Shortest Paths in Directed Planar Graphs with Negative Lengths: a Linear-Space $O(n \log^2 n)$-Time Algorithm

PHILIP N. KLEIN and SHAY MOZES
Brown University
and
OREN WEIMANN
Massachusetts Institute of Technology

We give an $O(n \log^2 n)$-time, linear-space algorithm that, given a directed planar graph with positive and negative arc-lengths, and given a node $s$, finds the distances from $s$ to all nodes.

Categories and Subject Descriptors: G.2.2 [Graph Theory]: Graph algorithms; Path and circuit problems; F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems—Computations on discrete structures

General Terms: Theory, Algorithms

Additional Key Words and Phrases: planar graphs, Monge, shortest paths, replacement paths

1. INTRODUCTION

The problem of directed shortest paths with negative lengths is as follows: Given a directed graph $G$ with positive and negative arc-lengths containing no negative cycles,\(^1\) and given a source node $s$, find the distances from $s$ to all the nodes in the graph. This is a classical problem in combinatorial optimization. For general graphs, the Bellman-Ford algorithm solves the problem in $O(mn)$ time, where $m$ is the number of arcs and $n$ is the number of nodes. For integer lengths whose absolute values are bounded by $N$, the algorithm of Gabow and Tarjan [1989] takes $O(\sqrt{nm \log(nN)})$. For integer lengths exceeding $-N$, the algorithm of Goldberg [1995] takes $O(\sqrt{nm \log N})$ time. For non-negative lengths, the problem is easier and can be solved using Dijkstra’s algorithm in $O((n + m) \log n)$ time if elementary data structures are used [Johnson 1977], and in $O(n \log n + m)$ time when implemented with Fibonacci heaps [Fredman and Tarjan 1987].

For planar graphs, there has been a series of results yielding progressively better bounds. The first algorithm that exploited planarity was due to Lipton, Rose, and Tarjan [1979], who gave an $O(n^{3/2})$ algorithm. Henzinger et al. [1997] gave an $O(n^{4/3} \log^{2/3} D)$ algorithm where $D$ is the sum of the absolute values of the lengths. Fakcharoenphol and Rao [2006] gave an algorithm requiring $O(n \log^3 n)$ time and $O(n \log n)$ space.

Our result is as follows:

**Theorem 1.1.** There is an $O(n \log^2 n)$-time, linear-space algorithm to find shortest paths in planar directed graphs with negative lengths.

Applications.

In addition to being a fundamental problem in combinatorial optimization, shortest paths in planar graphs with negative lengths arises in solving other problems. Miller and Naor [1995] show that, by using planar duality, the following problem can be reduced to shortest paths in a planar directed graph:

---

\(^1\) Algorithms for this problem can also be used to detect negative cycles.

---

Author’s address: P.N. Klein, S. Mozes, Department of Computer Science, Brown University, Providence RI 02912-1910, USA. {klein,shay}@cs.brown.edu. O. Weimann, Massachusetts Institute of Technology, Cambridge, MA 02139, USA. oweimann@mit.edu.

Klein and Mozes supported by NSF Grant CCF-0635089. Work done while Klein was visiting MIT.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 20 ACM 0000-0000/20/0000-0001 $5.00
Feasible circulation: Given a directed planar graph with upper and lower arc-capacities, find an assignment of flow to the arcs so that each arc’s flow is between the arc’s lower and upper capacities, and, for each node, the flow into the node equals the flow out.

They further show that the following problem can in turn be reduced to feasible circulation:

Feasible flow: Given a directed planar graph with arc-capacities and node-demands, find an assignment of flow that respects the arc-capacities and such that, for each node, the flow into the node minus the flow out equals the node’s demand.

For integer-valued capacities and demands, the solutions obtained to the above problems are integer-valued. Consequently, as Miller and Naor point out, the problem of finding a perfect matching in a bipartite planar graph can be solved using an algorithm for feasible flow.

Our new shortest-path algorithm thus gives $O(n \log^2 n)$ algorithms for bipartite planar perfect matching, feasible flow, and feasible circulation.

Several techniques for computer vision, including image segmentation algorithms by Cox, Rao, and Zhong [1996] and by Jermyn and Ishikawa [2001; 2001], and a stereo matching technique due to Veksler [2002], involve finding negative-length cycles in graphs that are essentially planar. Thus our algorithm can be used to implement these techniques.

Summary of the Algorithm.

Like the other planarity-exploiting algorithms for this problem, our algorithm uses planar separators [Lipton and Tarjan 1979; Miller 1986]. Given an $n$-node planar embedded directed graph $G$ with arc-lengths, and given a source node $s$, the algorithm first finds a Jordan curve $C$ that passes through $O(\sqrt{n})$ nodes (and no arcs) such that between $n/3$ and $2n/3$ nodes are enclosed by $C$.

A node through which $C$ passes is called a boundary node. Cutting the planar embedding along $C$ and duplicating boundary nodes yields two subgraphs $G_0$ and $G_1$ such that, for $i = 0, 1$, in $G_i$ the boundary nodes lie on the boundary of a single face $F_i$. Refer to Fig. 1 for an illustration. Let $r$ be an arbitrary boundary node.

Our algorithm consists of five stages. The first four stages alternate between working with negative lengths and working with only positive lengths.

Recursive call: The first stage recursively computes the distances from $r$ within $G_i$ for $i = 0, 1$. The remaining stages use these distances in the computation of the distances in $G$. 2
In a special kind of matrix. Each iteration is performed in $O(n \log n)$ time. The number of iterations is $O(\sqrt{n})$, so the overall running time of the algorithm is $O(n^{\alpha(n)})$.

Intra-part boundary-distances:. For each graph $G_i$ we use a method due to Klein [2005] to compute all distances in $G_i$ between boundary nodes. This takes $O(n \log n)$ time.

