~~Datalogisk Institut~~
KØBENHAVNS UNIVERSITET

DIKU

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

Christian Wulff-Nilsen

Technical Report no. 09/02
ISSN: 0107-8283

Dept. of Computer Science
University of Copenhagen • Universitetsparken 1
DK-2100 Copenhagen • Denmark

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

Christian Wulff-Nilsen [*]

March 26, 2009

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 is an improvement of a recent time bound of $O(n \log^2 n)$ by Klein et al.

1 Introduction

Computing shortest paths in graphs is one of the most fundamental problems in combinatorial optimization with a rich history. Classical shortest path algorithms are the Bellman-Ford algorithm and Dijkstra's algorithm which both find distances from a given source vertex to all other vertices in the graph. The Bellman-Ford algorithm works for general graphs and has running time $O(mn)$ 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 this problem when all edge lengths are non-negative [2]. For planar graphs with arbitrary real edge lengths and with no negative cycles, Lipton, Rose, and Tarjan [6] gave an $O(n^{3/2})$ time algorithm. Henzinger, Klein, Rao, and Subramanian [2] obtained a (not strongly) polynomial bound of $\tilde{O}(n^{4/3})$. Later, Fakcharoenphol and Rao [1] showed how to solve the problem in $O(n \log^3 n)$ time and $O(n \log n)$ space. Recently, Klein, Mozes, and Weimann [5] presented a linear space $O(n \log^2 n)$ time algorithm.

In this paper, we improve the result in [5] by exhibiting a linear space algorithm with $O(n \log^2 n / \log\log n)$ running time.

From the observations in [5], our algorithm can be used to solve bipartite planar perfect matching, feasible flow, and feasible circulation in planar graphs in $O(n \log^2 n / \log\log n)$ time.

[*]Department of Computer Science, University of Copenhagen, koolooz@diku.dk, http://www.diku.dk/~koolooz/

The organization of the paper is as follows. In Section 2, we give some definitions and some basic results, most of them related to planar graphs. Our algorithm is very similar to that of Klein et al. so in Section 3, we give an overview of some of their ideas. We then show how to improve the running time in Section 4. Finally, we make some concluding remarks in Section 5.

2 Definitions and Basic Results

In the following, $G = (V, E)$ denotes an n-vertex planar directed graph with real edge lengths and with no negative cycles. For vertices $u, v \in V$, we 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 shortest path distances in G will not be affected.

Given a graph H, we let V_H and E_H denote its vertex set and edge set, respectively. For an edge $e \in E_H$, we 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. We let $|P| = m$. For $1 \leq i \leq j \leq m$, $P[u_i, u_j]$ denotes the subpath $u_i, \ldots, u_j$. A path P' is said to *intersect* P if $V_P \cap V_{P'} \neq \emptyset$. If $P' = u_m, \ldots, u_{m'}$ is another path, we define $PP' = u_1, \ldots, u_{m-1}, u_m, u_{m+1}, \ldots, u_{m'}$.

Define a *region* R (of G) to be the subgraph of a subset of V induced by G. In G, the vertices of V_R that are 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 [7] states that, given an m-vertex 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 [1] showed how to recursively apply the cycle separator theorem such that in $O(n \log n)$ time, G is divided into $O(n/r)$ regions with some nice 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 a constant number of faces, defined by simple cycles.

We refer to such a division as an *r-division* of G. The bounded faces of a region are its *holes*. To simplify the description of our algorithm, we will refer to all vertices of the cycles containing the boundary of a region as boundary vertices of that region. Furthermore, we will assume that for each region R in an r-division, R is contained in the bounded region defined by one of the cycles C in the boundary of R. Clearly, this can always be achieved by adding a new cycle if needed. We refer to C as the *external face* 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 ϕ 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 [5].

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 $e \in E_H$. An example of a feasible price function is $u \mapsto d_H(s, u)$ for any $s \in V_H$. This assumes that $d_H(s, u) < \infty$ for all $u \in V_H$ which can always be achieved by, say, triangulating H with edges of sufficiently large length so that shortest paths in H will not be affected.

3 The Algorithm of Klein et al.

In this section, we give an overview of the algorithm of [5] before describing our improved algorithm in Section 4.

Let s be a vertex of G. To find SSSP distances in G with source s, the algorithm starts by applying the cycle separator theorem to G. This gives a Jordan curve C which separates G into two subgraphs, G_0 and G_1.

