Linear-Time Algorithms for Max Flow and Multiple-Source Shortest Paths in Unit-Weight Planar Graphs

David Eisenstat*
Computer Science Department
Brown University
Providence, RI 02912
david@cs.brown.edu

Philip N. Klein*
Computer Science Department
Brown University
Providence, RI 02912
klein@brown.edu

ABSTRACT

We give simple linear-time algorithms for two problems in planar graphs: max st-flow in directed graphs with unit capacities, and multiple-source shortest paths in undirected graphs with unit lengths.

Categories and Subject Descriptors: G.2.2 [Discrete Mathematics]: Graph Theory—*Network problems*; F.2.2 [Analysis of Algorithms]: Nonnumerical Algorithms—*Computations on discrete structures*

General Terms: Algorithms, Performance

Keywords: combinatorial optimization; planar graphs; max flow; multiple-source shortest paths; Menger problem

1. INTRODUCTION

In this paper, we address two fundamental problems in planar graphs: directed max flow and multiple-source shortest paths (MSSP). Via planar duality, these are closely related problems. The best algorithms known for these problems each take $O(n \log n)$ time for arbitrary nonnegative weights. Through the work of Erickson [13] and that of Cabello, Chambers, and Erickson [8, 9], it has become apparent that these algorithms are also closely related. Since these algorithms have emerged as crucial subroutines in algorithms for other problems, researchers in fast algorithms for planar graphs are very interested in finding faster algorithms for these problems.

For example, Italiano, Nussbaum, Sankowski, and Wulff-Nilsen [20] have given an $O(n \log \log n)$ algorithm for max flow in the special case of undirected planar graphs (i.e., symmetric capacities).

For undirected multiple-source shortest paths, we give an $\Omega(n \log n)$ time lower bound in the linear-decision-tree model. Motivated by this result, we consider the special case in which the edge-weights are small integers (given in unary). For this case, we can give a linear-time algorithm for undirected multiple-source shortest paths.

As a consequence, we can obtain improved bounds for several other problems under the same special case, e.g., Steiner tree approximation [4], dynamic shortest-path distances [15, 23], and negative-cycle detection [26, 29].

We use the same technique to obtain a simple linear-time algorithm for max flow in directed planar graphs with small integer capacities. A linear-time time algorithm for max-flow in unit-weight directed planar graphs was previously given by Brandes and Wagner [6] but the algorithm and its proof of correctness are quite complicated.

1.1 Unit-weight directed max flow in linear time

Study of max st-flow in planar graphs has a long history. The first publication on max st-flow, that of Ford and Fulkerson in 1956, gave an algorithm for the special case where s and t are on the same face (the *st-planar* case). Since then there have been many papers giving faster algorithms for various cases. At this point, the best upper bounds known for arbitrary positive capacities are: $O(n \log n)$ for directed planar graphs [3], $O(n \log \log n)$ for undirected planar graphs [20], and $O(n)$ for the st-planar case [18, 19].

The algorithm for undirected planar graphs uses a combination of several sophisticated data structures and algorithmic techniques: an $O(\log^6 n)$-division [16] based on cycle separators, a sophisticated data structure [15], and the multiple-source shortest-paths algorithm. The linear-time algorithm for the st-planar case uses a recursive r-division [16] with roughly $O(\log^* n)$ levels.

For the case where every edge has unit capacity, Weihe [33] gave a linear-time algorithm for undirected planar graphs, and Brandes and Wagner [6] gave a linear-time algorithm for directed planar graphs. These algorithms are rather complicated, and make use of a sophisticated data structure due to Gabow and Tarjan [17] that requires preprocessing to construct tables containing solutions to all small instances. In addition, the proofs of correctness are quite intricate and involved.

Once a maximum st-flow is found, a minimum st-cut can be obtained in linear time, and in most applications one is interested in the st-cut. The original motivation for the study of max-flow (see [32]) was a navy think-tank's study of interdiction of the Soviet rail network. The problem also arises in a variety of low-level computer vision techniques,

*Research supported in part by NSF Grant CCF-0964037

STOC'13, June 1–4, 2013, Palo Alto, California, USA.

and in fact a version of the algorithm of [3] has been implemented by computer-vision researchers [31]. An ecologist is using min-cuts in very large planar graphs to identify critical barriers to dispersal and gene-flow between subdivided populations of threatened species, such as jaguars in South America [T. H. Keitt, personal communication].

In this paper, we give a linear-time algorithm for max st-flow in directed planar graphs with unit-capacity arcs. Our algorithm is simple both to implement and analyze. It uses no data structure more complicated than a queue for breadth-first search, and uses no separators or divide-and-conquer.

