Shortest Paths in Planar Graphs with Real Lengths in $O(n \log^2 n / \log\log n)$ Time

Shay Mozes*[1] and Christian Wulff-Nilsen[2]

[1] Department of Computer Science, Brown University, Providence, RI 02912, USA
shay@cs.brown.edu
[2] Department of Computer Science, University of Copenhagen, DK-2100, Copenhagen, Denmark. koolooz@diku.dk

Abstract. Given an n-vertex planar directed graph with real edge lengths and with no negative cycles, we show how to compute single-source shortest path distances in the graph in $O(n \log^2 n / \log\log n)$ time with $O(n)$ space. This improves on a recent $O(n \log^2 n)$ time bound by Klein et al.

1 Introduction

Computing shortest paths in graphs is one of the most fundamental problems in combinatorial optimization. The Bellman-Ford algorithm and Dijkstra's algorithm are classical algorithms that find distances from a given vertex to all other vertices in the graph. The Bellman-Ford algorithm works for general graphs and runs in $O(mn)$ time where m resp. n is the number of edges resp. vertices of the graph. Dijkstra's algorithm runs in $O(m + n \log n)$ time when implemented with Fibonacci heaps but it only works for graphs with non-negative edge lengths.

We are interested in the single-source shortest path (SSSP) problem for planar directed graphs. There is an optimal $O(n)$ time algorithm for SSSP when all edge lengths are non-negative [4]. For planar graphs with arbitrary real edge lengths and with no negative cycles[3], Lipton, Rose, and Tarjan [8] gave an $O(n^{3/2})$ time algorithm. Henzinger, Klein, Rao, and Subramanian [4] obtained a (not strongly) polynomial bound of $\tilde{O}(n^{4/3})$. Later, Fakcharoenphol and Rao [3] showed how to solve the problem in $O(n \log^3 n)$ time and $O(n \log n)$ space. Recently, Klein, Mozes, and Weimann [7] presented a linear space $O(n \log^2 n)$ time recursive algorithm.

In this paper, we present a linear space algorithm with $O(n \log^2 n / \log\log n)$ running time. The speed-up comes from a reduction of the recursion depth of the algorithm in [7] from $O(\log n)$ to $O(\log n / \log\log n)$ levels. Each recursive step now becomes more involved. To deal with this, we show a new technique for using the Monge property in graphs that do not necessarily posses that property. Both [3] and [7] showed how to partition a set of distances that are not Monge, into subsets, each of which is Monge. Exploiting this property, the

* Supported by NSF grant CCF-0635089

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

distances within each subset can be processed efficiently. Here we extend that technique by exhibiting sets of Monge distances whose union is a *superset* of the distances we are actually interested in. We believe this technique may be useful in solving other problems, not necessarily in the context of the Monge property.

From observations in [7], our algorithm can be used to solve bipartite perfect matching, feasible flow, and feasible circulation in planar graphs in $O(n \log^2 n / \log\log n)$ time. Chambers et al. claimed [1] that the algorithm in [7] generalizes to bounded genus graphs. However, the proof (Theorem 3.2 in [1]) is not detailed, and it seems that our new technique is required for its correctness [2]. The resulting running time for fixed genus is also improved to $O(n \log^2 n / \log\log n)$.

The organization of the paper is as follows. In Section 2, we give some definitions and review some basic results. In Section 3 we give an overview of the algorithm of Klein et al. In Section 4 we show how to improve the running time. Finally, we make some concluding remarks in Section 5.

2 Preliminaries

In the following, $G = (V, E)$ denotes an n-vertex planar directed graph with real edge lengths and with no negative cycles. For vertices $u, v \in V$, let $d_G(u, v) \in \mathbb{R} \cup \{\infty\}$ denote the length of a shortest path in G from u to v. We extend this notation to subgraphs of G. We will assume that G is triangulated such that there is a path of finite length between each ordered pair of vertices of G. The new edges added have sufficiently large lengths so that finite shortest path distances in G will not be affected.

Given a graph H, let V_H and E_H denote its vertex set and edge set, respectively. For an edge $e \in E_H$, let $l(e)$ denote the length of e (we omit H in the definition but this should not cause any confusion). Let $P = u_1, \ldots, u_m$ be a path in H, where $|P| = m$. For $1 \leq i \leq j \leq m$, $P[u_i, u_j]$ denotes the subpath $u_i, \ldots, u_j$. If $P' = u_m, \ldots, u_{m'}$ is another path, we define $PP' = u_1, \ldots, u_{m-1}, u_m, u_{m+1}, \ldots, u_{m'}$. Path P' is said to *intersect* P if $V_P \cap V_{P'} \neq \emptyset$.