Let $r \in C$ be any boundary vertex. The algorithm consists of five stages:

Recursive call: SSSP distances in G_i with source r are computed recursively for $i = 0, 1$.

Intra-part boundary distances: The distances in G_i between each pair of boundary vertices of G_i are computed using the algorithm of [4] for $i = 0, 1$. This stage takes $O(n \log n)$ time.

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. The algorithm consists of $O(\sqrt{n})$ iterations and each iteration runs in $O(\sqrt{n}\alpha(n))$ time using a result of Klawe and Kleitman [3]. This stage therefore runs in $O(n\alpha(n))$ time.

Single-source inter-part distances: Distances in 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: A price function is obtained from the computed distances from r in G. This price function is feasible for G and Dijkstra's algorithm is applied to obtain SSSP distances in G with source s in $O(n \log n)$ time.

4 Improved Algorithm

As can be seen above, the last four stages of the algorithm in [5] 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 now describe how to improve this time bound.

The 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. 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 stage one in the original algorithm 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 the speed-up comes from the reduced number of recursion levels. Since the size of regions is reduced by a factor of at least p for each recursion level, the depth of the recursion tree is only $O(\log n / \log p)$. 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 $n \log n = np\alpha(n)$. Solving this, we get $p = \log n/\alpha(n)$ which gives a running time of $O(n \log^2 n / \log \log n)$ as requested.

What remains is 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.

So 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.

We will assume in the following that no region has holes. Then all its boundary vertices are cyclically ordered on its external face. We consider holes in Section 4.4.

The remaining part of the algorithm consists of four stages very similar to the algorithm of Klein et al. We give an overview of them here and describe them in greater detail in the subsections below. Each stage 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.

```
1. initialize vector $e_j[v]$ for $j = 0, \ldots, b$ and $v \in B$
2. $e_0[v] := \infty$ for all $v \in B$
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 $v \in B$
```

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

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 [4] 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 that 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 [5] but also an important difference: in [5], 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.

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 $\mathcal{R}$.*

Proof. P is simple so it can use a boundary vertex at most once. There are b boundary vertices in total. A path can only enter and leave a region through boundary vertices of that region. □

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 $\mathcal{R}$.*

Proof. The proof is by induction on j. We omit it since it is similar to that in [5]. $\square$

It follows from (1) and Lemmas 1 and 2 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.

As for running time, note that line 7 can be implemented to run in $O(|V_C|\alpha(|V_C|))$ time using ideas from [5]. This follows from the assumption that R has no holes so its boundary vertices are cyclically ordered on its external face C. 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 requested.

4.3 Single-source Inter-region Distances

To compute distances in G from boundary vertex r to all vertices of G we consider one region at a time. So let R be a region. We need to compute distances in G from r to each vertex of R.

Let R' be the graph obtained from R by adding an edge from r to each boundary vertex of R; the length of this edge is equal to the distance in G from r to the boundary vertex. Note that $d_G(r, v) = d_{R'}(r, v)$ for all $v \in V_{R'}$. Also note that R' has $O(|V_R|)$ vertices and edges and can be computed in $O(|V_R|)$ time, given the distances computed in the previous stage.

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. We 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'. If $e \in E_R$ 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. If $e \notin R$ then $u = r$ and $v = b$ for some $b \in B_R$ so $\phi(u) + l(e) - \phi(v) = D + d_G(r, b) - d_R(r_R, b) \geq 0$ by definition of D. This shows that ϕ is a feasible price function for R'. $\square$

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' (and hence in G) 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 needed the assumption that no region has holes. In this section, we remove this restriction. As mentioned in Section 2, we may assume w.l.o.g. that each region of $\mathcal{R}$ has at most a constant h number of holes.

Intra-region boundary distances: Let us first show how to compute intra-region boundary distances when regions have holes. The reason why it works in $O(n \log n)$ time in Section 4.1 is that all boundary vertices of each region are on the external face, allowing us to apply the multiple-source shortest path algorithm of [4].

Now, consider a region R. If we apply [4] to R we get distances from boundary vertices on the external face of R to all boundary vertices of R. This is not enough. We also need distances from boundary vertices belonging to the holes of R.