Vertex capacities. While traditional max flow addresses edge-capacities, the case of vertex-capacities has also been addressed. Rippenhausen-Lipa, Wagner, and Weihe [30] gave a linear-time algorithm for max st-flow in undirected planar graphs with unit vertex-capacities Kaplan and Nussbaum [21] gave a linear-time reduction from st-flow in directed planar graphs with vertex capacities to st-flow in directed planar graphs with edge capacities. Use of this reduction together with the linear-time algorithm of Brandes and Wagner [6] yields a linear-time algorithm for max st-flow in directed planar graphs with unit vertex capacities. Our algorithm can be substituted for that of Brandes and Wagner to get a simpler algorithm for this problem.

1.2 Unit-weight multiple-source shortest paths in linear time

The problem *multiple-source shortest paths in planar graphs* (MSSP) is informally as follows: given a planar embedded graph, find a representation of shortest-path trees rooted at each of the vertices on the boundary of the infinite face. In Section 4, we give more details on the representation; the basic idea is to list the changes to the shortest-path tree as the root goes from one vertex to another in order around the boundary of the infinite face. Klein [23] first described the problem, showed that the representation had size $O(n)$, gave an $O(n \log n)$-time algorithm to find this representation, and showed that the representation facilitates finding distances from these roots to other specified vertices in $O(\log n)$ time per distance.

There has been much subsequent work. Cabello, Chambers, and Erickson [8, 9] described a simpler $O(n \log n)$ algorithm and generalized it to graphs embedded on higher-genus surfaces. Fast MSSP plays an essential role in a variety of recent fast algorithms for planar and bounded-genus graphs [5, 7, 11, 22, 26, 10, 20, 27, 29, 28]. Other algorithms, including a series of fast approximation schemes, use variants of MSSP [1, 2, 4, 12, 14, 24, 25, 34].

Due to the many uses of MSSP, there is intense interest within the field of fast planar-graph algorithms in obtaining an algorithm for the whose running time is $o(n \log n)$. We show that, in a model in which the edge-lengths can only be added and compared, there is no such algorithm: the MSSP problem requires $\Omega(n \log n)$ time.

On the other hand, we show that, for planar graphs with undirected unit-length edges, there is a *linear-time* algorithm to find the representation of all the shortest-path trees. In fact, our algorithm can handle directed graphs where arcs have nonnegative integer lengths and each arc has a reverse arc. For a graph whose arc lengths sum to L, the running time of our algorithm is $O(n + L)$.

In many published uses of the MSSP algorithm, the algorithm is applied to a n-vertex planar graph whose infinite face has $O(\sqrt{n})$ vertices, and the goal is to find all the distances between vertices on the infinite face. The complete graph between these vertices, with edge-lengths defined to be the distances, is called the *dense distance graph* [15]. As pointed out in [23], since the algorithm requires $O(\log n)$ per distance, this can be done in $O(n \log n)$ time. In this paper, we show that our MSSP algorithm can be extended to compute all of these distances in *linear time*.

Note that our lower bound for MSSP does not apply to this more specific problem, finding the distance between all vertices on the infinite face.

1.3 $O(n\alpha(n) \log n)$ algorithm for shortest paths in planar graphs with small positive and negative lengths

Consider the problem of computing single-source shortest paths in a directed planar graph with positive and negative lengths. For this problem, Klein, Mozes, and Weimann gave an $O(n \log^2 n)$ algorithm [26], and Mozes and Wulff-Nilsen gave an improvement to $O(n \log^2 n / \log \log n)$ time [29]. The bottleneck is computing the dense distance graph.

For the case where the lengths are (positive and negative) integers of bounded magnitude, the $O(n \log^2 n)$ algorithm becomes an $O(n\alpha(n) \log n)$ algorithm when using our linear-time algorithm for computing the dense distance graph. For example, when the lengths are in $\{+1, -1, 0\}$, this gives an $O(n\alpha(n) \log n)$ algorithm to find a negative-length cycle.

1.4 Linear-time approximation scheme for unit-weight Steiner tree

Borradaile, Klein, and Mathieu [4] described an $O(n \log n)$ approximation scheme for Steiner tree in undirected planar graphs. The theoretical bottleneck in this approximation scheme is a construction called *strips*, from [24]. This construction is an adaptation of the MSSP algorithm. For planar Steiner-tree instances with undirected unit-length edges, we can carry out the construction in linear time and consequently obtain a linear-time approximation scheme for such instances.

2. PRELIMINARIES

2.1 Darts

Fix a directed graph $G = (V, E)$. Each arc e corresponds to two darts, one co-oriented with e (same head and tail) and one oppositely oriented (head of one is tail of the other). The head and tail of a dart d are written $\text{head}_G(d)$ and $\text{tail}_G(d)$. (The subscript can be omitted when the choice of graph is clear.) The function $\text{rev}(\cdot)$ is the involution on the dart-set that maps each dart to the oppositely directed dart corresponding to the same arc. A *dart vector* is a vector assigning a number to each dart. In this paper, we restrict our attention to *integer-valued* dart vectors.