Define a *region* R to be the subgraph of G induced by a subset of V. In G, the vertices of V_R adjacent to vertices in $V \setminus V_R$ are called *boundary vertices* (of R) and the set of boundary vertices of R is called the *boundary* of R. Vertices of V_R that are not boundary vertices of R are called *interior vertices* (of R).

The cycle separator theorem of Miller [9] states that, given an m-vertex triangulated plane graph, there is a Jordan curve C intersecting $O(\sqrt{m})$ vertices and no edges such that between $m/3$ and $2m/3$ vertices are enclosed by C. Furthermore, this Jordan curve can be found in linear time.

Let $r \in (0, n)$ be a parameter. Fakcharoenphol and Rao [3] showed how to recursively apply the cycle separator theorem so that in $O(n \log n)$ time, (a plane embedding of) G is divided into $O(n/r)$ regions with the following properties:

1. Each region contains at most r vertices and $O(\sqrt{r})$ boundary vertices,
2. No two regions share interior vertices,
3. Each region has a boundary contained in $O(1)$ faces, defined by simple cycles.

We refer to such a division as an *r-division* of G. For simplicity we assume that the $O(1)$ faces in property 3 contain boundary vertices only. This can always be achieved by adding edges between consecutive boundary vertices on each face. Let R be a region in an r-division. We assume that R is enclosed by one of the cycles C in the boundary of R. This can be achieved by adding a new cycle if needed. C is the *external face* of R. Let F be one of the $O(1)$ faces defining the boundary of R. If F is not the external face of R then the subgraph of G enclosed by F (including the boundary vertices of R in F) is called a *hole* of R.

For a graph H, a *price function* is a function $p : V_H \to \mathbb{R}$. The *reduced cost function* induced by p is the function $w_p : E_H \to \mathbb{R}$, defined by

$$w_p(u, v) = p(u) + l(u, v) - p(v).$$

We say that p is a *feasible* price function for H if for all $e \in E_H$, $w_p(e) \geq 0$.

It is well known that reduced cost functions preserve shortest paths, meaning that we can find shortest paths in H by finding shortest paths in H with edge lengths defined by the reduced cost function w_p. Furthermore, given p and the distance in H w.r.t. w_p from a $u \in V_H$ to a $v \in V_H$, we can extract the original distance in H from u to v in constant time [7].

Observe that if p is feasible, Dijkstra's algorithm can be applied to find shortest path distances since then $w_p(e) \geq 0$ for all $e \in E_H$. The distances $d_H(s, u)$ from any $s \in V_H$ are an example of a feasible price function $u \mapsto d_H(s, u)$ (recall that we have assumed that $d_H(s, u) < \infty$ for all $u \in V_H$).

A matrix $M = (M_{ij})$ is *totally monotone* if for every i, i', j, j' such that $i < i'$, $j < j'$, $M_{ij} \leq M_{ij'}$ implies $M_{i'j} \leq M_{i'j'}$. Totally monotone matrices were introduced by Aggarwal et al. [10], who gave an algorithm, nicknamed SMAWK, that, given a totally monotone $n \times m$ matrix M, finds all row minima of M in just $O(n + m)$ time. A matrix $M = (M_{ij})$ is *convex 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 easy to see that if M is convex Monge then it is totally monotone, and that SMAWK can be used to find the column minima of a convex Monge matrix. The algorithm in [7] uses a generalization of SMAWK to so called *falling staircase matrices*, due to Klawe and Kleitman [5]. Klawe and Kleitman's algorithm finds all column minima in $O(m\alpha(n) + n)$ time, where $\alpha(n)$ is the inverse Ackerman function.

3 The Algorithm of Klein et al.

In this section, we give an overview of the algorithm of [7]. Let s be a vertex of G. To find SSSP distances in G with source s, the algorithm finds a cycle separator C with $O(\sqrt{n})$ boundary vertices that separates G into two subgraphs, G_0 and G_1. Let r be any of these boundary vertices. The algorithm consists of five stages:

Recursion: SSSP distances from r are computed recursively in G_0 and G_1.

Intra-part boundary distances: Distances in G_i between every pair of boundary vertices of G_i are computed in $O(n \log n)$ time using the algorithm of [6] for $i = 0, 1$.

Single-source inter-part boundary distances: A variant of Bellman-Ford is used to compute SSSP distances in G from r to all boundary vertices on C. The algorithm consists of $O(\sqrt{n})$ iterations. Each iteration runs in $O(\sqrt{n}\alpha(n))$ time using the algorithm of Klawe and Kleitman [5]. This stage takes $O(n\alpha(n))$ time.

