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

Philip Klein[1], Shay Mozes[1] and Oren Weimann[2]

[1] Department of Computer Science, Brown University, Providence RI 02912-1910, USA
{klein,shay}@cs.brown.edu
Supported by NSF Grant CCF-0635089. Work done while Klein was visiting MIT.
[2] Massachusetts Institute of Technology, Cambridge, MA 02139, USA. oweimann@mit.edu

Abstract. 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. The best previously known algorithm requires $O(n \log^3 n)$ time and $O(n \log n)$ space.

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,[3] 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 [7] takes $O(\sqrt{n}m\log(nN))$. For integer lengths exceeding $-N$, the algorithm of Goldberg [8] takes $O(\sqrt{n}m\log N)$ time. For non-negative lengths, the problem is easier and can be solved using Dijkstra's algorithm in $O((n+m)\lg n)$ time if elementary data structures are used [12], and in $O(n\log n + m)$ time when implemented with Fibonacci heaps [6].

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 [15], who gave an $O(n^{3/2})$ algorithm. Henzinger et al. [9] 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 [5] gave an algorithm requiring $O(n\log^3 n)$ time and $O(n\log n)$ space. Our result is as follows:

Theorem 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 [18] show that, by using planar duality, the following problem can be reduced to shortest paths in a planar directed graph:

> *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 [3] and by Jermyn and Ishikawa [11, 10], and a stereo matching technique due to Veksler [20], involve finding negative-length cycles in graphs that are essentially planar. Thus our algorithm can be used to implement these techniques.

[3] Algorithms for this problem can also be used to detect negative cycles.

Summary of the Algorithm

Like the other planarity-exploiting algorithms for this problem, our algorithm uses planar separators [16, 17]. 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 the 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.

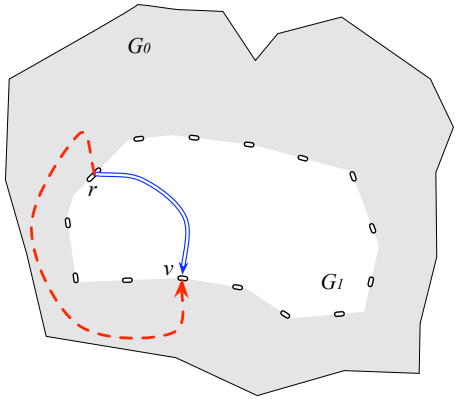

Fig. 1. A graph G and a decomposition using a Jordan curve into an external subgraph G_0 (in gray) and an internal subgraph G_1 (in white). Only boundary nodes are shown. r and v are boundary nodes. The double-lined blue path is an r-to-v shortest path in G_1. The dashed red path is an r-to-v shortest path in G_0.

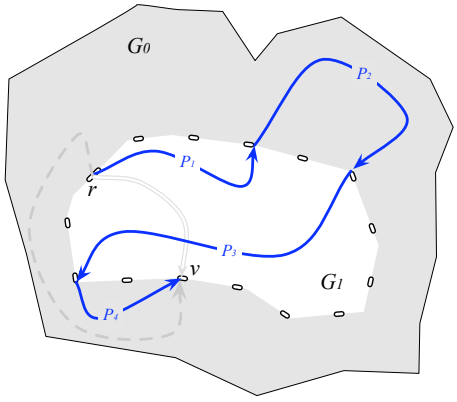

Fig. 2. The solid blue path is an r-to-v shortest path in G. It can be decomposed into four subpaths. The subpaths P_1 and P_3 are shortest paths in G_1 between boundary nodes. The subpaths P_2 and P_4 are shortest paths in G_0 between boundary nodes. The r-to-v shortest paths in G_0 and G_1 can be seen in gray in the background.

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.

Intra-part boundary-distances: For each graph G_i we use a method due to Klein [14] 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 [19] property (discussed later), each iteration can be implemented by two executions of an algorithm due to Klawe and Kleitman [13] 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 [15], 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. [9] achieved an improvement by using a multi-part decomposition based on planar separators. Fakcharoenphol and Rao's algorithm [5] 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 use this idea in our algorithm as well, although we exploit it using a different approach. 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 [14] and a basic priority queue for implementing Dijkstra's algorithm.[4]

The Replacement-Paths Problem

We note that the algorithm of Klawe and Kleitman [13] that we use in our algorithm is useful in addressing 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. [4] 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 (defined formally in the appendix) which has the Monge property. By using [13], we obtain an $O(n \log^2 n\alpha(n))$-time algorithm. Details appear in the appendix.