Darts are useful in discussing both network flow and planar embeddings.

2.2 Max flow

A *flow assignment* is a dart vector $\mathbf{\Phi}$ satisfying *antisymmetry*, i.e., $\mathbf{\Phi}[\text{rev}(d)] = -\mathbf{\Phi}[d]$ for each dart d. A flow assignment $\mathbf{\Phi}$ satisfies conservation at a vertex v if the sum

$\sum_d \boldsymbol{\Phi}[d]$ is zero, where the sum is over all darts d with head v. An st-flow is a flow assignment that satisfies conservation at every vertex except s and t. The *value* of an st flow is the sum $\sum_d \boldsymbol{\Phi}[d]$ where the sum is over all darts d with head t.

A dart vector $\boldsymbol{c}$ can be interpreted as an assignment of capacities to darts. A flow assignment $\boldsymbol{\Phi}$ is feasible with respect to a capacity assignment $\boldsymbol{c}$ if $\boldsymbol{\Phi}[d] \leq \boldsymbol{c}[d]$ for every dart d. Max st-flow is the problem of finding a feasible st-flow of maximum value.

Note that $\boldsymbol{c}$ can assign different capacities to the two darts corresponding to a single edge. In traditional max-flow, the capacities are all nonnegative. The *undirected* max st-flow problem corresponds to having symmetric capacities, i.e., $\boldsymbol{c}[d] = \boldsymbol{c}[\text{rev}(d)]$ for every dart d. Typically, the *directed* max st-flow problem corresponds to the case where, for each edge e, the oppositely directed dart has capacity zero.

Many algorithms for max flow use the notion of *residual capacities*. Given a capacity vector $\boldsymbol{c}$ and a flow vector $\boldsymbol{\Phi}$, the *residual capacity vector* is $\boldsymbol{c} - \boldsymbol{\Phi}$. The residual capacity of a dart d is then $\boldsymbol{c}[d] - \boldsymbol{\Phi}[d]$. A dart is *residual* if its residual capacity is positive.

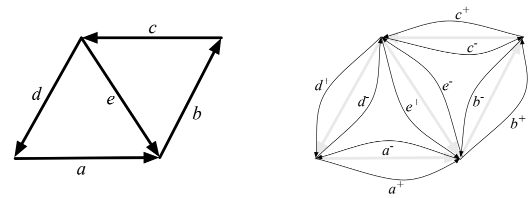
2.3 Shortest paths

A dart vector $\boldsymbol{c}$ can also be interpreted as an assignment of lengths to darts in defining an instance of shortest paths. Again, the case of shortest paths in undirected graphs corresponds to the symmetric case: $\boldsymbol{c}[d] = \boldsymbol{c}[\text{rev}(d)]$. In traditional *directed* shortest-path problems, for each edge e, the oppositely directed dart has *infinite* length. The shortest-path algorithms introduced in this paper cannot cope with infinite lengths, so they cannot handle traditional directed shortest-path problems. However, the algorithms *can* handle asymmetric length assignments; the running time depends on the sum $L = \sum_d \boldsymbol{c}[d]$ of the lengths of all darts.

Our algorithm for max-flow computes shortest paths in the dual with respect to asymmetric length assignments. Note that whenever we refer to a shortest path, we mean a path of *darts* and the possibly asymmetric length assignment is used to measure the length of the path.

2.4 Graph embeddings

A planar embedded graph is a graph drawn on the plane or the sphere such that no edges cross. The faces are the connected components of the set of points not in the image of the embedding. Such geometric embeddings are helpful for intuition but, for the purposes of formal proofs and of implementation, combinatorial embeddings are more useful. A *combinatorial embedding* of a graph is a permutation π of the graph's darts with the following property: the orbits are the sets $\{\{\text{darts with head } v\} : v \in V\}$. That is, each of the permutation cycles comprising π is a permutation cycle on the darts with a common head. A drawing can be obtained from a combinatorial embedding by arranging the darts about a vertex v in counterclockwise or clockwise order according to the corresponding permutation cycle. In the following figure, for example,



the permutation cycle associated with the top-left vertex is $(d^- \ e^- \ c^+)$, and the one associated with the bottom-right vertex is $(b^- \ e^+ \ a^+)$.