Single-source inter-part boundary distances:. A shortest path in $G$ passes back and forth between $G_0$ and $G_1$. Refer to Fig. 1 and Fig. 2 for an illustration. We use a variant of Bellman-Ford to compute the distances in $G$ from $r$ to all the boundary nodes. Alternating iterations use the all-boundary-distances in $G_0$ and $G_1$. Because the distances have a Monge property [Monge 1781] (discussed later), each iteration can be implemented by two executions of an algorithm due to Klawe and Kleitman [1990] for finding row-minima in a special kind of matrix. Each iteration is performed in $O(\sqrt{n} \alpha(n))$, where $\alpha(n)$ is the inverse Ackerman function. The number of iterations is $O(\sqrt{n})$, so the overall time for this stage is $O(n \alpha(n))$.

Single-source inter-part distances:. The distances computed in the previous stages are used, together with a Dijkstra computation within a modified version of each $G_i$, to compute the distances in $G$ from $r$ to all the nodes. Dijkstra’s algorithm requires the lengths in $G_i$ to be non-negative, but we can use the recursively computed distances to transform the lengths in $G_i$ into non-negative lengths without changing the shortest paths. This stage takes $O(n \log n)$ time.

Rerooting single-source distances:. The algorithm has obtained distances in $G$ from $r$. In the last stage these distances are used to transform the lengths in $G$ into nonnegative lengths, and again uses Dijkstra’s algorithm, this time to compute distances from $s$. This stage also requires $O(n \log n)$ time.

Relation to Previous Work.

All known planarity-exploiting algorithms for this problem, starting with that of Lipton, Rose, and Tarjan [1979], use planar separators, and use Bellman-Ford on a dense graph whose nodes are those comprising a planar separator. The algorithm of Henzinger et al. [1997] achieved an improvement by using a multi-part decomposition based on planar separators. Fakcharoenphol and Rao’s algorithm [2006] introduced several innovations. Among these is the exploitation of a Monge property of the boundary-to-boundary distances to enable fast implementation of an iteration of Bellman-Ford. We exploit the Monge property to the same end, although we do so using a different technique. Another key ingredient of Fakcharoenphol and Rao is an ingenious data structure to implement a version of Dijkstra’s algorithm, where each node is processed $O(\log n)$ times (rather than once, as in Dijkstra’s algorithm) and many arcs can be relaxed at once.

A central concept of the algorithm of Fakcharoenphol and Rao is the dense distance graph. This consists of a recursive decomposition of a graph using separators, together with a table for each subgraph arising in the decomposition giving the distances between all boundary nodes for that subgraph. This structure has size $\Omega(n \log n)$ for an $n$-node graph. The first phase of their algorithm computes this structure in $O(n \log^3 n)$ time. The second phase uses the structure to compute distances from a node to all other nodes, also in $O(n \log^3 n)$ time.

The structure of our algorithm is different—it is a simple divide-and-conquer, in which the recursive problem is the same as the original problem, single-source shortest-path distances. In addition, we require no data structures aside from the dynamic-tree data structure used in [Klein 2005] and a basic priority queue for implementing Dijkstra’s algorithm.

The Replacement-Paths Problem.

We note that finding row-minima based on a Monge property can be used for other problems in planar graphs. Consider the replacement-paths problem: we are given a directed graph with non-negative arc lengths and two nodes $s$ and $t$. We are required to compute, for every arc $e$ in the shortest path between $s$ and $t$, the length of an $s$-to-$t$ shortest path that avoids $e$.

Emek et al. [2008] give an $O(n \log^3 n)$-time algorithm for solving the replacement-paths problem in a directed planar graph. Procedure District in Section 4 of their paper solves a problem that can be viewed as finding the row-minima of a certain matrix which has a Monge property. By further exploiting the Monge property, we obtain an $O(n \log^2 n)$-time algorithm. Details appear in Section 7.

---

2This stage can actually be implemented in $O(n)$ using the algorithm of Henzinger et al. [1997]. This however does not change the overall running time of the algorithm.
Theorem 1.2. There is an $O(n \log^2 n)$-time algorithm for solving the replacement-paths problem in a directed planar graph.

2. PRELIMINARIES

2.1 Embedded Planar Graphs

A planar embedding of a graph assigns each node to a distinct point on the plane, and assigns each edge to a simple arc between the points corresponding to its endpoints, with the property that no arc-arc or arc-point intersections occur except for those corresponding to edge-node incidence in the graph. A graph is planar if it has a planar embedding. Consider the set of points on the plane that are not assigned to any node or edge; each connected component of this set is a face of the embedding.

2.2 Jordan Separators for Embedded Planar Graphs

Miller [1986] gave a linear-time algorithm that, given a triangulated two-connected $n$-node planar embedded graph, finds a simple cycle separator consisting of at most $2\sqrt{2n}$ nodes, such that at most $2n/3$ nodes are strictly enclosed by the cycle, and at most $2n/3$ nodes are not enclosed.

For an $n$-node planar embedded graph $G$ that is not necessarily triangulated or two-connected, we define a Jordan separator to be a Jordan curve $C$ that intersects the embedding of the graph only at nodes such that at most $2n/3$ nodes are strictly enclosed by the curve and at most $2n/3$ nodes are not enclosed. The nodes intersected by the curve are called boundary nodes and denoted $V_c$. To find a Jordan separator with at most $2\sqrt{2n}$ boundary nodes, add artificial edges with sufficiently large lengths to triangulate the graph and make it two-connected without changing the distances in the graph. Now apply Miller’s algorithm.

The internal part of $G$ with respect to $C$ is the embedded subgraph consisting of the nodes and edges enclosed by $C$, i.e. including the nodes intersected by $C$. Similarly, the external part of $G$ with respect to $C$ is the subgraph consisting of the nodes and edges not strictly enclosed by $C$, i.e. again including the nodes intersected by $C$.