Theorem 2. *There is an $O(n \log^2 n\alpha(n))$-time algorithm for solving the replacement-paths problem in a directed planar graph.*

[4] In planar graphs there is no need for Fibonacci heaps as $m = O(n)$.

2 Preliminaries

2.1 Jordan Separators for Planar Graphs

Miller [17] 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 $2\sqrt{2}\sqrt{n}$ 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 $2\sqrt{2}\sqrt{n}$ boundary nodes, add artificial edges to triangulate the graph and make it two-connected, then apply Miller's algorithm.

The *internal part of G with respect to C* is the 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.

In the internal part of G with respect to C, every edge is enclosed by C, so C is contained by some face. Since every boundary node is intersected by C, it follows that all boundary nodes lie on the boundary of a single face of the internal part of G. Similarly, in the external part of G with respect to C, all boundary nodes lie on the boundary of a single face.

2.2 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 $M_{ij} \leq M_{ij'}$, we have $M_{i'j} \leq M_{i'j'}$. Totally monotone matrices were introduced by Aggarwal et al. in [2], 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 a clever 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 *Monge* if for every i, i', j, j' such that $i < i'$, $j < j'$, we have $M_{ij} + M_{i'j'} \geq M_{ij'} + M_{i'j}$. It is immediate that if M is Monge then it is totally monotone. It is also easy to see that the matrix obtained by transposing M and then reversing the order of the columns is also totally monotone. Therefore finding the column minima and maxima of a Monge matrix is as easy as finding its row maxima.

In [13] Klawe and Kleitman define a *falling staircase matrix* 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. for $i < k, j < l \leq f(i)$, and $M_{ij} \leq M_{il}$, 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 [1] 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 [13] 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. Again, it is not hard to see that both these algorithms can be used to find the column minima, by transposing and then reversing the order of the columns, as above.

2.3 Price Functions and Reduced Lengths

For a directed graph G with arc-lengths $\ell(\cdot)$, a *price function* is a function ϕ from the nodes of G to the reals. For an arc uv, the *reduced length with respect to* ϕ 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 $\ell_\phi(\cdot)$ iff 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 ϕ 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 with* ϕ.

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 edge uv, $\phi(v) \leq \phi(u) + \ell(uv)$, so $\ell_\phi(uv) \geq 0$.

2.4 Multiple-Source Shortest Paths: Computing Boundary-to-Boundary Distances

Klein [14] 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 $O(n \log n)$ 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.

For graphs with negative arc-lengths, given a node r and the distances $d(v)$ from r to all nodes v in G, we can use d to compute the reduced lengths and then run Klein's algorithm on these non-negative lengths. Given d, running Klein's algorithm with these reduced lengths will be called *running the multiple-source shortest paths algorithm with input* d.

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 2.4. 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

procedure **shortest-paths**(G, s)
input: a directed embedded planar graph G with arc-lengths, and a node s of G
output: a table d giving distances in G from s to all nodes of G

0 if G has ≤ 2 nodes, the problem is trivial; return the result
1 find a Jordan separator C of G with $O(\sqrt{n})$ boundary nodes
2 let G_0,G_1 be the external and internal parts of G with respect to C
3 let r be a boundary node

Recursive call 4 for $i = 0, 1$: let $d_i =$ shortest-paths(G_i, r)

intra-part boundary distances 5 for $i = 0, 1$: use d_i as input to the multiple-source shortest-path algorithm to compute a table δ_i such that $\delta_i[u, v]$ is the u-to-v distance in G_i for every pair u, v of boundary nodes

single-source inter-part boundary distances 6 use δ_0 and δ_1 to compute a table B such that $B[v]$ is the r-to-v distance in G for every boundary node v

single-source inter-part distances 7 for $i = 0, 1$: use table B and Dijkstra's algorithm to compute a table d'_i such that $d'_i[v]$ is the r-to-v distance in G for every node v of G_i

rerooting single-source distances 8 define a price function ϕ for G such that $\phi[v]$ is the r-to-v distance in G:

$$\phi[v] = \begin{cases} d'_0[v] & \text{if } v \text{ belongs to } G_0 \\ d'_1[v] & \text{otherwise} \end{cases}$$

9 use Dijkstra's algorithm with price function ϕ to compute a table d such that $d[v]$ is the s-to-v distance in G for every node v of G

10 return d

Fig. 3. The shortest-path algorithm

the temporary source node. These distances constitute a feasible price function, as described in Section 2.3, 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 δ_0 and δ_1, the all-pairs distances in G_0 and in G_1 between nodes in V_c which were computed in the previous stage. The following structural lemma stands in the core of the computation. The same lemma has been implicitly used by previous planarity-exploiting algorithms. The proof is given in the appendix.