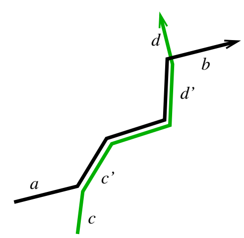
For a combinatorially embedded graph, the *faces* are the permutation cycles comprising $\text{rev} \circ \pi$, where $\circ$ denotes functional composition. A combinatorially embedded graph is planar embedded if

$$\#\text{vertices} - \#\text{edges} + \#\text{faces} = 2\#\text{components}.$$

Let G be an embedded graph, and let T be a spanning tree of G. An edge e of G that is not in T forms a simple cycle C with T. For each dart d of e, there is an orientation of C consistent with d. We say d is *right-to-left* with respect to T if the orientation is counterclockwise.

2.5 Crossings

Let $a\ P\ b$ and $c\ Q\ d$ be two paths that are identical except for their first and last darts, which differ. Let c' be the successor of c in Q and let d' be the predecessor of d in Q. We say Q forms a *crossing configuration* with P

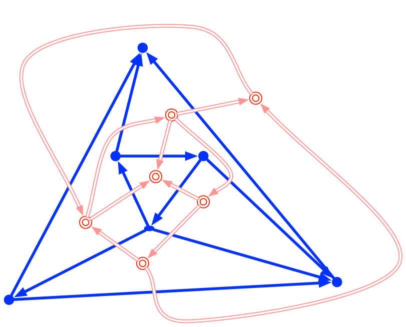


if the permutation cycle at head(c) induces the cycle $(c\ c'\ \text{rev}(a))$ and the permutation cycle at tail(d) induces the cycle $(\text{rev}(d')\ b\ d)$, or the analogous statement with P and Q interchanged.

We say a walk P *crosses* a walk Q if a subwalk of P and a subwalk of Q form a crossing configuration.

2.6 Dual

For a geometric embedding of a connected planar graph, the *dual* is another embedded planar graph. The vertices of the dual are the faces.



Corresponding to each edge e of the primal (the original graph), there is an edge of the dual; it crosses the embedding of e at an angle. For notational convenience, we identify the edge of the dual with the corresponding edge of the primal.

Alternatively, given a combinatorial embedding π of a graph G, the dual is the embedded graph whose embedding is $\text{rev} \circ \pi$. The dual of a planar embedded graph is a planar embedded graph.

The combinatorial definition of the dual defines the orientations of edges of the dual in terms of the orientations of edges of the primal.

For the geometric and combinatorial definitions to be consistent, one must adopt a *counterclockwise* interpretation for the primal and a *clockwise* interpretation for the dual, or vice versa.

Both of the two algorithms we present involve maintaining a shortest-path tree. In the max flow algorithm, the shortest-path tree is a tree in the *dual*; in the multiple-source shortest-path algorithm, the shortest-path tree is in the *primal*. In order for our theorems about shortest-path trees to be consistent, we adopt the following convention. A dart d in the graph containing the shortest-path tree is rotated ninety degrees to get the same dart in the dual graph.

A classical result in planar graphs that underlies the algorithms is:

THEOREM 2.1 (VON STAUDT, 1847). *Let T be a spanning tree of a planar embedded graph. The edges of the graph that are not in T form a spanning tree of the dual.*

3. MAX ST-FLOW

Our algorithm for max *st*-flow is a modification of the algorithm of Borradaile and Klein [3]. Our analysis of the algorithm builds on the elegant formulation and analysis of Erickson [13].[1] In particular, we strengthen the theorem "each dart is inserted into T^* at most once" to "each dart is (either inserted into T^* or replaced in T by its reverse) at most once".

3.1 High-level description of the algorithm

Let $\boldsymbol{c}$ be the capacity vector on darts. Since the infinite face f^∞ can be chosen arbitrarily, we select it to be incident to the sink t.

The algorithm calculates an f^∞-rooted shortest-path tree T^* in the dual G^* using $\boldsymbol{c}$ as a cost function. The algorithm represents T^* by a table pred$[\cdot]$ giving, for each face f other than f^∞, the parent dart in T^*, i.e., the dart in T^* whose head is f. This representation of T^* was suggested by Schmidt et al. [31] and by Erickson [13].

The algorithm initializes $\boldsymbol{\Phi}$ to be the circulation whose potential function is the function giving the distances of faces in the dual shortest-path tree. This circulation obeys the capacities $\boldsymbol{c}$. The algorithm builds a spanning tree T of the primal G using edges not represented in the dual shortest-path tree.

While T contains an s-to-t path, the algorithm repeats the following steps. If there is a nonresidual dart on the path, the algorithm selects the first such dart $\hat{d}$ and removes it from T. The algorithm adds $\hat{d}$ to T^*, removing the dart in T^* with the same head so as to preserve the property that T is a rooted tree. The reverse of the dart removed from T^* is then added to T. (Such an iteration is called a *pivot*.) Once there is no nonresidual dart on the s-to-t path in T, the algorithm pushes one unit of flow along this path.