Let $G_1(G_0)$ denote the internal (external) part of $G$ with respect to $C$. Since $C$ is a Jordan curve, the set of points of the plane strictly exterior to $C$ form a connected region. Furthermore, it contains no point or arc corresponding to a node or edge of $G_1$. Therefore, the region remains connected when these points and arcs are removed, so the region is a subset of some face of $G_1$. Since every boundary node is intersected by $C$, it follows that all boundary nodes lie on the boundary of a single face of $G_1$. Similarly, in $G_0$, all boundary nodes lie on the boundary of a single face.

2.3 Monotonicity, Monge and Matrix Searching

A matrix $M = (M_{ij})$ is totally monotone if for every $i, i', j, j'$ such that $i < i', j < j'$ and $M_{ij} \leq M_{i'j'}$, we also have $M_{i'j} \leq M_{ij'}$. Totally monotone matrices were introduced by Aggarwal et al. in [Aggarwal et al. 1987], who showed that a wide variety of problems in computational geometry could be reduced to the problem of finding row-maxima or row-minima in totally monotone matrices. Aggarwal et al. also give an algorithm, nicknamed SMAWK, that, given a totally monotone $n \times m$ matrix $M$, finds all row-maxima of $M$ in just $O(n + m)$ time. It is easy to see that by negating each element of $M$ and reversing the order of its columns, SMAWK can be used to find the row-minima of $M$ as well.

A matrix $M = (M_{ij})$ is convex Monge (concave Monge) if for every $i, i', j, j'$ such that $i < i', j < j'$, we have $M_{ij} + M_{i'j'} \geq M_{ij'} + M_{i'j}$ ($M_{ij} + M_{i'j'} \leq M_{ij'} + M_{i'j}$). It is immediate that if $M$ is convex Monge then it is totally monotone. It is also easy to see that the matrix obtained by transposing $M$ is also totally monotone. Thus SMAWK can also be used to find the column minima and maxima of a convex Monge matrix.

A falling staircase matrix is defined in [Aggarwal and Klawe 1990] and [Klawe and Kleitman 1990] to be a lower triangular fragment of a totally monotone matrix. More precisely, $(M, \{f(i)\}_{0 \leq i \leq n+1})$ is an $n \times m$ falling staircase matrix if

1. for $i = 0, \ldots, n + 1$, $f(i)$ is an integer with $0 = f(0) < f(1) \leq f(2) \leq \cdots \leq f(n) < f(n + 1) = m + 1$.
2. $M_{ij}$ is a real number if and only if $1 \leq i \leq n$ and $1 \leq j \leq f(i)$. Otherwise, $M_{ij}$ is blank.
3. (total monotonicity) for $i < k$ and $j < l \leq f(i)$, and $M_{ij} \leq M_{kl}$, we have $M_{kj} \leq M_{kl}$.
Finding the row-maxima in a falling staircase matrix can be easily done using SMAWK in $O(n + m)$ time after replacing the blanks with sufficiently small numbers so that the resulting matrix is totally monotone. However, this trick does not work for finding the row-minima. Aggarwal and Klawe [1990] give an $O(m \log \log n)$ time algorithm for finding row-minima in falling staircase matrices of size $n \times m$. Klawe and Kleitman give in [Klawe and Kleitman 1990] a more complicated algorithm that computes the row-minima of an $n \times m$ falling staircase matrix in $O(m \alpha(n) + n)$ time, where $\alpha(n)$ is the inverse Ackerman function. If $M$ satisfies the above conditions with total monotonicity replaced by the convex Monge property then $M$ and the matrix obtained by transposing $M$ and reversing the order of rows and of columns are falling staircase. In this case both algorithms can be used to find the column-minima as well as the row-minima.

### 2.4 Price Functions and Reduced Lengths

For a directed graph $G$ with arc-lengths $\ell(\cdot)$, a **price function** is a function $\phi$ from the nodes of $G$ to the reals. For an arc $uv$, the **reduced length with respect to $\phi$** is $\ell_\phi(uv) = \ell(uv) + \phi(u) - \phi(v)$. A **feasible** price function is a price function that induces nonnegative reduced lengths on all arcs of $G$.

Feasible price functions are useful in transforming a shortest-path problem involving positive and negative lengths into one involving only nonnegative lengths, which can then be solved using Dijkstra’s algorithm. For any nodes $s$ and $t$, for any $s$-to-$t$ path $P$, $\ell_\phi(P) = \ell(P) + \phi(s) - \phi(t)$. This shows that an $s$-to-$t$ path is shortest with respect to $\ell_\phi(\cdot)$ if and only if it is shortest with respect to $\ell(\cdot)$. Moreover, the $s$-to-$t$ distance with respect to the original lengths $\ell(\cdot)$ can be recovered by adding $\phi(t) - \phi(s)$ to the $s$-to-$t$ distance with respect to $\ell_\phi(\cdot)$.

Suppose $\phi$ is a feasible price function. Running Dijkstra’s algorithm with the reduced lengths and modifying the distances thereby computed to obtain distances with respect to the original lengths will be called **running Dijkstra’s algorithm with $\phi$**.

An example of a feasible price function comes from single-source distances. Suppose that, for some node $r$ of $G$, for every node $v$ of $G$, $\phi(v)$ is the $r$-to-$v$ distance in $G$ with respect to $\ell(\cdot)$. Then for every arc $uv$, $\phi(u) \leq \phi(u) + \ell(uv)$, so $\ell_\phi(uv) \geq 0$. Here we assume, without loss of generality, that all distances are finite (i.e., that all nodes are reachable from $r$) since we can always add arcs with sufficiently large lengths to make all nodes reachable without affecting the shortest paths in the graph.