Single-source inter-part distances: Distances from the previous stage are used to modify G such that all edge lengths are non-negative without changing the shortest paths. Dijkstra's algorithm is then used in the modified graph to obtain SSSP distances in G with source r. Total running time for this stage is $O(n \log n)$.

Rerooting single-source distances: The computed distances from r in G form a feasible price function for G. Dijkstra's algorithm is applied to obtain SSSP distances in G with source s in $O(n \log n)$ time.

The last four stages of the algorithm in [7] run in a total of $O(n \log n)$ time. Since there are $O(\log n)$ recursion levels, the total running time is $O(n \log^2 n)$. We next describe how to improve this time bound.

4 An Improved Algorithm

The main idea is to reduce the number of recursion levels by applying the cycle separator theorem of Miller not once but several times at each level of the recursion. More precisely, for a suitable p, we obtain an n/p-division of G in $O(n \log n)$ time. For each region R_i in this n/p-division, we pick an arbitrary boundary vertex r_i and recursively compute SSSP distances in R_i with source r_i. This is similar to the first stage of the algorithm in [7], except that we recurse on $O(p)$ regions instead of just two.

We will show how all these recursively computed distances can be used to compute SSSP distances in G with source s in $O(n \log n + np\alpha(n))$ additional time. This bound is no better than the $O(n \log n)$ bound of the original algorithm but does result in fewer recursion levels. Since the size of regions is reduced by a factor of p with each recursive call, the depth of the recursion is only $O(\log n / \log p)$. Furthermore, by recursively applying the separator theorem of Miller as done by Fakcharoenphol and Rao [3], the subgraphs at the kth recursion level define an r-division of G where $r = n/p^k$. This r-division consists of $O(n/r)$ regions each containing at most r vertices, implying that the total time spent at the kth recursion level is $O(n/r(r \log r + rp\alpha(r))) = O(n \log n + np\alpha(n))$. Summing over all $O(\log n / \log p)$ levels, it follows that the total running time of our algorithm is

$$O\left(\frac{\log n}{\log p}(n \log n + np\alpha(n))\right).$$

To minimize this expression, we set $n \log n = np\alpha(n)$, so $p = \log n / \alpha(n)$. This gives the desired $O(n \log^2 n / \log \log n)$ running time.

It remains to show how to compute SSSP distances in G with source s in $O(n \log n + np\alpha(n)) = O(n \log n)$ time, excluding the time for recursive calls. Assume that we are given an n/p-division of G and that for each region R, we

are given SSSP distances in R with some boundary vertex of R as source. Note that the number of regions is $O(p)$ and each region contains at most n/p vertices and $O(\sqrt{n/p})$ boundary vertices.

The main technical difficulty arises from the existence of holes. We will first describe a generalization of [7] using multiple regions instead of just two, but assuming that no region has holes. In this case, as is the case of [7], all of the boundary vertices in a region are cyclically ordered on its external face. In section 4.4 we show how to handle the existence of holes.

Without holes, the remaining four steps of the algorithm are very similar to those in the algorithm of Klein et al. We give an overview here and go into greater detail in the subsections below. Each step takes $O(n \log n)$ time.

Intra-region boundary distances: For each region R, distances in R between each pair of boundary vertices of R are computed.

Single-source inter-region boundary distances: Distances in G from an arbitrary boundary vertex r of an arbitrary region to all boundary vertices of all regions are computed.

Single-source inter-region distances: Using the distances obtained in the previous stage to obtain a modified graph, distances in G from r to all vertices of G are computed using Dijkstra's algorithm on the modified graph.

Rerooting single-source distances: Identical to the final stage of the original algorithm.

4.1 Intra-region Boundary Distances

Let R be a region. Since R has no holes, we can apply the multiple-source shortest path algorithm of [6] to R since we have a feasible price function from the recursively computed distances in R. Total time for this is $O(|V_R| \log |V_R|)$ time which is $O(n \log n)$ over all regions.

4.2 Single-source Inter-region Boundary Distances

Let r be some boundary vertex of some region. We need to find distances in G from r to all boundary vertices of all regions. To do this, we use a variant of Bellman-Ford similar to the one used in stage three of the original algorithm.

Let $\mathcal{R}$ be the set of $O(p)$ regions, let $B \subseteq V$ be the set of boundary vertices over all regions, and let $b = |B| = O(p\sqrt{n/p}) = O(\sqrt{np})$. Note that a vertex in B may belong to several regions.