In the following pseudocode, we use $dist_{\boldsymbol{c}}(f)$ to denote the f^∞-to-f distance in G^* with respect to costs $\boldsymbol{c}$.

def MAXFLOW($G, \boldsymbol{c}, s, t, f^\infty$):
1 $T^* := f^\infty$-rooted shortest-path tree in G^* w.r.t. $\boldsymbol{c}$
2 for each vertex $f \neq f^\infty$ of G^*,
3 pred$[f]$:= the dart in T^* whose head is f
4 for each dart d,
5 $\boldsymbol{\Phi}[d] := dist_{\boldsymbol{c}}(\text{head of } d \text{ in } G^*) - dist_{\boldsymbol{c}}(\text{tail of } d \text{ in } G^*)$
6 let T be the tree formed by edges not represented in T^*
7 while t is reachable from s in T:
8 while $\exists$ a nonresidual dart on the s-to-t path in T,
9 let $\hat{d}$ be the first such nonresidual dart
10 let q be the head of $\hat{d}$ in G^*
11 eject $\hat{d}$ from T and insert rev(pred$[q]$) into T
12 pred$[q] := \hat{d}$
13 for each dart d on the s-to-t path in T,
14 $\boldsymbol{\Phi}[d] := \boldsymbol{\Phi}[d] + 1$.

It can be shown (see [3, 13]) that the algorithm maintains the following invariants:

1: $\boldsymbol{\Phi}[d] \leq \boldsymbol{c}[d]$ for each dart d.
2: $\boldsymbol{\Phi}[d] = \boldsymbol{c}[d]$ for every dart d in T^*.
3: Until the current flow is maximum, Line 11 preserves the existence of an s-to-t path in T and Line 12 preserves the property that T^* is a rooted, oriented tree.

The vector $\boldsymbol{c} - \boldsymbol{\Phi}$ defines the residual capacities of darts. Interpret these as costs in G^*. By Property 1, every dart has nonnegative cost. By Property 2, every dart in T^* has zero cost. By Property 3, until the flow is maximum, T^* is a rooted tree; it follows that it is a shortest-path tree with respect to $\boldsymbol{c} - \boldsymbol{\Phi}$.

3.2 Noncrossing

We briefly review the concepts underlying Erickson's analysis, and then we state a theorem we need for the analysis of our algorithm. Fix an s-to-t path Q of darts in G and let D be a set of darts. The *crossing number* of D with respect to Q is

$$\pi_Q(D) = \sum_{d \in P} (1 \text{ if } d \in D,\ -1 \text{ if } \text{rev}(d) \in D,\ 0 \text{ otherwise}).$$

LEMMA 3.1 (ERICKSON). *Consider an iteration, and let $\hat{d}$ be the dart ejected from the primal tree T. The iteration increases $\pi_Q(T^*[head_{G^*}(\hat{d})])$ by 1.*

LEMMA 3.2 (ERICKSON). *For any face f, $T^*[f]$ is the shortest f^∞-to-f path with respect to $\boldsymbol{c}$ among all such paths with crossing number $\pi_Q(T^*[f])$.*

To model shortest paths with given crossing numbers, Erickson derives from G^* an infinite planar embedded graph $\bar{G}^*$ as follows. Create an infinite number of copies of G^*: $\ldots, G^*_{-2}, G^*_{-1}, G^*_0, G^*_1, G^*_2, \ldots$. For every integer i and every dart xy in P, replace the copy of xy in G^*_i with a dart whose tail is the copy of x in G^*_i and whose tail is the copy of y in

[1]However, Erickson's analysis uses a *genericity* assumption about the capacities. As he points out, this assumption can be achieved by using tiny perturbations of capacities. We cannot use such a technique in this paper; instead we use the fact that our algorithm always selects the leafmost nonresidual dart to be inserted into the shortest-path tree.

G^*_{i+1}. The reverses of darts in P are treated similarly but go from G^*_{i+1} to G^*_i. The costs of darts in $\bar{G}^*$ are defined to be the costs of the corresponding original darts in G^*.

A path in G^* from f^∞ to f with crossing number k corresponds in $\bar{G}^*$ to a path from $(f^\infty)_0$ to f_k, a path from $(f^\infty)_1$ to f_{k+1}, a path from $(f^\infty)_{-1}$ to f_{k-1}, and so on, where we use subscripts to indicate which copy of f^∞ or f we mean.

Fix a face f. For $j = 0, 1, 2\ldots$, let P^f_j denote the f^∞-to-f path in T^* after j iterations of the algorithm. Let $\bar{P}^f_j$ denote the corresponding path in $\bar{G}^*$ from $(f^\infty)_{-k}$ to f_0 where $k = \pi_Q(P^f_j)$. Lemma 3.2 then implies the following.