### 2.5 Multiple-Source Shortest Paths: Computing Boundary-to-Boundary Distances

Klein [2005] gives a multiple-source shortest-path algorithm with the following properties. The input consists of a directed planar embedded graph $G$ with non-negative arc-lengths, and a face $f$. For each node $u$ in turn on the boundary of $f$, the algorithm computes (an implicit representation of) the shortest-path tree rooted at $u$. The basic algorithm takes a total of $O(n \log n)$ time and $O(n)$ space on an $n$-node input graph. In addition, given a set of pairs $(u,v)$ of nodes of $G$ where $u$ is on the boundary of $f$, the algorithm computes the $u$-to-$v$ distances. The time per distance computed is $O(\log n)$. In particular, given a set $S$ of $O(\sqrt{n})$ nodes on the boundary of a single face, the algorithm can compute all $S$-to-$S$ distances in $O(n \log n)$ time.

In fact, the multiple-source shortest-path algorithm does not require that the arc-lengths be nonnegative if the input also includes a table of distances to all nodes from some node on the face $f$. This observation follows from careful inspection of the algorithm [Klein 2005] itself. Alternatively, it also follows from the price-function technique of Section 2.4: the table of distances can be used to obtain nonnegative reduced lengths, and these lengths can be supplied as input to Klein’s algorithm.

### 3. THE ALGORITHM

The high-level description of the algorithm appears in Figure 3. After finding a Jordan separator and selecting a boundary node as a temporary source node, the algorithm consists of five major steps. The **recursive call** step is straightforward. Computing **intra-part boundary distances** uses the algorithm described in Section 2.5. Computing **single-source inter-part boundary distances** is described in Section 4; it is based on the Bellman-Ford algorithm. **Single-source inter-part distances** is described in Section 5, and is based on Dijkstra’s algorithm. It yields distances to all nodes from the temporary source node. These distances constitute a feasible price function, as described in Section 2.4, that enables us, in **rerooting single-source distances**, to use Dijkstra’s algorithm once more to finally compute distances from the given source.
4. COMPUTING SINGLE-SOURCE INTER-PART BOUNDARY DISTANCES

In this section we describe how to efficiently compute the distances in $G$ from $r$ to all boundary nodes (i.e., the nodes of $V_c$). This is done using $\delta_0$ and $\delta_1$, the all-pairs distances in $G_0$ and in $G_1$ between nodes in $V_c$ which were computed in the previous stage.

**Theorem 4.1.** Let $G$ be a directed graph with arbitrary arc-lengths. Let $C$ be a Jordan separator in $G$ and let $G_0$ and $G_1$ be the external and internal parts of $G$ with respect to $C$. Let $\delta_0$ and $\delta_1$ be the all-pairs distances between nodes in $V_c$ in $G_0$ and in $G_1$ respectively. Let $r \in V_c$ be an arbitrary node on the boundary. There exists an algorithm that, given $\delta_0$ and $\delta_1$, computes the from-$r$ distances in $G$ to all nodes in $V_c$ in $O(|V_c|^2 \alpha(|V_c|))$ time and $O(|V_c|)$ space.

The rest of this section describes the algorithm, thus proving Theorem 4.1. The following structural lemma stands in the core of the computation. The same lemma has been implicitly used before by previous planarity-exploiting algorithms.

**Lemma 4.2.** Let $P$ be a simple $r$-to-$v$ shortest path in $G$, where $v \in V_c$. Then $P$ can be decomposed into at most $|V_c|$ subpaths $P = P_1 P_2 P_3 \ldots$, where the endpoints of each subpath $P_i$ are boundary nodes, and $P_i$ is a shortest path in $G_i mod 2$.

**Proof.** Consider a decomposition of $P = P_1 P_2 P_3 \ldots$ into maximal subpaths such that the subpath $P_i$ consists of nodes of $G_{i \mod 2}$. Since $r$ and $v$ are boundary nodes, and since the boundary nodes are the only nodes common to both $G_0$ and $G_1$, each subpath $P_i$ starts and ends on a boundary node. If $P_i$ were not a shortest path in $G_{i \mod 2}$ between its endpoints, replacing $P_i$ in $P$ with a shorter path would yield a shorter $r$-to-$v$ path, a contradiction.

It remains to show that there are at most $|V_c|$ subpaths in the decomposition of $P$. Since $P$ is simple,
Each node, and in particular each boundary node appears in $P$ at most once. Hence there can be at most $|V_c| - 1$ non-empty subpaths in the decomposition of $P$. Note, however, that if $P$ starts with an arc of $G_0$ then $P_1$ is a trivial empty path from $r$ to $r$. Hence, $P$ can be decomposed into at most $|V_c|$ subpaths.

Lemma 4.2 gives rise to a dynamic-programming solution for calculating the from-$r$ distances to nodes of $C$, which resembles the Bellman-Ford algorithm. The pseudocode is given in Fig. 4. Note that, at this level of abstraction, there is nothing novel about this dynamic program. Our contribution is in an efficient implementation of Step 4.

The algorithm consists of $|V_c|$ iterations. On odd iterations, it uses the boundary-to-boundary distances in $G_1$, and on even iterations it uses the boundary-to-boundary distances in $G_0$.

**Lemma 4.3.** After the table $e_j$ is updated by the algorithm, $e_j[v]$ is the length of a shortest path in $G$ from $r$ to $v$ that can be decomposed into at most $j$ subpaths $P = P_1P_2P_3 \ldots P_j$, where the endpoints of each subpath $P_i$ are boundary nodes, and $P_j$ is a shortest path in $G_{i \mod 2}$.