Lemma 1. *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 $P = P_1P_2P_3\ldots$, where the endpoints of each subpath P_i are boundary nodes, and P_i is a shortest path in $G_{i \bmod 2}$.*

Lemma 1 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.

1: $e_0[v] = \infty$ for all $v \in V_c$
2: $e_0[r] = 0$
3: for $j = 1, 2, 3, \ldots, |V_c|$
4: $\quad e_j[v] = \left\{ \begin{matrix} \min_{w \in V_c}\{e_{j-1}[w] + \delta_1[w,v]\}, \text{if } j \text{ is odd} \\ \min_{w \in V_c}\{e_{j-1}[w] + \delta_0[w,v]\}, \text{if } j \text{ is even} \end{matrix} \right\}, \forall v \in V_c$
5: $B[v] \leftarrow e_{|V_c|}[v]$ for all $v \in V_c$

Fig. 4. Pseudocode for the single-source inter-part boundary distances stage for calculating shortest-path distances in G from r to all nodes in V_c using just δ_0 and δ_1.

Lemma 2. *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_i is a shortest path in $G_{i \bmod 2}$.*

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 for some boundary node w. Hence, by the inductive hypothesis, when e_j is updated in Line 4, $e_{j-1}[w]$ already stores the length of P'. Thus $e_j[v]$ is updated in line 4 to be at most $e_{j-1}[w] + \delta_{j \bmod 2}[w,v]$. Since, by definition, $\delta_{j \bmod 2}[w,v]$ is the length of the shortest path in $G_{j \bmod 2}$ from w to v, it follows that $e_{j+1}[v]$ is at most the length of P. For the opposite direction, since for any boundary node w, $e_j[w]$ is the length of some path that can be decomposed into at most $j-1$ subpaths as above, $e_j[v]$ is updated in Line 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, $e_j[v]$ is at least the length of P. □

From Lemma 1 and Lemma 2, it immediately follows that the table $e_{|V_c|}$ stores the from-r shortest path distances in G, so the assignment in Line 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 Line 4 in $O(|V_c|\alpha(|V_c|))$ time. Let $i = j \bmod 2$, 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 $A_{k\ell} = d_{i-1}(v_k) + \delta_1(v_k, v_\ell)$. Note that computing all minima in Line 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 $\{A_{k\ell} : k \leq \ell\}$ of A. Similarly, the lower triangle of A consists of all the elements on or below the main diagonal of A.

Lemma 3. *For any four indices k, k', ℓ, ℓ' 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 $1 \le k \le k' \le \ell \le \ell' \le |V_c|$ or $1 \le \ell \le \ell' \le k \le k' \le |V_c|$), the Monge property holds:*

$$A_{k\ell} + A_{k'\ell'} \ge A_{k\ell'} + A_{k'\ell}.$$

Proof. Consider the case $1 \le k \le k' \le \ell \le \ell' \le |V_c|$, as in Fig. 5. Since G_i is planar, any pair of paths in G_i from k to ℓ and from k' to ℓ' must cross at some node w of G_i. Let $b_k = d_{j-1}(v_k)$ and let $b_{k'} = d_{j-1}(v_{k'})$. Let $\Delta(u, v)$ denote the u-to-v distance in G_i for any nodes u, v of G_i. Note that $\Delta(u, v) = \delta_i(u, v)$ for $u, v \in V_c$. We have

$$\begin{aligned} A_{k,\ell} + A_{k',\ell'} &= (b_k + \Delta(v_k, w) + \Delta(w, v_\ell)) + (b_{k'} + \Delta(v_{k'}, w) + \Delta(w, v_{\ell'})) \\ &= (b_k + \Delta(v_k, w) + \Delta(w, v_{\ell'})) + (b_{k'} + \Delta(v_{k'}, w) + \Delta(w, v_\ell)) \\ &\ge (b_k + \Delta(v_k, v_{\ell'})) + (b_{k'} + \Delta(v_{k'}, v_\ell)) \\ &= (b_k + \delta_i(v_k, v_{\ell'})) + (b_{k'} + \delta_i(v_{k'}, v_\ell)) = A_{k,\ell'} + A_{k',\ell} \end{aligned}$$