Corollary 3.3. *For $j = 0, 1, \ldots$, $\bar{P}^f_j$ is the shortest path in $\bar{G}^*$ from $(f^\infty)_{-\pi_Q(P^f_j)}$ to f_0.*

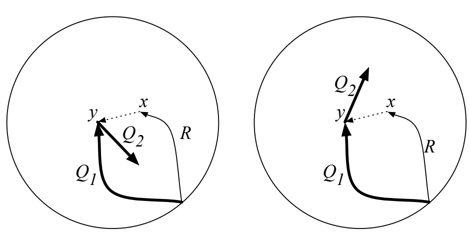
Our analysis is based on the following theorem.

Theorem 3.4. *For every vertex f of G^*, the paths $\bar{P}^f_0, \bar{P}^f_1, \bar{P}^f_2, \ldots$ are mutually noncrossing.*

Under Erickson's genericity assumption, there is a unique shortest path between every pair of vertices in $\bar{G}^*$, so the theorem is immediate (since a crossing would give rise to two distinct shortest paths with the same endpoints). Since we cannot make this assumption, we depend for the proof on the fact that, in Line 9, the algorithm selects the *first* nonresidual dart.

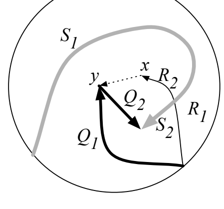
Proof. Assume that the theorem is not true. Let j be the minimum integer such that, for some vertex v and some integer $i < j$, the lifted path $\bar{P}^v_j$ crosses $\bar{P}^v_i$. The path P^v_j is obtained from P^v_{j-1} by a pivot.

Let T^* be the shortest-path tree just before the pivot, and let xy be the dart inserted into T^* by the pivot. Then P^v_{j-1} is the path to v in T^*. Write $P^v_{j-1} = Q_1\ Q_2$ where the end of Q_1 (and the start of Q_2) is y. Let R be the path in T^* to x after $j-1$ pivots. Then $P^v_j = R\ xy\ Q_2$.



Let $S = P^v_i$. By our choice of j, the lifted path $\bar{S}$ cannot cross $\bar{Q}_1$ or $\bar{Q}_2$, so it must cross $\bar{R}$. Write $S = S_1\ S_2$ and $R = R_1\ R_2$ where S_1 and R_1 end at the final crossing of $\bar{S}$ and $\bar{R}$.

First suppose that $\bar{S}$ crosses $\bar{R}$ an odd number of times.

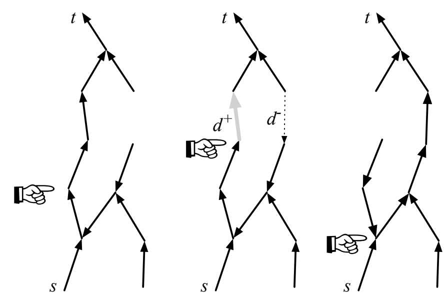


Since S_2 is a shortest path with the same crossing number as $R_2\ xy\ Q_2$, all of its darts have slack zero, and the cycle $R_2\ xy\ Q_2\ \mathrm{rev}(S_2)$ does not enclose the face s of G^*. At least one of the darts of S_2 is not in T^* (or else T^* contains a cycle), so at least one these darts is a proper descendant in T of xy. Therefore the leafmost rule does not select xy to pivot in, which is a contradiction.

Now suppose that $\bar{S}$ crosses $\bar{R}$ an even number of times. By the minimality of j, $\bar{S}$ does not cross $\bar{R}_1 = \bar{P}^z_{j-1}$ where z is the end of R_1. In consequence, however, $\bar{S}$ terminates in a region bounded by parts of $\bar{R}_1$ and $\bar{S}$ and the boundary of the face t that excludes the end of S, which is a contradiction. □

3.3 Fast implementation

The challenge in implementing MaxFlow in linear time is searching the s-to-t path in T to find the first nonresidual dart $\hat{d}$. What makes this challenging is that the s-to-t path changes in each pivot. We use a simple strategy: sequential search with backtracking:



The left diagram shows a state of the tree T at some point in the algorithm. The "finger" points to some vertex on the s-to-t path in T, indicating that the darts on the path from s to that vertex are all residual. The algorithm examines the dart whose tail is indicated by the finger. That dart is also residual, so the algorithm advances the finger by one dart.

The middle diagram shows the next state. Suppose the dart whose tail is indicated by the finger is found to be nonresidual. The algorithm sets d^+ to be that dart, inserts it into T^*, and identifies the dart d^- in T^* that is being replaced by d^+. The reverse of d^- is inserted into T. As shown in the right diagram, maintaining that T is directed towards t requires that the algorithm *reverse* the path from the tail of d^+ to the head of d^-. Finally, the algorithm moves the finger back to the vertex where the new s-to-t path diverged from the old s-to-t path.