**Proof.** By induction on $j$. For the base case, $e_0$ is initialized to be infinity for all nodes other than $r$, trivially satisfying the lemma. For $j > 0$, assume that the lemma holds for $j - 1$, and let $P$ be a shortest path in $G$ that can be decomposed into $P_1P_2 \ldots P_j$ as above. Consider the prefix $P'$, $P' = P_1P_2 \ldots P_{j-1}$. $P'$ is a shortest $r$-to-$w$ path in $G$ that can be decomposed into at most $j - 1$ subpaths as above for some boundary node $w$. Hence, by the inductive hypothesis, when $e_j$ is updated in Step 4, $e_{j-1}[w]$ already stores the length of $P'$. Thus $e_j[w]$ is updated in Step 4 to be at most $e_{j-1}[w] + \delta_j$, the length of the shortest path in $G_{j \mod 2}$ from $w$ to $v$. Since, by definition, $\delta_j$ is the length of the shortest path in $G_{j \mod 2}$ from $w$ to $v$, it follows that $e_j[w]$ is at least the length of $P$. For the opposite direction, since for any boundary node $w$, $e_{j-1}[w]$ is the length of some path that can be decomposed into at most $j - 1$ subpaths as above, $e_j[w]$ is updated in Step 4 to the length of some path that can be decomposed into at most $j$ subpaths as above. Hence, since $P$ is the shortest such path, $e_j[w]$ is at least the length of $P$. 

From Lemma 4.2 and Lemma 4.3, it immediately follows that the table $e_{|V_c|}$ stores the from-$r$ shortest path distances in $G$, so the assignment in Step 5 is justified, and the table $B$ also stores these distances.

We now show how to perform all the minimizations in the $j$th iteration of Step 4 in $O(|V_c|\alpha(|V_c|))$ time. Let $i = j \mod 2$, so this iteration uses distances in $G_i$. Since all boundary nodes lie on the boundary of a single face of $G_i$, there is a natural cyclic clockwise order $v_1, v_2, \ldots, v_{|V_c|}$ on the nodes in $V_c$. Define a $|V_c| \times |V_c|$ matrix $A$ with elements $A_{\ell k} = e_{j-1}(v_k) + \delta_j(v_k, v\ell)$. Note that computing all minima in Step 4 is equivalent to finding the column-minima of $A$. We define the upper triangle of $A$ to be the elements of $A$ on or above the main diagonal. More precisely, the upper triangle of $A$ is the portion $\{A_{\ell k} : k < \ell\}$ of $A$. Similarly, the lower triangle of $A$ consists of all the elements on or below the main diagonal of $A$.

**Lemma 4.4.** For any four indices $k, k', \ell, \ell'$ such that either $A_{k\ell}, A_{k'\ell'}, A_{k'\ell}$ and $A_{k\ell'}$ are all in $A$'s upper triangle, or are all in $A$'s lower triangle (i.e., either $1 \leq k \leq k' \leq \ell \leq \ell' \leq |V_c|$ or $1 \leq \ell \leq \ell' \leq k \leq k' \leq |V_c|$), the convex Monge property holds:

\[
A_{k\ell} + A_{k'\ell'} \geq A_{k\ell'} + A_{k'\ell}.
\]

**Proof.** Consider the case $1 \leq k \leq k' \leq \ell \leq \ell' \leq |V_c|$, as in Fig. 5. Since $G_i$ is planar, any pair of paths in $G_i$ from $k$ to $\ell$ and from $k'$ to $\ell'$ must cross at some node $w$ of $G_i$. Let $b_k = e_{j-1}(v_k)$ and let $b_{k'} = e_{j-1}(v_{k'})$. Let $\Delta(u, v)$ denote the $u$-to-$v$ distance in $G_i$ for any nodes $u, v$ of $G_i$. Note that $\Delta(u, v) = \delta_j(u, v)$ for
The case \(1 \leq \ell \leq \ell'\) time. Note that we never actually compute and store the entire matrix \(A\).

Let \(A\) be the column-minima of this falling staircase matrix. By [Klawe and Kleitman 1990], the column-minima of this falling staircase matrix can be computed in \(O(|V_c|)\) time. Another consequence of Lemma 4.4 is that the column-minima of the upper triangle of \(A\) may also be computed using the algorithm in [Klawe and Kleitman 1990]. To see this consider a counterclockwise ordering of the nodes of \(|V_c|\) such that \(v'_k = v_{|V_c|+1-k}\). This reverses the order of both the rows and the columns of \(A\), thus turning its upper triangle into a lower triangle. Again, replacing the upper triangle of this matrix with blanks yields a falling staircase matrix.

We thus conclude that \(A\)'s column-minima can be computed in \(O(2|V_c|\cdot \alpha(|V_c|)) = O(|V_c|^2 \cdot \alpha(|V_c|))\) time. Note that we never actually compute and store the entire matrix \(A\) as this would take \(O(|V_c|^2)\) time. We compute the entries necessary for the computation on the fly in \(O(1)\) time per element.

Lemma 4.5 shows that the time it takes the algorithm described in Fig. 4 to compute the distances between \(r\) and all nodes of \(V_c\) is \(O(|V_c|^2 \cdot \alpha(|V_c|))\). We have thus proved Theorem 4.1. The choice of separator ensures \(|V_c| = O(\sqrt{n})\), so this computation is performed in \(O(n\alpha(n))\) time.

5. COMPUTING SINGLE-SOURCE INTER-PART DISTANCES

In the previous section we showed how to compute a table \(B\) that stores the distances from \(r\) to all the boundary nodes in \(G\). In this section we describe how to compute the distances from \(r\) to all other nodes of \(G\). We do so by computing tables \(d'_0\) and \(d'_1\) where \(d'_i(v)\) is the \(r\)-to-\(v\) distance in \(G\) for every node \(v\) of \(G_i\).
We will show that at each stage of our algorithm, the necessary information has been correctly computed in the time for this stage. This however does not change the overall running time of our algorithm.

Recall that we have already computed the table $d_i$ that stores the $r$-to-$v$ distance in $G_i$ for every node $v$ of $G_i$.

The pseudocode given in Fig. 6 describes how to compute $d'$. The idea is to use $d_i$ and $B$ in order to construct a modified version of $G_i$, denoted $G'_i$, so that the from-$r$ distances in $G'_i$ are the same as the from-$r$ distances in $G$. We then construct a feasible price function $\phi_i$ for $G'_i$ and use Dijkstra’s algorithm on $G'_i$ with the price function $\phi_i$ in order to compute these from-$r$ distances.