So consider one of the holes of R. We can transform R in linear time such that this hole becomes the external face of R. Having done this transformation, we can apply the algorithm of [4] to get distances from boundary vertices of this hole to all boundary vertices of R. If we repeat 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: What remains is the problem of computing single-source inter-region boundary distances when regions have holes. Let C be the external face of region R. Line 7 in Figure 1 only relaxes edges with both endpoints on C. We need to relax all edges having boundary vertices of R as endpoints.

To do this, 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 $C_1 \neq C_2$, for otherwise we can relax edges as described in Section 4.2.

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 will not change. Where appropriate, we will regard R as some fixed planar embedding of that region.

Let $r_1 \in V_{C_1}$ and let T be a shortest path tree in R from r_1 to all vertices of C_2. Let P be the simple path in T from r_1 to an $r_2 \in V_{C_2}$. Define $\overleftarrow{E}$ resp. $\overrightarrow{E}$ as the set of edges e of R with exactly one endpoint on P such that when e is directed away from P it is to the left resp. right of P in the direction of P.

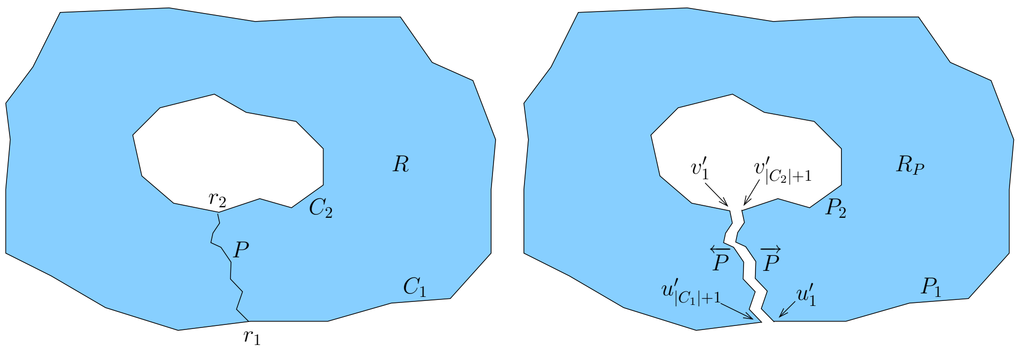

Figure 2: Region R_P is obtained from R essentially by cutting open at P the "ring" bounded by C_1 and C_2.

Now, take a copy R_P of R and remove P and all edges incident to P in R_P. Add two copies, $\overleftarrow{P}$ and $\overrightarrow{P}$, of P to R_P. Connect path $\overleftarrow{P}$ resp. $\overrightarrow{P}$ to 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 $\overleftarrow{P}$ and $\overrightarrow{P}$ in R_P.

In order to relax edges from boundary vertices of C_1 to boundary vertices of C_2 in R, the first step is to relax edges in R_P, defined in the following.

Note that a simple, say counter-clockwise, walk $(u_1 = r_1), u_2, \ldots, u_{|C_1|}, (u_{|C_1|+1} = r_1)$ of C_1 in R starting and ending in 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 $i = 2, \ldots, |C_1|$. Vertex $u_1 = r_1$ 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.

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 e.g. say that we relax all edges of R_P starting in vertices of C_1 and ending in vertices of C_2, we really refer to edges starting in the corresponding vertices of P_1 and ending in the corresponding vertices of P_2. More precisely, suppose we are in iteration j. Then relaxing an edge from a $u \in V_{C_1} \setminus \{r_1\}$ to a $v \in V_{C_2} \setminus \{r_2\}$ in R_P means updating

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

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 $v'_{|C_2|+1}$. We extend these definitions to graphs with a structure similar to R_P.

As the following lemma shows, relaxing edges in R_P can be done efficiently by exploiting the cyclic order of boundary vertices of R_P as we did above for regions with no holes.

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

Proof. Let paths P_1 and P_2 in R_P be defined as above. Consider iteration j. Define a $|P_1| \times |P_2|$ matrix A with elements $A_{kl} = e_{j-1}[u_k] + d_{R_P}(u'_k, v'_l)$. Observe that relaxing all edges from V_{C_1} to V_{C_2} in R_P is equivalent to finding all column-minima of A (compare this to [5]).