On the other hand, if the finger advances all the way to t, the algorithm has discovered that the s-to-t path is residual. In this case, it increases the flow on every dart in this path, and resets the finger to point to s.

The algorithm uses the following simple data structures:

- *to represent the flow assignment:* an array $\mathbf{\Phi}[\cdot]$ indexed by darts;
- *to represent the tree T:* an array succ$[\cdot]$ mapping vertices of G (other than t) to darts (succ$[v]$ is the dart of T whose tail is v);
- *to represent the tree T^*:* an array pred$[\cdot]$ mapping vertices of G^* (other than f^∞) to darts (pred$[f]$ is the dart of T^* whose head in G^* is f);

- *to represent the set of those vertices on the s-to-t path that the finger has visited:* a boolean array visited[·] indexed by vertices;
- *to represent the finger:* a pointer `ptr` to a vertex on the s-to-t path.

We use these data structures to implement MaxFlow. Note that Line 1 can be executed in linear time (e.g., using a bucket queue) since the lengths are small. Lines 7-14 are implemented as follows:

```
repeat
  ptr := s
  while ptr ≠ t,
    if visited[ptr] then exit
    while Φ[succ[ptr]] = c[succ[ptr]],
      visited[ptr] := true
      d⁺ := succ[ptr]
      d⁻ := pred[head_{G*}(d⁺)]
      # Next, reverse the path and reset some visited entries
      ReverseToPtr(head(d⁻))
      succ[head(d⁻)] := rev(d⁻)
    ptr := head(succ(ptr))
  v := s
  while v ≠ t,
    Φ[succ[v]] := Φ[succ[v]] + 1
    Φ[rev(succ[v])] := Φ[rev(succ[v])] − 1
    visited[v] := false
```

which uses a subroutine ReverseToPtr to reverse darts to restore the invariant that T is oriented towards t. This subroutine also resets ptr to point the appropriate vertex of the s-to-t path, and sets visited[v] to false for vertices v that are no longer on this path.

```
def ReverseToPtr(v):
  if v ≠ ptr,
    ReverseToPtr(head(succ[v]))
    succ[head(succ[v])] := rev(succ[v])
  if visited[v],
    ptr := v
    visited[v] := false
```

Remark: the flow values along the s-to-t path could be increased not by 1 but by the minimum residual capacity of the path, which must be at least 1. Up to a constant factor, this does not decrease the worst-case running time, but it might be an improvement in practice.

3.4 Implications of noncrossing paths

Lemma 3.5. *The following statements apply to the evolving shortest path tree of both maximum flow and MSSP. Let i_1, i_2, i_3, i_4 be times (i.e., numbers of pivots) such that either $i_1 < i_2 < i_3 < i_4$ or $i_4 < i_1 < i_2 < i_3$. Let uv be an arbitrary dart.*

1. *If dart uv is right-to-left at times i_1 and i_3, then it is right-to-left at time i_2 or i_4.*
2. *If dart uv is in the shortest path tree at times i_1 and i_3, then it is in the shortest path tree at time i_2 or i_4.*
3. *If dart uv is right-to-left at time i_1 and in the shortest path tree at time i_3, then it is right-to-left or in the shortest path tree at time i_2.*

In consequence, for each edge uv, there are four cyclically contiguous periods, in order: dart uv is right-to-left, dart uv is in the shortest path tree, dart vu is right-to-left, dart vu is in the shortest path tree.

Proof. See Figure 1. For $j \in \{1, 2, 3, 4\}$ and $x \in \{u, v\}$, let P_{jx} be (MSSP) the algorithm's shortest path to vertex x at time i_j in G or (maximum flow) the algorithm's lifted shortest path to vertex x_0 at time i_j in $\bar{G}^*$. Recall that, by Theorems 3.4 and 4.7, for all j, the paths P_{ju} and P_{jv} are noncrossing, and, for all i and j, the paths P_{iu} and P_{ju} are noncrossing, and the paths P_{iv} and P_{jv} are noncrossing.

For claims 1 and 2, assume symmetrically that dart uv is not enclosed by the cycle $\mathrm{rev}(P_{1u})\ P_{3u}$. For claim 3, we prove this unconditionally by showing that the path P_{1u} enters vertex u between vu exclusive and P_{3u} inclusive in counterclockwise order. The path P_{1u} crosses neither the path P_{1v} nor the path P_{3u}. Accordingly, P_{1u} does not cross the cycle $\mathrm{rev}(P_{1v})\ P_{3v} = \mathrm{rev}(P_{1v})\ P_{3u}\ uv$. Since dart uv is right-to-left at time i_1, the path P_{1u} contains a dart belonging to the interior of the cycle, and the conclusion follows.