The following two lemmas motivate the definition of $G'_i$ and show that it captures the true from-$r$ distances in $G$.

**Lemma 5.1.** Let $P$ be an $r$-to-$v$ shortest path in $G$, where $v \in G_i$. Then $P$ can be expressed as $P = P_1P_2$, where $P_1$ is a (possibly empty) shortest path from $r$ to a node $u \in V_c$, and $P_2$ is a (possibly empty) shortest path from $u$ to $v$ that only visits nodes of $G_i$.

**Proof.** Let $u$ be the last boundary node visited by $P$. Let $P_1$ be the $r$-to-$u$ prefix of $P$, and let $P_2$ be the $u$-to-$v$ suffix of $P$. Since $P_1$ and $P_2$ are subpaths of a shortest path in $G$, they are each shortest as well. By choice of $u$, $P_2$ has no internal boundary nodes, so it is a path in $G_i$. □

**Lemma 5.2.** Let $G'_i$ be the graph obtained from $G_i$ by removing arcs entering $r$, and adding an arc $ru$ of length $B[u]$ for every boundary node $u$. The from-$r$ distances in $G'_i$ are equal to the from-$r$ distance in $G$.

**Proof.** Distances in $G'_i$ are not shorter than in $G$ since each arc of $G'_i$ corresponds to some path in $G$. For the opposite direction, consider an $r$-to-$v$ shortest path in $G$. Let $P_1, P_2, u$ be as in Lemma 5.1. $P_1$ is a shortest path in $G$ from $r$ to some $u \in V_c$. By definition of $G'_i$, the length of the new arc $ru$ in $G'_i$ is equal to the length of $P_1$ in $G$. Furthermore, $P_2$ is a path in $G'_i$ since it only consists of arcs in $G_i$. Since the shortest $r$-to-$v$ path is simple, non of these arcs enters $r$, and therefore all of them are in $G'_i$. Hence the length of the path in $G'_i$ that consists of the new arc $ru$ followed by $P_2$ equals the length of $P$ in $G$. □

Since $G'_i$ contains arcs not in $G_i$, we cannot use the from-$r$ distances in $d_i$ as a feasible price function. We slightly modify them to ensure non-negativity as shown by the following lemma.

**Lemma 5.3.** $\phi_i$ defined in Step 3 of Fig. 6 is a feasible price function for $G'_i$.

**Proof.** First note that $d_i[r] = 0$ and $B[r] = 0$, so $p_i \geq d_i[r]$. Let $uv$ be an arc of $G'_i$. If $uv$ is an arc of $G_i$ then $v \neq r$ since $G'_i$ does not contain any arcs entering $r$. Since $d_i[v] \leq d_i[u] + \ell[uv]$, we have $\ell_{\phi_i}[uv] = \phi_i[u] + \ell[uv] + \phi_i[v] \geq d_i[u] + \ell[uv] - d_i[v] \geq 0$. Otherwise, $u = r$ and $v$ is a boundary node, so

$$\ell_{\phi_i}[rv] = \phi_i[r] + B[v] - \phi_i[v] = p_i - (d_i[v] - B[v]) \geq 0$$

by definition of $\phi_i$.

Computing the auxiliary graphs $G'_i$ and the price functions $\phi_i$ can be easily done in linear time. Therefore, the time required for this stage is dominated by the $O(n \log n)$ running time of Dijkstra’s algorithm. We note that one may use the algorithm of Henzinger et al. [1997] instead of Dijkstra to obtain a linear running time for this stage. This however does not change the overall running time of our algorithm.

**6. CORRECTNESS AND ANALYSIS**

We will show that at each stage of our algorithm, the necessary information has been correctly computed and stored. The recursive call in Step 4 computes and stores the from-$r$ distances in $G_i$. The conditions for
applying Klein’s algorithm in Step 5 hold since all boundary nodes lie on the boundary of a single face of \(G_i\) and since the from-\(r\) distances in \(G_i\) constitute a feasible price function for \(G_i\) (see also the discussion at the end of Section 2.5). The correctness of the single-source inter-part boundary distances stage in Step 6 and of the single-source inter-part distances stage in Step 7 was proved in Sections 4 and 5. Thus, the \(r\)-to-\(v\) distances in \(G\) for all nodes \(v\) of \(G\) are stored in \(d_0^r\) for \(v \in G_0\) and in \(d_1^r\) for \(v \in G_1\). Note that \(d_0^r\) and \(d_1^r\) agree on distances from \(r\) to boundary nodes. Therefore, the price function \(\phi\) defined in Step 8 is feasible for \(G\), so the conditions to run Dijkstra’s algorithm in Step 9 hold, and the from-\(s\) distances in \(G\) are correctly computed. We have thus established the correctness of our algorithm.

To bound the running time of the algorithm we bound the time it takes to complete one recursive call to shortest-paths. Let \(|G|\) denote the number of nodes in the input graph \(G\), and let \(|G_i|\) denote the number of nodes in each of its subgraphs. Computing the intra-subgraph boundary-to-boundary distances using Klein’s algorithm takes \(O(|G_i| \log |G_i|)\) for each of the two subgraphs, which is in \(O(|G| \log |G|)\). Computing the single-source distances in \(G\) to the boundary nodes is done in \(O(|G| \alpha(|G|))\), as we explain in Section 4. The extension to all nodes of \(G\) is again done in \(O(|G_i| \log |G_i|)\) for each subgraph. Distances from the given source are computed in an additional \(O(|G| \log |G|)\) time. Thus the total running time of one invocation is \(O(|G| \log |G|)\). Therefore the running time of the entire algorithm is given by

\[
T(|G|) = T(|G_0|) + T(|G_1|) + O(|G| \log |G|)
\]

Here we used the properties of the separator, namely that \(|G_i| \leq 2|G|/3\) for \(i = 0, 1\), and that \(|G_0| + |G_1| = |G| + O(\sqrt{|G|})\). The formal proof of this recurrence is given in the following lemma.