Now, since $P_1\overleftarrow{P}P_2\overrightarrow{P}$ is a cycle, it follows easily from results of [5] that for $1 \leq k \leq k' \leq |P_1|$ and $1 \leq l \leq l' \leq |P_2|$,

$$A_{kl} + A_{k'l'} \geq A_{kl'} + A_{k'l}.$$

From [5], this suffices to show that A is a so called falling staircase matrix and by [3], its column-minima can thus be found in $O((|V_{C_1}| + |V_{C_2}|)\alpha(|V_{C_1}| + |V_{C_2}|))$ time. □

Unfortunately, relaxing edges between boundary vertices in R_P will not suffice since some shortest paths in R may not exist in R_P. In the following, we show how to obtain from R two other graphs (due to symmetry, we only focus on one of them) with a structure similar to that of R_P such that relaxing all edges of these graphs will cover the remaining edges.

First, we need some more definitions. Define graphs $\overleftarrow{R_P}$, $\overrightarrow{R_P}$, and $\overleftrightarrow{R_P}$ as follows. Graph $\overleftarrow{R_P}$ is obtained from R_P by adding an edge of length zero from each vertex of $\overrightarrow{P}$ to the corresponding vertex of $\overleftarrow{P}$. Similarly, $\overrightarrow{R_P}$ is obtained from R_P by adding an edge of length zero from each vertex of $\overleftarrow{P}$ to the corresponding vertex of $\overrightarrow{P}$. Finally, $\overleftrightarrow{R_P}$ is obtained from R_P by adding the union of the edges added to $\overleftarrow{R_P}$ and $\overrightarrow{R_P}$. Paths in these three graphs correspond to paths of the same length in R when "contracting" $\overleftarrow{P}$ and $\overrightarrow{P}$. Also, distances between vertices of R are the same as distances in $\overleftrightarrow{R_P}$.

We say that a path in R from a $u \in V_{C_1}$ to a $v \in V_{C_2}$ *crosses* P from the left resp. right if a corresponding path in $\overleftrightarrow{R_P}$ uses one of the added zero-length edges in $\overleftarrow{R_P}$ resp. $\overrightarrow{R_P}$ (in Figure 3(a), path P_v from u to v crosses P from the left and path $P_{v'}$ from u to v' crosses P from the right). We say that a path crosses P if it does so from the left or right. Since we deal with shortest paths, we will assume that any path that crosses P uses exactly one of the added zero-length edges. We extend the definitions to other paths than P.

Note that for any $u \in V_{C_1}$ and $v \in V_{C_2}$, $d_R(u,v) \leq d_{R_P}(u,v)$, with equality if and only if some shortest path in R from u to v does not cross P. Similarly, $d_R(u,v) \leq d_{\overleftarrow{R_P}}(u,v)$ resp. $d_R(u,v) \leq d_{\overrightarrow{R_P}}(u,v)$ with equality if and only if some shortest path in R from u to v does not cross P from the right resp. left.

Let $\overleftarrow{V_1}$ be the set of vertices $u \in V_{C_1}$ for which $d_R(u,v) < d_{\overleftarrow{R_P}}(u,v)$ for some $v \in V_{C_2}$. Similarly, let $\overrightarrow{V_1}$ be the set of vertices $u \in V_{C_1}$ for which $d_R(u,v) < d_{\overrightarrow{R_P}}(u,v)$ for some $v \in V_{C_2}$.

Lemma 5. *$\overleftarrow{V_1} \cap \overrightarrow{V_1} = \emptyset$.*

Proof. Suppose for the sake of contradiction that u is a vertex in $\overleftarrow{V_1} \cap \overrightarrow{V_1}$. Let v and v' be vertices of V_{C_2} such that $d_R(u,v) < d_{\overleftarrow{R_P}}(u,v)$ and $d_R(u,v') < d_{\overrightarrow{R_P}}(u,v')$. Then there is a shortest path P_v in R from u to v that crosses P from the left and a shortest path $P_{v'}$ in R from u to v' that crosses P from the right, see Figure 3(a).