The second case ($1 \le \ell \le \ell' \le k \le k' \le |V_c|$) is similar. □

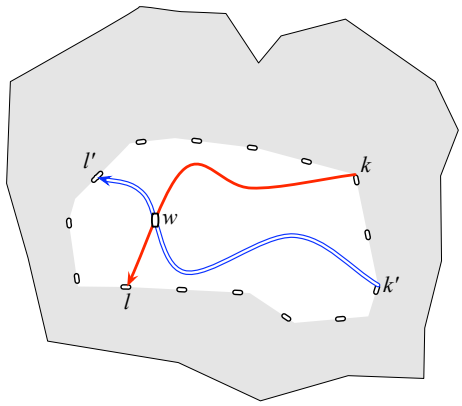

Fig. 5. Nodes $k < k' < l < l'$ in clockwise order on the boundary nodes. Paths from k to l and from k' to l' must cross at some node w. This is true both in the internal and the external subgraphs of G

Lemma 4. *A single iteration of the dynamic program can be computed in $O(|V_c|\alpha(|V_c|))$ time.*

Proof. We need to show how to find the column minima of the matrix A. It follows directly from Lemma 3 that the lower triangle of A is a falling staircase matrix. By [13], its column minima can be computed in $O(|V_c|\alpha(|V_c|))$ time. Another consequence of Lemma 3 is that the column-minima of the upper triangle of A may also be computed using the algorithm in [13]. To see this consider a counterclockwise ordering of the nodes of $|V_c|$ $v'_1, v'_2, \ldots, v'_{|V_c|}$ such that $v'_k = v_{|V_c|+1-k}$. With this order, the upper triangle of A is in fact a falling staircase matrix.

Finally, we obtain A's column minima by comparing the two minima obtained for each column in each of the two triangular portions of A. We thus conclude that A's column minima can be computed in $O(2|V_c| \cdot \alpha(|V_c|) + |V_c|) = O(|V_c| \cdot \alpha(|V_c|))$ time. □

Hence the time it takes to compute the distances between r and all nodes of V_c is $O(|V_c|^2 \cdot \alpha(|V_c|)$. 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. 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'_i. 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 ϕ_i for G'_i and use Dijkstra's algorithm on G'_i with the price function ϕ_i in order to compute these from-r distances.

1: 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
2: let $p_i = \max\{B[u] - d_i[u] \ : \ u \text{ a boundary node}\}$
3: define a price function ϕ_i for G'_i:

$$\phi_i[v] = \begin{cases} p_i & \text{if } v = r \\ d_i[v] & \text{otherwise} \end{cases}$$

4: use Dijkstra's algorithm with price function ϕ_i to compute a table d'_i such that $d'_i[v]$ is the r-to-v distance in G'_i for every node v of G'_i

Fig. 6. Pseudocode for the single-source intra-part distances stage for computing the shortest path distances from r to all nodes.

The following two lemmas motivate the definition of G'_i and show that it captures the true from-r distances in G. The proof of Lemma 5 is given in the appendix.

Lemma 5. *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 visits only nodes of G_i.*

Lemma 6. *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. To prove the opposite direction, consider the distance from r to v in G. Let P_1, P_2, u be as in Lemma 5. 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 path in G'_i that consists of the new arc ru followed by P_2 has the same length as the path 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 7. *ϕ_i is a feasible price function for G'_i.*

Proof. Let uv be an arc of G'_i. If uv is an arc of G_i, then $d_i[v] \leq d_i[u] + \ell[uv]$, so $\ell_{\phi_i}[uv] \geq 0$. Otherwise, $u = r$ and

$$\begin{aligned}\ell_{\phi_i}[rv] &= \phi_i[r] + B[v] - \phi_i[v] \\ &= p_i + B[v] - d_i[u] \qquad \text{by definition of } \phi_i \\ &\geq 0 \qquad\qquad\qquad\quad \text{by definition of } p_i\end{aligned}$$