Pseudocode of the algorithm is shown in Figure 1. Notice the similarity with the algorithm in [7] but also an important difference: in [7], each table entry $e_j[v]$ is updated only once. Here, it may be updated several times in iteration

j since more than one region may have v as a boundary vertex. For $j \geq 1$, the final value of $e_j[v]$ will be

$$e_j[v] = \min_{w \in B_v} \{e_{j-1}[w] + d_R(w, v)\}, \tag{1}$$

where B_v is the set of boundary vertices of regions having v as boundary vertex.

1. initialize vector $e_j[v]$ for $j = 0, \ldots, b$ and $v \in B$
2. $e_j[v] := \infty$ for all $v \in B$ and $j = 0, \ldots, b$
3. $e_0[r] := 0$
4. **for** $j = 1, \ldots, b$
5. **for** each region $R \in \mathcal{R}$
6. let C be the cycle defining the boundary of R
7. $e_j[v] := \min\{e_j[v], \min_{w \in V_C} \{e_{j-1}[w] + d_R(w, v)\}\}$ for all $v \in V_C$
8. $D[v] := e_b[v]$ for all $v \in B$

Fig. 1. Pseudocode for single-source inter-region boundary distances algorithm.

To show the correctness of the algorithm, we need the following two lemmas.

Lemma 1. *Let P be a simple r-to-v shortest path in G where $v \in B$. Then P can be decomposed into at most b subpaths $P = P_1P_2P_3 \ldots$, where the endpoints of each subpath P_i are boundary vertices and P_i is a shortest path in some region of $\mathcal{R}$.*

Lemma 2. *After iteration j of the algorithm in Figure 1, $e_j[v]$ 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 vertices and P_i is a shortest path in a region of $\mathcal{R}$.*

Both lemmas are straightforward generalizations of the corresponding lemmas in [7]. They imply that after b iterations, $D[v]$ holds the distance in G from r to v for all $v \in B$. This shows the correctness of our algorithm.

Line 7 can be executed in $O(|V_C|\alpha(|V_C|))$ time using the technique of [7] using the distances $d_R(w, v)$ which have been precomputed in the previous stage for all $v, w \in V_C$. It is important to note that the techniques of [7] only apply since we have assumed that all boundary vertices of R are cyclically ordered on its external face. Thus, each iteration of lines 4–7 takes $O(b\alpha(n))$ time, giving a total running time for this stage of $O(b^2\alpha(n)) = O(np\alpha(n))$. Recalling that $p = \log n/\alpha(n)$, this bound is $O(n \log n)$, as desired.

4.3 Single-source Inter-region Distances

In this step we need to compute, for each region R, the distances in G from r to each vertex of R. We apply a nearly identical construction to the one used in the corresponding step of [7].

Let R be a region. Let R' be the graph obtained from R by adding a new vertex r' and an edge from r' to each boundary vertex of R whose length is set to the distance in G from r to the boundary vertex. Note that $d_G(r, v) = d_{R'}(r', v)$ for all $v \in V_R$, so it suffices to find distances in R' from r' to each vertex of V_R.

Let r_R be the boundary vertex of R for which distances in R from r_R to all vertices of R have been recursively computed. Define a price function ϕ for R' as follows. Let B_R be the set of boundary vertices of R and let $D = \max\{d_R(r_R, b) - d_G(r, b) | b \in B_R\}$. Then for all $v \in V_{R'}$,

$$\phi(v) = \begin{cases} d_R(r_R, v) & \text{if } v \neq r' \\ D & \text{if } v = r'. \end{cases}$$

Lemma 3. *Function ϕ defined above is a feasible price function for R'.*

Proof. Let $e = (u, v)$ be an edge of R'. By construction, no edges enter r' so $v \neq r'$. If $u \neq r'$ then $\phi(u) + l(e) - \phi(v) = d_R(r_R, u) + l(u, v) - d_R(r_R, v) \geq 0$ by the triangle inequality so assume that $u = r'$. Then $v \in B_R$ so $\phi(u) + l(e) - \phi(v) = D + d_G(r, v) - d_R(r_R, v) \geq 0$ by definition of D. This shows the lemma. □

Price function ϕ can be computed in time linear in the size of R and Lemma 3 implies that Dijkstra's algorithm can be applied to compute distances in R' from r' to all vertices of V_R in $O(|V_R| \log |V_R|)$ time. Over all regions, this is $O(n \log n)$, as requested.