Let $w \in V_P \cap V_{P_v}$ and $w' \in V_P \cap V_{P_{v'}}$. There are two possible cases: either $P_v[u,w]$ intersects $P_{v'}[w',v']$ or $P_v[w,v]$ intersects $P_{v'}[u,w']$. Let x be a vertex of intersection. In the first case, $P_v[u,x]P_{v'}[x,v']$ is a shortest path in R from u to v' that does not cross P, implying $d_R(u,v') = d_{\overrightarrow{R_P}}(u,v')$. In the second case, $P_{v'}[u,x]P_v[x,v]$ is a

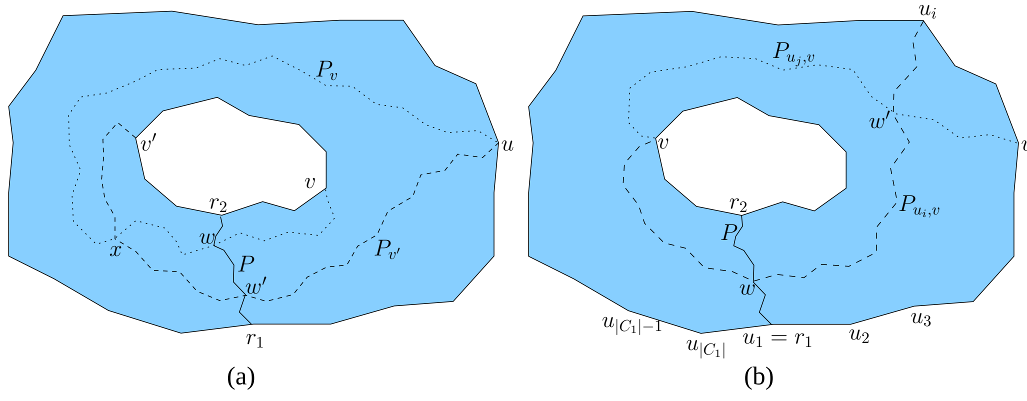

Figure 3: (a): The situation in the proof of Lemma 5. Either $P_v[u, w]$ intersects $P_{v'}[w', v']$ or $P_v[w, v]$ intersects $P_{v'}[u, w']$.(b): The situation in the proof of Lemma 6. If path $P_{u_j,v}$ does not cross P from the right, it has to intersect $P_{u_i,v}[u_i, w]$ in a vertex w'.

shortest path in R from u to v that does not cross P, implying $d_R(u, v) = d_{\overleftarrow{R_P}}(u, v)$. In both cases, we have a contradiction. □

We may restrict our attention to relaxing edges starting in vertices of $\overleftarrow{V_1} \cup \overrightarrow{V_1}$ since all other edges can be relaxed in R_P. Symmetry allows us to further restrict our attention to edges starting in vertices of $\overrightarrow{V_1}$. The following lemma shows how these vertices occur on C_1.

Lemma 6. *Let $(r_1 = u_1), u_2, \ldots, u_{|C_1|}, u_1$ be a counter-clockwise walk of C_1. Then $\overrightarrow{V_1} = \{u_2, \ldots, u_i\}$ for some $i \in \{1, \ldots, |C_1|\}$ (this includes the case $\overrightarrow{V_1} = \emptyset$).*

Proof. For any $v \in V_{C_2}$, the simple path from r_1 to v in shortest path tree T can be regarded as a shortest path in R that does not cross P. Hence, $u_1 \notin \overrightarrow{V_1}$. Pick the largest $i \in \{2, \ldots, |C_1|\}$ such that $u_i \in \overrightarrow{V_1}$ (if no such i exists, $u_1 \notin \overrightarrow{V_1}$ implies that $\overrightarrow{V_1} = \emptyset$ and the lemma is satisfied with $i = 1$). Let $j \in \{2, \ldots, i\}$. We need to show $u_j \in \overrightarrow{V_1}$.

Since $u_i \in \overrightarrow{V_1}$, there is a $v \in V_{C_2}$ such that $d_R(u_i, v) < d_{\overrightarrow{R_P}}(u_i, v)$, implying that any shortest path in R from u_i to v crosses P from the right, see Figure 3(b). Let $P_{u_j,v}$ be a simple shortest path in R from u_j to v. It suffices to show that this path crosses P from the right.