□

Computing the auxiliary graphs G'_i and the price functions ϕ_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. [9] 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 Line 4 computes and stores the from-r distances in G_i. The conditions for applying Klein's algorithm in Line 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. The correctness of the single-source inter-part boundary distances stage in Line 6 and of the single-source inter-part distances stage in Line 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 for $v \in G_0$ and in d'_1 for $v \in G_1$. Note that d'_0 and d'_1 agree on distances from r to boundary nodes. Therefore, the price function ϕ defined in Line 8 is feasible for G, so the conditions to run Dijkstra's algorithm in Line 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

$$\begin{aligned}T(|G|) &= T(|G_0|) + T(|G_1|) + O(|G| \log |G|) \\ &= O(|G| \log^2 |G|).\end{aligned}$$

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 complete proof of this recurrence is given in the appendix. Thus, 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 $O(|G|)$. The same storage can be used for the two recursive calls, so the space is given by

$$\begin{aligned}S(|G|) &= \max\{S(|G_0|), S(|G_1|\} + O(|G|) \\ &= O(|G|) \quad \text{because} \max\{|G_0|, |G_1|\} \leq 2|G|/3\end{aligned}$$

We have proved Theorem 1.

References

1. A. Aggarwal and M. M. Klawe. Applications of generalized matrix searching to geometric algorithms. *Discrete Applied Mathematics*, 27(1-2):3–23, 1990.
2. A. Aggarwal, M. M. Klawe, S. Moran, P. Shor, and R. Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2(1):195–208, 1987.
3. I. Cox, S. Rao, and Y. Zhong. Ratio regions: A technique for image segmentation. *International Conference on Pattern Recognition*, 02:557, 1996.
4. Y. Emek, D. Peleg, and L. Roditty. A near-linear time algorithm for computing replacement paths in planar directed graphs. In *SODA '08: Proceedings of the Nineteenth Annual ACM-SIAM symposium on Discrete Algorithms*, pages 428–435, Philadelphia, PA, USA, 2008. Society for Industrial and Applied Mathematics.
5. J. Fakcharoenphol and S. Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. *J. Comput. Syst. Sci.*, 72(5):868–889, 2006.
6. M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.
7. H. N. Gabow and R. E. Tarjan. Faster scaling algorithms for network problems. *SIAM Journal on Computing*, 18(5):1013–1036, 1989.
8. A. V. Goldberg. Scaling algorithms for the shortest paths problem. *SIAM J. Comput.*, 24(3):494–504, 1995.
9. M. R. Henzinger, P. N. Klein, S. Rao, and S. Subramanian. Faster shortest-path algorithms for planar graphs. *Journal of Computer and System Sciences*, 55(1):3–23, 1997.
10. H. Ishikawa and I. Jermyn. Region extraction from multiple images. In *8th IEEE International Conference on Computer Vision*, pages 509–516, Los Alamitos, CA, USA, 2001. IEEE Computer Society.
11. I. H. Jermyn and H. Ishikawa. Globally optimal regions and boundaries as minimum ratio weight cycles. *IEEE Trans. Pattern Anal. Mach. Intell.*, 23(10):1075–1088, 2001.
12. D. B. Johnson. Efficient algorithms for shortest paths in sparse graphs. *Journal of the ACM*, 24:1–13, 1977.
13. M. M. Klawe and D. J. Kleitman. An almost linear time algorithm for generalized matrix searching. *SIAM Journal On Discrete Math*, 3(1):81–97, 1990.
14. P. N. Klein. Multiple-source shortest paths in planar graphs. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 146–155, 2005.
15. R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized nested dissection. *SIAM Journal on Numerical Analysis*, 16:346–358, 1979.
16. R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.
17. G. L. Miller. Finding small simple cycle separators for 2-connected planar graphs. *Journal of Computer and System Sciences*, 32(3):265–279, 1986.
18. G. L. Miller and J. Naor. Flow in planar graphs with multiple sources and sinks. *SIAM Journal on Computing*, 24(5):1002–1017, 1995.
19. G. Monge. Mémoire sur la théorie des déblais et ramblais. *Mém. Math. Phys. Acad. Roy. Sci. Paris*, pages 666–704, 1781.
20. O. Veksler. Stereo correspondence with compact windows via minimum ratio cycle. *IEEE Trans. Pattern Anal. Mach. Intell.*, 24(12):1654–1660, 2002.

