, Constructing belts in two-dimensional arrangements with applications, SIAM J. Comput. 15 (1986) 271–284.
[17] D. Eppstein, D. Hart, An efficient algorithm for shortest paths in vertical and horizontal segments, in: Proc. 5th International Workshop on Algorithms and Data Structures, in: Lecture Notes in Comput. Sci., Vol. 1272, Springer-Verlag, 1997, pp. 234–247.
[18] D. Eppstein, D. Hart, Shortest paths in an arrangement with k line orientations, in: Proc. 10th Annual ACM–SIAM Symp. Discrete Algorithms, 1999, pp. 310–316.

[19] P. Erdös, L. Lovász, A. Simmons, 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.
[20] D.S. Hirschberg, A linear-space algorithm for computing maximal common subsequences, Comm. ACM 18 (6) (1975) 341–343.
[21] X. Hu, D.Z. Chen, R. Sambandam, Efficient list-approximation techniques for floorplan area minimization, ACM Trans. Design Automat. Electron. Systems 6 (3) (2001) 372–400.
[22] P.N. Klein, S. Rao, M.H. Rauch, S. Subramanian, Faster shortest-path algorithms for planar graphs, in: Proc. 26th Annual ACM Symp. Theory of Computing, 1994, pp. 27–37.
[23] E.L. Lawler, Combinatorial Optimization: Networks and Matroids, Holt, Rinehart and Winston, New York, 1976.
[24] L. Lovász, On the number of halving lines, Ann. Univ. Sci. Budapest. Eötvös Sec. Math. 14 (1971) 107–108.
[25] U. Manber, Introduction to Algorithms: A Creative Approach, Addison–Wesley, Reading, MA, 1989.
[26] S. Martello, P. Toth, Knapsack Problems: Algorithms and Computer Implementations, Wiley, New York, 1990.
[27] C.H. Papadimitriou, K. Steiglitz, Combinatorial Optimization: Algorithms and Complexity, Prentice Hall, Englewood Cliffs, NJ, 1982.
[28] F.P. Preparata, M.I. Shamos, Computational Geometry: An Introduction, Springer-Verlag, New York, 1985.
[29] H.N. Psaraftis, M.M. Solomon, T.L. Magnanti, T.-U. Kim, Routing and scheduling on a shoreline with release times, Manage. Sci. 36 (2) (1990) 212–223.
[30] S.S. Skiena, The Algorithm Design Manual, Springer-Verlag, New York, 1998.
[31] R.A. Wagner, M.J. Fischer, The string-to-string correction problem, J. ACM 21 (1) (1974) 168–173.