Let $P_{u_i,v}$ be a simple shortest path from u_i to v in R and let w be the first vertex of $P_{u_i,v} \cap P$ when traversing $P_{u_i,v}$. Suppose for the sake of contradiction that $P_{u_j,v}$ does not cross P from the right. Then it has to intersect $P_{u_i,v}[u_i, w]$ in some vertex w' and $P_{u_i,v}[u_i, w']P_{u_j,v}[w', v]$ is a shortest path in R from u_i to v not crossing P from the right, a contradiction. □

Let r_1' be the vertex u_i satisfying Lemma 6, see Figure 4. Let $\overrightarrow{V_2}$ be the set of vertices $v \in V_{C_2}$ such that $v = r_2$ or the simple path in T from r_1 to v is to the right of P, i.e. it uses no edges of $\overleftarrow{E}$. Let r_2' be the last vertex of $\overrightarrow{V_2}$ in a counter-clockwise

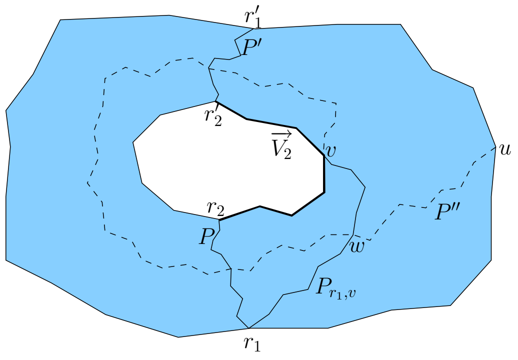

Figure 4: The situation in the proof of Lemma 7. If P'' crosses P from the right, it must intersect $P_{r_1,v}$ in a vertex w before crossing P.

walk of C_2 starting in r_2. By planarity of T, it easily follows that the vertices from r_2 to r_2' in this counter-clockwise walk are exactly the vertices of $\overrightarrow{V_2}$.

Let P' be a simple shortest path in R from r_1' to r_2'. We can assume that P' does not cross P. For suppose otherwise. Since $r_1' \in \overrightarrow{V_1}$, Lemma 5 implies $r_1' \notin \overleftarrow{V_1}$ so we may assume that P' crosses P from the right. Since the simple path P'' in T from r_1 to r_2' is not to the left of P, part of P' and part of P'' together give a shortest path from r_1' to r_2' not crossing P, as requested.

Now, define graph $R_{P'}$ as R_P with P replaced by P' in the definition. The following lemma implies that after relaxing edges of R_P in any iteration, all remaining edges in R starting in vertices of $V_{C_1} \setminus \overleftarrow{V_1}$ can be relaxed in $R_{P'}$. Clearly, Lemma 4 applies to $R_{P'}$, implying that in R, all edges starting in $V_{C_1} \setminus \overleftarrow{V_1}$ can be relaxed efficiently.

Lemma 7. *For any $u \in V_{C_1} \setminus \overleftarrow{V_1}$, $v \in V_{C_2}$, there is a simple shortest path in R from u to v which does not cross both P and P'.*

Proof. We may assume that $u \in \overrightarrow{V_1}$, $u \neq r_1, r_1'$, and $v \neq r_2, r_2'$. Let P'' be a simple shortest path in R from u to v and suppose it crosses both P and P'. Then $v \in \overrightarrow{V_2}$, see Figure 4.

Let $P_{r_1,v}$ be the simple path in T from r_1 to v. Since $v \in \overrightarrow{V_2}$, $P_{r_1,v}$ is not to the left of P.

By Lemma 5, $u \in \overrightarrow{V_1} \Rightarrow u \notin \overleftarrow{V_1}$ so there is a shortest path in R from u to v that crosses P from the right. Hence, we may assume that P'' crosses P before P' (or both simultaneously). But then P'' must intersect $P_{r_1,v}$ in a vertex w before crossing P so $P''[u,w]P_{r_1,v}[w,v]$ is a shortest path in R from u to v which does not cross P, as requested. □

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 5.

Assume that R_P and $R_{P'}$ and distances between pairs of boundary vertices in these graphs have been precomputed.

In each iteration j, we relax edges from vertices of $V_{C_1} \setminus \overleftarrow{V_1}$ to all $v \in V_{C_2}$ in R_P and $R_{P'}$ (lines 9 and 10). Lemma 7 implies that this corresponds to relaxing all

```
1.  initialize vector $e_j[v]$ for $j = 0, \ldots, b$ and $v \in B$
2.  $e_0[v] := \infty$ for all $v \in B$
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 [5]
8.              else (assume $C_1$ is external and that $d_{R_P}$ and $d_{R_{P'}}$ have been precomputed)
9.                  $e_j[v] := \min\{e_j[v], \min_{w \in V_{C_1} \setminus \overleftarrow{V_1}} \{e_{j-1}[w] + d_{R_P}(w, v)\}\}$ for all $v \in V_{C_2}$
10.                 $e_j[v] := \min\{e_j[v], \min_{w \in V_{C_1} \setminus \overleftarrow{V_1}} \{e_{j-1}[w] + d_{R_{P'}}(w, v)\}\}$ for all $v \in V_{C_2}$
11.                 update $e_j$ similarly for $w \in \overleftarrow{V_1}$
12. $D[v] := e_b[v]$ for all $v \in B$
```