**Lemma 6.1.** Let \(T(n)\) satisfy the recurrence \(T(n) = T(n_1) + T(n_2) + O(n \log n)\), where \(n \leq n_1 + n_2 \leq n + 4\sqrt{n}\) and \(n_1, n_2 \leq \frac{2n}{3}\). Then \(T(n) = O(n \log^2 n)\).

**Proof.** We show by induction that for any \(n \geq N_0\), \(T(n) \leq Cn \log^2 n\) for some constants \(N_0, C\). For a choice of \(N_0\) to be specified below, let \(C_0\) be such that for any \(N_0 \leq n \leq 3N_0\), \(T(n) \leq C_0n \log^2(n)\). Let \(C_1\) be a constant such that \(T(n) \leq T(n_1) + T(n_2) + C_1n \log n\). Let \(C = max\left\{C_0, \frac{C_1}{\log(3/2)}\right\}\). Note that this choice serves as the base of the induction since it satisfies the claim for any \(N_0 \leq n \leq 3N_0\). We assume that the claim holds for all \(n'\) such that \(3N_0 \leq n' < n\) and show it holds for \(n\). Since for \(i = 1, 2\), \(n_i \leq 2n/3\) and \(n_1 + n_2 \geq n\), it follows that \(n_i \geq n/3 > N_0\). Therefore, we may apply the inductive hypothesis to obtain:

\[
T(n) \leq C(n_1 \log^2 n_1 + n_2 \log^2 n_2) + C_1n \log n
\]

\[
\leq C(n_1 + n_2) \log^2(2n/3) + C_1n \log n
\]

\[
\leq C(n + 4\sqrt{n}) \log^2(2/3) + C_1n \log n
\]

\[
\leq Cn \log^2 n + 4C \sqrt{n} \log^2 n - 2C \log(3/2)n \log n
\]

\[
+ C(n + 4\sqrt{n}) \log^2(2/3) + C_1n \log n
\]

It therefore suffices to show that

\[
4C \sqrt{n} \log^2 n - 2C \log(3/2)n \log n + C(n + 4\sqrt{n}) \log^2(2/3) + C_1n \log n \leq 0,
\]

or equivalently that

\[
\left(1 - \frac{C_1}{2C \log(3/2)}\right)n \log n \geq \frac{2}{\log(3/2)} \sqrt{n} \log^2 n + \frac{\log(3/2)}{2} (n + 4\sqrt{n}).
\]

Let \(N_0\) be such that the right hand side is at most \(\frac{1}{2}n \log n\) for all \(n > N_0\). The above inequality holds for every \(n > N_0\) since we chose \(C > \frac{C_1}{\log(3/2)}\), so the coefficient in the left hand side is at least \(\frac{1}{2}\). \(\square\)

We have thus proved that the total running time of our algorithm is \(O(n \log^2 n)\). We turn to the space bound. The space required for one invocation is \(O(|G|)\). Each of the two recursive calls can use the same memory locations one after the other, so the space is given by

\[
S(|G|) = \max\{S(|G_0|), S(|G_1|) + O(|G|)\}
\]

\[
= O(|G|) \text{ because } \max\{|G_0|, |G_1|\} \leq 2|G|/3.
\]
Thus, we can write \( O(d,u) \) and a right-going arc if \( d \neq u \). For \( i < i' < j < j' \), the \( i,j \) path and the \( i',j' \) path must cross at some node \( z \).

For nodes \( x,y \), the \( x\text{-to-}y \) distance is denoted by \( \delta_G(x,y) \). The distances \( \delta_G(s,u_i) \) and \( \delta_G(u_i,t) \) for \( i = 1, \ldots, p+1 \) are computed from \( P \) in \( O(p) \) time, and stored in a table.

Let \( P = (u_1, u_2, \ldots, u_{p+1}) \) be the shortest path from \( s = u_1 \) to \( t = u_{p+1} \) in the graph \( G \). Consider the replacement \( s\text{-to-}t \) path \( Q \) that avoids the arc \( e \) in \( P \). \( Q \) can be decomposed as \( Q_1 Q_2 Q_3 \) where \( Q_1 \) is a prefix of \( P \), \( Q_3 \) is a suffix of \( P \), and \( Q_2 \) is a subpath from some \( u_i \) to some \( u_j \) that avoids any other vertex in \( P \). If in a clockwise traversal of the arcs incident to some node \( u_i \) of \( P \), starting from the arc \( (u_{i-1}, u_i) \) we encounter an arc \( e \) before we encounter the arc \( (u_i, u_{i+1}) \), then we say that \( e \) is to the right of \( P \). Otherwise, \( e \) is to the left of \( P \). The first arc of \( Q_2 \) can be left or right of \( P \) and the last arc of \( Q_2 \) can be left or right of \( P \). In all four cases \( Q_2 \) never crosses \( P \) (see Fig. 7 and Fig. 8).

For nodes \( x,y \), the \( x\text{-to-}y \) distance is denoted by \( \delta_G(x,y) \). The distances \( \delta_G(s,u_i) \) and \( \delta_G(u_i,t) \) for \( i = 1, \ldots, p+1 \) are computed from \( P \) in \( O(p) \) time, and stored in a table.

The \( p \times p \) matrix \( \text{len}_{d,d'}(i,j) \) is defined in [Emek et al. 2008] as follows: for any \( 1 \leq i \leq p \) and \( 1 \leq j \leq p \), let \( \text{len}_{d,d'}(i,j) \) be the length of the shortest \( s\text{-to-}t \) path of the form \( Q_1 Q_2 Q_3 \) described above where \( Q_2 \) starts at \( u_i \) via a left-going arc if \( d = L \) or a right-going arc if \( d = R \), and \( Q_2 \) ends at \( u_{j+1} \) via a left-going arc if \( d' = L \) and a right-going arc if \( d' = R \). The length of \( Q_2 \) is denoted \( \text{PAD-query}_{G,d,d'}(i,j) \). It can be computed in \( O(\log n) \) time by a single query to a data structure that Emek et al. call PADO (Path Avoiding Distance Oracle). Thus, we can write

\[
\text{len}_{d,d'}(i,j) = \delta_G(s,u_i) + \text{PAD-query}_{G,d,d'}(i,j) + \delta_G(u_{j+1},t),
\]

and query any entry of \( \text{len}_{d,d'} \) in \( O(\log n) \) time.