We omit the description of the last stage where single-source distances are rerooted to source s since it is identical to the last stage of the original algorithm. We have shown that all stages run in $O(n \log n)$ time and it follows that the total running time of our algorithm is $O(n \log^2 n / \log \log n)$. It remains to deal with holes in regions.

4.4 Dealing with Holes

In Sections 4.1 and 4.2, we made the assumption that no region has holes. In this section we remove this restriction. This is the main technical contribution of this paper. As mentioned in Section 2, each region of $\mathcal{R}$ has at most a constant number, h, of holes.

Intra-region boundary distances: In Section 4.1 we used the fact that all boundary vertices of each region are on the external face, to apply the multiple-source shortest path algorithm of [6]. Consider a region R with h holes. If we apply [6] to R we get distances from boundary vertices on the external face of R to all boundary vertices of R. This does not compute distances from boundary vertices belonging to the holes of R. Consider one of the holes of R. We can apply the algorithm of [6] with this hole considered as the external face to get the distances from the boundary vertices of this hole to all boundary vertices of R. Repeating this for all holes, we get distances in R between all pairs of boundary vertices of R in time $O(|V_R| \log |V_R| + h|V_R| \log |V_R|) = O(|V_R| \log |V_R|)$ time. Thus, the time bound in Section 4.1 still holds when regions have holes.

Single-source inter-region boundary distances: It remains to show how to compute single-source inter-region boundary distances when regions have holes. Let C be the external face of region R. Let H_R be the directed graph having the boundary vertices of R as vertices and having an edge (u, v) of length $d_R(u, v)$ between each pair of vertices u and v.

As usual in this context, we say that we relax an edge if it is being considered by the algorithm as the next edge in the shortest path. Line 7 in Figure 1 relaxes all edges in H_R having both endpoints on C. We need to relax all edges of H_R. In the following, when we say that we relax edges of R, we really refer to the edges of H_R.

To relax the edges of R, we consider each pair of cycles (C_1, C_2), where C_1 and C_2 are C or a hole, and we relax all edges starting in C_1 and ending in C_2. This will cover all edges we need to relax.

Since the number of choices of (C_1, C_2) is $O(h^2) = O(1)$, it suffices to show that in a single iteration, the time to relax all edges starting in C_1 and ending in C_2 is $O((|V_{C_1}|+|V_{C_2}|)\alpha(|V_{C_1}|+|V_{C_2}|))$, with $O(|V_R| \log |V_R|)$ preprocessing time. We may assume that $C_1 \neq C_2$, since otherwise we can relax edges as described in Section 4.2.

Before going into the details, let us give an intuitive and informal overview of our approach. We transform R in such a way that C_1 is the external face of R and C_2 is a hole of R. Let P be a simple path from some vertex $r_1 \in V_{C_1}$ to some vertex $r_2 \in V_{C_2}$. Let R_P be the graph obtained by "cutting along P" (see Figure 2). Note that every shortest path in R_P corresponds to a shortest path in R that does not cross P. We will show that relaxing all edges in H_R from C_1 to C_2 with respect to distances in R_P can be done efficiently. Unfortunately, relaxing edges w.r.t. R_P does not suffice since shortest paths in R that do cross P are not represented in R_P. To overcome this obstacle we will identify two particular paths P_r and P_ℓ such that for any $u \in C_1, v \in C_2$ there exists a shortest path in R that does not cross both P_r and P_ℓ. Then, relaxing all edges between boundary vertices once in R_{P_r} and once in R_{P_ℓ} suffices to compute shortest path distances in R. More specifically, let T be a shortest path tree in R from r_1 to all vertices of C_2. The rightmost and leftmost paths in T satisfy the above property (see Figure 3).

We proceed with the formal description. In the following, we define graphs, obtained from R, required in our algorithm. It is assumed that these graphs are constructed in a preprocessing step. Later, we bound construction time.

We transform R in such a way that C_1 is the external face of R and C_2 is a hole of R. We may assume that there is a shortest path in R between every ordered pair of vertices, say, by adding a pair of oppositely directed edges between each consecutive pair of vertices of C_i in some simple walk of C_i, $i = 1, 2$ (if an edge already exists, a new edge is not added). The lengths of the new edges are chosen sufficiently large so that shortest paths in R and their lengths do not change. Where appropriate, we regard R as some fixed planar embedding of that region.

We say that an edge $e = (u, v)$ with exactly one endpoint on path P *emanates right (left) of* P if (a) e is directed away from P, and (b) e is to the right (left) of

P in the direction of P (see e.g., [6] for a more precise definition). If e is directed towards P, then we say that e *enters P from the right (left)* if (v, u) emanates right (left) of P. We extend these definitions to paths and say, e.g., that a path Q emanates right of path P if there is an edge of Q that emanates right of P.