Figure 5: Pseudocode for the Bellman-Ford variant that handles regions with holes.

edges in R from vertices of $V_{C_1} \setminus \overleftarrow{V_1}$ to vertices of V_{C_2}. Similarly, executing line 11 corresponds to relaxing all edges in R from vertices of $\overleftarrow{V_1}$ 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 line 9 can be implemented to run in $O((|V_{C_1}|+|V_{C_2}|)\alpha(|V_{C_1}|+|V_{C_2}|))$ time (actually, in Lemma 4 we relax *all* edges from V_{C_1} to V_{C_2} in R_P, not just those starting in $V_{C_1} \setminus \overleftarrow{V_1}$ but this does not change the correctness) and by symmetry, we can also execute lines 10 and 11 within the same time bound. Thus, each iteration of lines 6–11 takes $O((|V_{C_1}| + |V_{C_2}|)\alpha(|V_{C_1}| + |V_{C_2}|))$ time, as requested.

It remains to show that R_P and $R_{P'}$ and distances between boundary vertices in these graphs can be precomputed in $O(|V_R| \log |V_R|)$ time (we also need to construct a graph $R_{P''}$ similar to $R_{P'}$ and to compute distances between boundary vertices in $R_{P''}$ in order to relax edges in line 11. Due to symmetry, we omit the analysis for $R_{P''}$).

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 feasible price function ϕ obtained from the recursively computed distances in R. Given T, we can find P and R_P in $O(|V_R|)$ time. We can then apply Klein's algorithm [4] to compute distances between all pairs of boundary vertices in R_P 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).

All of these ideas also apply to $R_{P'}$ so the only remaining problem is how to obtain P'. If we are given endpoints r'_1 and r'_2, we can find P' in $O(|V_R| \log |V_R|)$ time using Dijkstra so let us show how to find r'_1 and r'_2 within this time bound.

Since we are given T, we can traverse it to find r'_2 in $O(|V_R|)$ time. As for r'_1, let $u \in V_{C_1}$. Using Klein's algorithm and price function ϕ, we can compute distances between boundary vertices in $\overrightarrow{R_P}$ in $O(|V_R| \log |V_R|)$ time. Given these distances and the distances between boundary vertices in R, we can determine whether $u \in \overrightarrow{V_1}$, in $O(|V_{C_2}|)$ time. Thus, r'_1 can be found in $O(|V_{C_1}||V_{C_2}|) = O(|V_R|)$ time.

We can now 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.*

Proof. We gave the bound on running time above. To bound the space, first note that finding an n/p-division of G using the algorithm of [1] requires $O(n)$ space. Klein's algorithm [4] and Dijkstra also has linear space requirement. The recursively computed distances take up a total of $O(p\frac{n}{p}) = O(n)$ space. In the intra-region boundary distances stage, the total memory spent on storing distances is $O(p(\sqrt{n/p})^2) = O(n)$.

In the single-source inter-region boundary distances stage, we need to bound the space for our Bellman-Ford variant. The size of each table is $O(b) = O(n)$. Since we only need to keep tables from the current and previous iteration in memory, Bellman-Ford uses $O(n)$ space.

It is easy to see that the last two stages use $O(n)$ space. Hence the entire algorithm has linear space requirement. $\square$

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. Running time is $O(n \log^2 n/ \log \log n)$, an improvement of a 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.

References

[1] J. Fakcharoenphol and S. Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. Available from the authors' webpages. Preliminary version in FOCS'01.

[2] 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.

[3] 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.

[4] P. N. Klein. Multiple-source shortest paths in planar graphs. Proceedings, 16th ACM-SIAM Symposium on Discrete Algorithms, 2005, pp. 146–155.

[5] 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.

[6] R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized nested dissection. SIAM Journal on Numerical Analysis, 16:346–358, 1979.

[7] G. L. Miller. Finding small simple cycle separators for 2-connected planar graphs. J. Comput. Syst. Sci., 32:265–279, 1986.