A Appendix

A.1 Proof of Lemma 1

Consider a decomposition of $P = P_1P_2P_3\ldots$ into maximal subpaths such that the subpath P_i consists of nodes of $G_{i \bmod 2}$. 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 $G_{i \bmod 2}$ 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. □

A.2 Proof of Lemma 5

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 is a path in G_i. □

A.3 Running-Time Recurrence

Lemma 8. *Let $T(n)$ satisfy the recurrence $T(n) = T(n_1) + T(n_2) + C_1(n \log n)$, where $n_1 + n_2 \leq n + 4\sqrt{n}$ and $n_i \leq \frac{2n}{3}$. 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 to be specified later. The base case holds since we may choose C so that $T(N_0) \leq CN_0 \log^2(N_0)$. Assume the claim holds for all n' such that $N_0 \leq n' < n$. Then:

$$\begin{aligned}
T(n) &\leq C(n_1 \log^2 n_1 + n_2 \log^2 n_2) + C_1 n \log n \\
&\leq C(n_1 + n_2) \log^2(2n/3) + C_1 n \log n \\
&\leq C(n + 4\sqrt{n}) \log^2(2n/3) + C_1 n \log n \\
&= C(n + 4\sqrt{n})(\log(2/3) + \log n)^2 + C_1 n \log n \\
&\leq Cn \log^2 n + 4C\sqrt{n} \log^2 n - 2C \log(3/2) n \log n + (n + 4\sqrt{n}) \log^2(2/3) + C_1 n \log n
\end{aligned}$$

It therefore suffices to show that

$$4C\sqrt{n} \log^2 n - 2C \log(3/2) n \log n + (n + 4\sqrt{n}) \log^2(2/3) + C_1 n \log n \leq 0.$$

or equivalently that

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

We therefore choose $C > \frac{C_1}{2 \log(3/2)}$, so that the above inequality holds since the coefficient in the left hand side is positive, and since $\frac{2}{\log(3/2)} \sqrt{n} \log^2 n + \frac{\log(2/3)}{2C}(n + 4\sqrt{n})$ is in $o(n \log n)$. □

A.4 Proof Sketch of Theorem 2

The $O(n \log^3 n)$ time-complexity of Emek et al. [4] origins in $O(\lg n)$ recursive calls to the `District` procedure. This procedure computes in $O(|G| \log^2 |G|)$ time all row minima of the matrix $\widehat{\text{len}}_{d,d'}$ associated with G. We next describe this matrix and show that its row minima can actually be found in $O(|G|\alpha(|G|) \lg |G|)$ time.

Let $P = (u_0, u_1, \ldots, u_{p+1})$ be the shortest path from $s = u_0$ to $t = u_{p+1}$ in the graph G. Consider the replacement s-to-t path Q that avoids the edge e in P. Q can be decomposed into $Q_1 Q_2 Q_3$ 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. The first edge of Q_2 can be left or right of P and the last edge of Q_2 can be left or right of P (see [4] for a common formal definition of the left-of and right-of relations). In all four cases Q_2 never crosses P (see Fig. 7 and Fig. 8).

The matrix $\widehat{\text{len}}_{d,d'}$ is defined in [4] as the length of the shortest s-to-t path $Q = Q_1 Q_2 Q_3$ with Q_2 as follows: Q_2 starts at u_i, its first edge is to the left of P if $d = L$ and to its right if $d = R$. Similarly, its last edge is to the left of P if $d' = L$ and to its right if $d' = R$. The last vertex

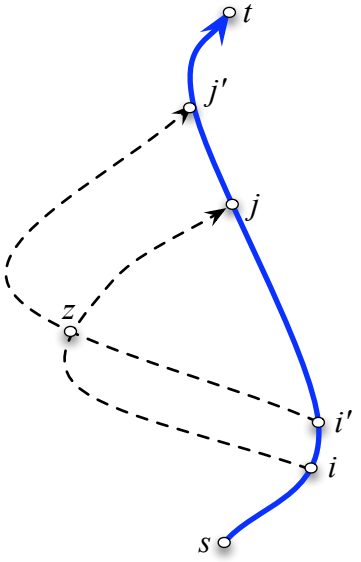

Fig. 7. The s-t shortest path P is shown in solid blue. Paths of type Q_2 (dashed black) do not cross P. Two LL paths (i.e., leaving and entering P from the left) are shown. For $i < i' < j < j'$, the ij path and the $i'j'$ path must cross at some node z.