For a simple path P from a vertex $r_1 \in V_{C_1}$ to a vertex $r_2 \in V_{C_2}$, take a copy R_P of R and remove P and all edges incident to P in R_P. let $\overleftarrow{E}$ resp. $\overrightarrow{E}$ be the set of edges that either emanate left resp. right of P or enter P from the left resp. right. Add two copies, $\overleftarrow{P}$ and $\overrightarrow{P}$, of P to R_P. Connect path $\overleftarrow{P}$ resp. $\overrightarrow{P}$ to the rest of R_P by attaching the edges of $\overleftarrow{E}$ resp. $\overrightarrow{E}$ to the path, see Figure 2. If $(u, v) \in E_R$, where $(v, u) \in E_P$, we add (u, v) to both $\overleftarrow{P}$ and $\overrightarrow{P}$ in R_P.

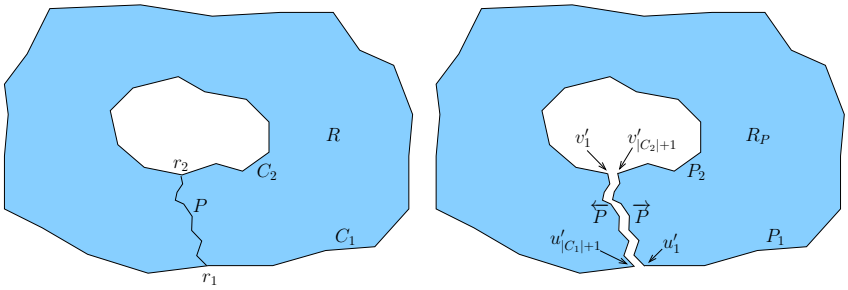

Fig. 2. Region R_P is obtained from R essentially by cutting open at P the "ring" bounded by C_1 and C_2.

A simple, say counter-clockwise, walk $u_1, u_2, \ldots, u_{|C_1|}, u_{|C_1|+1}$ of C_1 in R where $u_1 = u_{|C_1|+1} = r_1$ corresponds to a simple path $P_1 = u'_1, \ldots, u'_{|C_1|+1}$ in R_P. In the following, we identify u_i with u'_i for $i = 2, \ldots, |C_1|$. The vertex r_1 in R corresponds to two vertices in R_P, namely u'_1 and $u'_{|C_1|+1}$. We will identify both of these vertices with r_1. Similarly, a simple, say clockwise, walk of C_2 in R from r_2 to r_2 corresponds to a simple path $P_2 = v'_1, \ldots, v'_{|C_2|+1}$ in R_P. We make a similar identification between vertices of C_2 and P_2.

In the following, when we say that we relax all edges in R_P starting in vertices of C_1 and ending in vertices of C_2, we really refer to relaxing edges in H_R with respect to the distances between the corresponding vertices of P_1 and P_2 in R_P. More precisely, suppose we are in iteration j. Then relaxing all edges entering a vertex $v \in V_{C_2}$ in R_P means updating

$$e_j[v] := \min_{u \in V_{C_1}} \{e_{j-1}[v], e_{j-1}[u] + d_{R_P}(u', v')\}.$$

It is implicit in this notation that if $u = r_1$, we relax w.r.t. both u'_1 and $u'_{|C_1|+1}$ and if $v = r_2$, we relax w.r.t. both v'_1 and $v'_{|C_2|+1}$.

The fact that in R_P P_1 and P_2 both belong to the external face implies (see Lemma 4.3 in [7]):

Lemma 4. *Relaxing all edges from V_{C_1} to V_{C_2} in R_P can be done in $O(|V_{C_1}| + |V_{C_2}|)$ time in any iteration of Bellman-Ford.* □

As we have mentioned, relaxing edges between boundary vertices in R_P does not suffice since shortest paths in R that cross P are not represented in R_P. Let T be a shortest path tree in R from r_1 to all vertices of C_2. A *rightmost (leftmost)* path P in T is a path such that no other path Q in T emanates right (left) of P. Let P_r and P_ℓ be the rightmost and leftmost root-to-leaf simple paths in T, respectively; see Figure 3(a). Let $v_r \in C_2$ and $v_\ell \in C_2$ denote the leaves of P_r and P_ℓ, respectively.