The \( O(n \log^3 n) \) time-complexity of Emek et al. arises from the recursive calls to the \textbf{District} procedure.
paths problem thus reduces to computing the minimum element in range $\hat{s}$ among these four rectangular portions corresponding to the four possible $i,...,p$.

With this definition the length of the replacement $i,...,p$ is easy to explain the use of SMAWK.

We next give an alternative description of their algorithm. This new description is slightly simpler and makes it easy to explain the use of SMAWK.

Let range($i$) denote the rectangular portion of the matrix $\hat{\text{len}}_{d,d'}$ defined by rows $1,...,i$ and columns $i,...,p$. With this definition the length of the replacement $s$-to-$t$ path that avoids the edge $(u_i, u_{i+1})$ is equal to the minimal element in range($i$). Since $d \in \{L, R\}$ and $d' \in \{L, R\}$, we need to take the minimum among these four rectangular portions corresponding to the four possible $\hat{\text{len}}$ matrices. The replacement-paths problem thus reduces to computing the minimum element in range($i$) for every $i = 1,2,...,p$, and every $d,d' \in \{L, R\}$.

Given some $1 \leq a < b \leq p$ and some $d,d' \in \{L, R\}$, District($a,b$) computes the row and column-minima of the rectangular portion of the matrix $\hat{\text{len}}_{d,d'}$ defined by rows $a$ to $\lfloor (a+b)/2 \rfloor$ and columns $\lfloor (a+b)/2 \rfloor$ to $b$. Initially, District is called with $a = 1$ and $b = p$. This, in particular, computes the minimum of range($\lfloor p/2 \rfloor$). Then, District($a,\lfloor (a+b)/2 \rfloor - 1$) and District($\lfloor (a+b)/2 \rfloor + 1, b$) are called recursively. Notice that the previous call to District($a,b$), together with the current call to District($a,\lfloor (a+b)/2 \rfloor - 1$) suffice for computing all row and column-minima of range($\lfloor p/4 \rfloor$) (and hence also the global minimum of range($\lfloor p/4 \rfloor$)), as illustrated in Fig. 9. Similarly, District($a,b$), together with District($\lfloor (a+b)/2 \rfloor + 1, b$) suffice for computing all row and column-minima of range($\lfloor 3p/4 \rfloor$). The recursion stops when $b - a \leq 1$. Therefore, the depth of the recursion for District($1,p$) is $O(\log p)$, and it computes the minimum of range($i$) for all $1 \leq i \leq p$.

Emek et al. show how to compute District($a,b$) in $O((b-a) \log^2 (b-a) \log n)$ time, leading to a total of $O(n \log^3 n)$ time for computing District($1,p$). They use a divide and conquer technique to compute the row and column-minima in each of the rectangular areas encountered along the computation. Our contribution is in showing that instead of divide-and-conquer one can use SMAWK to find those minima. This enables computing District($a,b$) in $O((b-a) \log(b-a) \log n)$ time, which leads to a total of $O(n \log^2 n)$ time for District($1,p$), as shown by the following lemmas.

**Lemma 7.1.** The upper triangle of $\hat{\text{len}}_{d,d'}$ satisfies a Monge property.

**Proof.** Note that adding $\delta_G(s, u_i)$ to all of the elements in the $i^{th}$ row or $\delta_G(u_{j+1}, t)$ to all elements in the $j^{th}$ column preserves the Monge property. Therefore, it suffices to show that the upper triangle of PAD-query$_{G,d,d'}$ satisfies a Monge property.

When $d = d'$, the proof is essentially the same as that of Lemma 4.4 because the $Q_2$ paths have the same
crossing property as the paths in Lemma 4.4. This is illustrated in Fig. 7. We thus establish that the convex Monge property holds.

When \( d \neq d' \), Lemma 4.4 applies but with the convex Monge property replaced with the concave Monge property. To see this, consider the crossing paths in Fig. 8. In contrast to Fig. 7, this time the crossing paths are \( i \)-to-\( j' \) and \( i' \)-to-\( j \). □

**Lemma 7.2.** Procedure \( \text{District}(a, b) \) can be computed in \( O((b-a) \log(b-a) \log n) \) time.

**Proof.** Recall that, for every pair \( d, d' \), \( \text{District}(a, b) \) first computes the row and column-minima of the rectangular submatrix of \( \ell_{n, d} d' \) defined by rows \( a \) to \( [(a+b)/2] \) and columns \( [(a+b)/2] \) to \( b \). By Lemma 7.1, this entire submatrix has a Monge property. In the case of the convex Monge property, we can use SMAWK to find all row and column-minima of the submatrix. In the case of the concave Monge property, we cannot directly apply SMAWK. By negating all the elements we get a convex Monge matrix but we are now looking for its row and column maxima. As discussed in Section 2.3, SMAWK can be used to find row and column-maxima of a convex Monge matrix. Thus, in both cases, we find the row and column-minima of the submatrix by querying only \( O(b-a) \) entries each in \( O(\log n) \) time for a total of \( O((b-a) \log n) \) time. Therefore, \( T(a, b) \), the time it takes to compute \( \text{District}(a, b) \) is given by

\[
T(a, b) = T(a, (a+b)/2) + T((a+b)/2, b) + O((b-a) \log n) = O((b-a) \log(b-a) \log n). \quad \Box
\]

**REFERENCES**