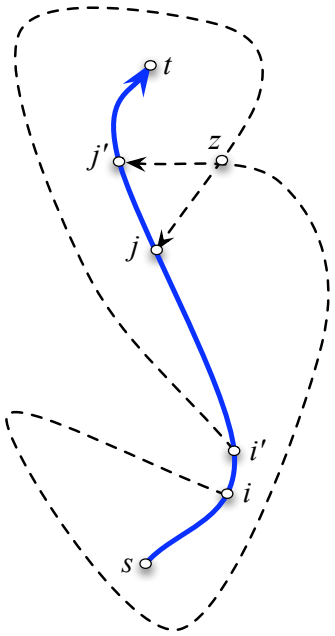

Fig. 8. The s-t shortest path P is shown in solid blue. Paths of type Q_2 (dashed black) do not cross P. Two LR paths (i.e., leaving P from the left and entering P from the right) are shown. For $i < i' < j < j'$, the ij' path and the $i'j$ path must cross at some node z.

of Q_2 is u_j. The lengths of Q_1 and Q_3 are denoted $\delta_G(s, u_i)$ and $\delta_G(u_j, t)$ and can be extracted in $O(1)$-time assuming a preprocessing Dijkstra step computed P. The length of Q_2 is denoted $\mathsf{PAD\text{-}query}_{G,d,d'}(i, j)$ and it can be computed in $O(\lg |G|)$ time by a single query to a data structure that Emek et al. call PADO (Path Avoiding Distance Oracle). Thus, we can write

$$\widehat{\text{len}}_{d,d'}(i, j) = \delta_G(s, u_i) + \mathsf{PAD\text{-}query}_{G,d,d'}(i, j) + \delta_G(u_j, t),$$

and query each entry of $\widehat{\text{len}}_{d,d'}$ in $O(\lg |G|)$ time.

Lemma 9. *The row minima of $\widehat{\text{len}}_{d,d'}$ can be computed in $O(|G|\alpha(|G|) \lg |G|)$ time.*

Proof. Recall that a *falling staircase matrix* is a lower triangular fragment of a totally monotone matrix. We first consider the case where $d = d'$ and show that the matrix $\mathsf{PAD\text{-}query}_{G,d,d'}$ is a falling staircase. Notice that $\widehat{\text{len}}_{d,d'}$ is obtained from $\mathsf{PAD\text{-}query}_{G,d,d'}$ by adding the same value $\delta_G(s, u_i)$ to all elements in the i'th row and then adding the same value $\delta_G(u_j, t)$ to all elements in the j'th column. It is not hard to verify that adding the same value to an entire row or column of a falling staircase matrix gives another falling staircase matrix.

When $d = d'$, the proof of $\mathsf{PAD\text{-}query}_{G,d,d'}$ being a falling staircase is essentially the same as the proof of Lemma 3 since the Q_2 paths have the same crossing property as the paths in Lemma 3. This is illustrated in Fig. 7. Since $\widehat{\text{len}}_{d,d'}$ is a falling staircase its row minima can be computed using Klawe and Kleitman's algorithm by querying only $O(|G|\alpha(|G|))$ matrix entries (see Section 2.2). The total running time is $O(|G|\alpha(|G|) \lg |G|)$ as each such query can be done in $O(\lg |G|)$.

We now turn to to the case of $d \neq d'$. In this case, Lemma 3 holds but the Monge condition is with a $\leq$ rather than a $\geq$. To see this, consider the crossing paths in Fig. 8. As opposed to Fig. 7,

this time the crossing paths are i-to-j' and i'-to-j. By negating all the elements of $\mathsf{PAD\text{-}query}_{G,d,d'}$ we get a falling staircase matrix with the original $\geq$ Monge condition and we are now looking for its row maxima. As we explained in Section 2.2, finding the row maxima in a falling staircase matrix can be easily done using SMAWK in $O(|G|)$ time after replacing the blanks with sufficiently small numbers so that the resulting matrix is totally monotone. The total running time for $d \neq d'$ is thus $O(|G| \lg |G|)$. □