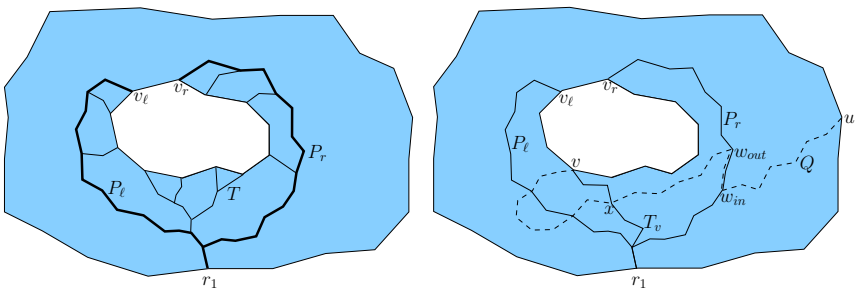

Fig. 3. (a): The rightmost root-to-leaf simple path P_r and the leftmost root-to-leaf simple path P_ℓ in T. (b): In the proof of Lemma 5, if Q first crosses P_r from right to left and then crosses P_ℓ from right to left then there is a u-to-v shortest path in R that does not cross P_ℓ.

In order to state the desired property of P_r and P_ℓ we now define what we mean when we say that path $Q = q_1, q_2, q_3, \ldots$ *crosses* path P. Let out_0 be the smallest index such that q_{out_0} does not belong to P. We recursively define in_i to be smallest index greater than out_{i-1} such that q_{in_i} belongs to P, and out_i to be smallest index greater than in_i such that q_{out_i} does not belong to P. We say that Q crosses P from the right (left) with entry vertex v_{in} and exit vertex v_{out} if (a) $v_{in} = q_{in_i}$ and $v_{out} = q_{out_i - 1}$ for some $i > 0$ and (b) $q_{in_i - 1} q_{in_i}$ enters P from the right (left) and (c) $q_{out_i - 1} q_{out}$ emanates left (right) of P.

Lemma 5. *For any $u \in V_{C_1}$ and any $v \in V_{C_2}$, there is a simple shortest path in R from u to v which does not cross both P_r and P_ℓ.*

Proof. Let Q be a simple u-to-v shortest path in R which is minimal with respect to the total number of time it crosses P_r and P_ℓ. If Q does not cross P_r or P_ℓ, we are done, so assume it crosses both. Also assume that Q crosses P_r first. The case where Q crosses P_ℓ first is symmetric. Let w_{in} and w_{out} be the entry and exit vertices of the first crossing, see Figure 3(b). There are two cases:

– Q first crosses P_r from left to right. In this case Q must cross P_ℓ at the same vertices. In fact, it must be that all root-to-leaf paths in T coincide

until w_{out} and that Q crosses them all. In particular, Q crosses the root-to-v path in T, which we denote by T_v. Since T_v does not cross P_r, the path $Q[u, w_{out}]T_v[w_{out}, v]$ is a shortest u-to-v path in R that does not cross P_r.

– Q first crosses P_r from right to left. Consider the path $S = Q[u, w_{out}]P_r[w_{out}, v_r]$. We claim that Q does not cross S. To see this, assume the contrary and let w' denote the exit point corresponding to the crossing. Since Q is simple, $w' \notin Q[u, w_{out}]$. So $w' \in P_r[w_{out}, v_r]$, but then $Q[u, w_{out}]P_r[w_{out}, w']Q[w', v]$ is a shortest path from u to v in R that crosses P_r and P_ℓ fewer times than Q. But this contradicts the minimality of Q.
Since Q first crosses P_r from right to left and never crosses S, its first crossing with P_ℓ must be right-to-left as well, see Figure 3(b). This implies that Q enters all root-to-leaf paths in T before (not strictly before) it enters P_ℓ. In particular, Q enters T_v. Let x be the entry vertex. Then $Q[u, x]T_v[x, v]$ is a u-to-v shortest path in R that does not cross P_ℓ. □

The algorithm: We can now describe our Bellman-Ford algorithm to relax all edges from vertices of C_1 to vertices of C_2. Pseudocode is shown in Figure 4.

Assume that R_{P_l} and R_{P_r} and distances between pairs of boundary vertices in these graphs have been precomputed. In each iteration j, we relax edges from vertices of V_{C_1} to all $v \in V_{C_2}$ in R_{P_ℓ} and in R_{P_r} (lines 9 and 10). Lemma 5 implies that this corresponds to relaxing all edges in R from vertices of V_{C_1} to vertices of V_{C_2}. By the results in Section 4.2, this suffices to show the correctness of the algorithm.

Lemma 4 shows that lines 9, 10 can each be implemented to run in $O(|V_{C_1}| + |V_{C_2}|)$ time. Thus, each iteration of lines 6–10 takes $O((|V_{C_1}| + |V_{C_2}|)\alpha(|V_{C_1}| + |V_{C_2}|))$ time, as desired.

1. initialize vector $e_j[v]$ for $j = 0, \ldots, b$ and $v \in B$
2. $e_j[v] := \infty$ for all $v \in B$ and $j = 0, \ldots, b$
3. $e_0[r] := 0$
4. **for** $j = 1, \ldots, b$
5. **for** each region $R \in \mathcal{R}$
6. **for** each pair of cycles, C_1 and C_2, defining the boundary of R
7. **if** $C_1 = C_2$, relax edges from C_1 to C_2 as in Section 4.2
8. **else** (assume C_1 is external and that $d_{R_{P_r}}$ and $d_{R_{P_\ell}}$ have been precomputed)
9. $e_j[v] := \min\{e_j[v], \min_{w \in V_{C_1}}\{e_{j-1}[w] + d_{R_{P_r}}(w', v')\}\}$ for all $v \in V_{C_2}$
10. $e_j[v] := \min\{e_j[v], \min_{w \in V_{C_1}}\{e_{j-1}[w] + d_{R_{P_\ell}}(w', v')\}\}$ for all $v \in V_{C_2}$
11. $D[v] := e_b[v]$ for all $v \in B$

Fig. 4. Pseudocode for the Bellman-Ford variant that handles regions with holes.

It remains to show that R_{P_r} and R_{P_ℓ} and distances between boundary vertices in these graphs can be precomputed in $O(|V_R| \log |V_R|)$ time. Shortest path tree T in R with source r_1 can be found in $O(|V_R| \log |V_R|)$ time with Dijkstra using the recursively computed distances in R as a feasible price function

ϕ. Given T, we can find its rightmost path in $O(|V_R|)$ time by starting at the root r_1. When entering a vertex v using the edge uv, leave that vertex on the edge that comes after vu in counterclockwise order. Computing R_{P_r} given P_r also takes $O(|V_R|)$ time. We can next apply Klein's algorithm [6] to compute distances between all pairs of boundary vertices in R_{P_r} in $O(|V_R| \log |V_R|)$ time (here, we use the non-negative edge lengths in R defined by the reduced cost function induced by ϕ). We similarly compute P_ℓ and pairwise distances between boundary vertices in R_{P_ℓ}. We can finally state our result.

Theorem 1. *Given a planar directed graph G with real edge lengths and no negative cycles and given a source vertex s, we can find SSSP distances in G with source s in $O(n \log^2 n / \log\log n)$ time and linear space.* □

5 Concluding Remarks

We gave a linear space algorithm for single-source shortest path distances in a planar directed graph with arbitrary real edge lengths and no negative cycles. The running time is $O(n \log^2 n / \log\log n)$, which improves on the previous bound by a factor of $\log\log n$. As corollaries, bipartite planar perfect matching, feasible flow, and feasible circulation in planar graphs can be solved in $O(n \log^2 n / \log\log n)$ time. The true complexity of the problem remains unsettled as there is a gap between our upper bound and the linear lower bound. Is $O(n \log n)$ time achievable?

References

1. E. W. Chambers, J. Erickson, and A. Nayyeri. Homology flows, cohomology cuts. Proc. 42nd Ann. ACM Symp. Theory Comput., 273–282, 2009.
2. J. Erickson, Private Communication, 2010.
3. 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.
4. M. R. Henzinger, P. N. Klein, S. Rao, and S. Subramanian. Faster shortest-path algorithms for planar graphs. J. Comput. Syst. Sci., 55(1):3–23, 1997.
5. 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.
6. P. N. Klein. Multiple-source shortest paths in planar graphs. Proceedings, 16th ACM-SIAM Symposium on Discrete Algorithms, 2005, pp. 146–155.
7. P. N. Klein, S. Mozes, and O. Weimann. Shortest Paths in Directed Planar Graphs with Negative Lengths: a Linear-Space $O(n \log^2 n)$-Time Algorithm. Proc. 19th Ann. ACM-SIAM Symp. Discrete Algorithms, p. 236–245, 2009.
8. R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized nested dissection. SIAM Journal on Numerical Analysis, 16:346–358, 1979.
9. G. L. Miller. Finding small simple cycle separators for 2-connected planar graphs. J. Comput. Syst. Sci., 32:265–279, 1986.
10. A. Aggarwal, M. Klawe, S. Moran, P. W. Shor, and R. Wilber. Geometric applications of a matrix searching algorithm. SCG '86: Proceedings of the second annual symposium on Computational geometry, 285–292, 1986.