To show claim 1, assume to the contrary that dart uv is not right-to-left at time i_2. The path P_{2v} does not contain dart uv, as otherwise, it would cross the path P_{3v}. It follows that P_{2v} crosses the path P_{3u}, which implies the contradiction that P_{2v} crosses the path P_{3v}.

To show claim 2, observe that the path P_{2v} crosses neither the path $P_{1v} = P_{1u}\ uv$ nor the path $P_{3v} = P_{3u}\ uv$ and thus contains dart uv.

To show claim 3, assume to the contrary that dart uv is not right-to-left at time i_2 and that P_{2v} does not contain dart uv. It follows that the dart uv belongs to the exterior of the cycle $\mathrm{rev}(P_{1v})\ P_{2v}$ and thus that the path P_{2u} crosses the path P_{1v}. The latter statement in turn implies the contradiction that P_{2u} crosses the path P_{1u}. □

3.5 Running time

Theorem 3.6. *Let C be the total capacity of all darts. The running time of* MaxFlow *is $O(n + C)$.*

Proof. The initialization runs in time $O(n + C)$. By Lemma 3.5, for every dart d, there is at most one time at which d is replaced in T by its reverse, so the total time spent in ReverseToPtr is $O(n)$. Excluding ReverseToPtr, each addition to the s-to-ptr path is accomplished in time $O(1)$, and only ReverseToPtr removes darts from that path. Updating the flow thus dominates the remainder of the running time. Since flow is pushed only on right-to-left edges, it follows again by Lemma 3.5 that the time spent updating the residual capacities of each particular edge is on the order of the sum of the capacities in each direction. □

4. MULTIPLE-SOURCE SHORTEST PATHS

In this section, we describe the multiple-source shortest-paths (MSSP) problem in greater detail, and we describe the abstract algorithm for solving it. Here is an informal specification of MSSP:

- *input:* a directed planar embedded graph G with a designated infinite face f^∞, and a vector $\boldsymbol{c}$ assigning nonnegative lengths to darts.

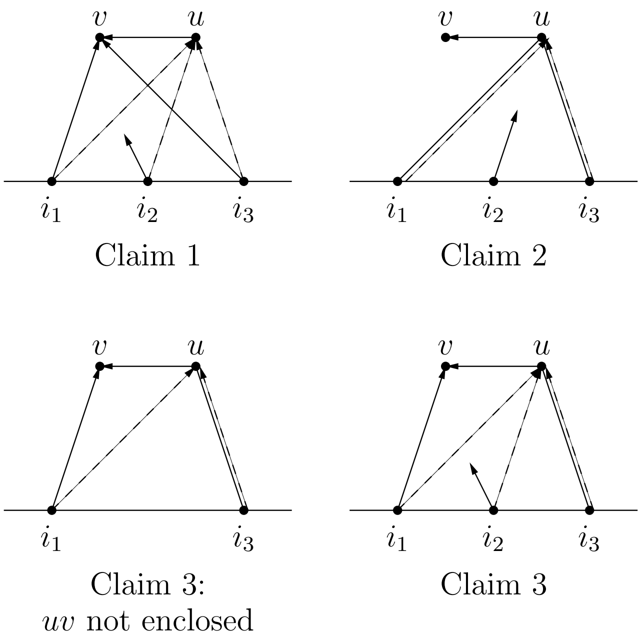


Figure 1. Illustrations of Lemma 3.5. The horizontal line denotes the boundary of the infinite face. The vertices labeled i_1, i_2, i_3 are the roots of the shortest path tree at those times.

- *output:* a representation of the shortest-path trees rooted at the vertices on the boundary of f^∞.

We make this more formal by specifying the representation. Let $d_1 \cdots d_k$ be the counterclockwise cycle of darts forming the boundary of f^∞. Let T_0 be the shortest-path tree rooted at tail(d_1). For $i = 1, \ldots, k$, let T_i be the shortest-path tree rooted at head(d_i). The goal of an algorithm for MSSP is to output the changes required to transform T_0 into T_1, the changes needed to transform T_1 into T_2, ..., and the changes needed to transform T_{k-1} into T_k.

4.1 Pivots

The basic unit of change in a rooted tree T, called a *pivot*, consists of ejecting one dart d_- and inserting another dart d_+ so that the result is again a rooted tree. A pivot is specified by the pair (d_-, d_+) of darts.[2]

Transforming T from T_{i-1} to T_i consists of

- a *special* pivot that ejects the dart whose head is head(d_i), and inserts the dart rev(d_i), after which T is head(d_i)-rooted, and
- a sequence of *ordinary* pivots each of which ejects a dart d^- and inserts a dart d^+ with the same head.

Klein [23] shows that the number of pivots required is $O(n)$ and describes an $O(n \log n)$ algorithm to find them. The algorithm was based on using a *dynamic-tree data structure* to represent the dual T^* of T. In each step, the algorithm used the *leafmost* rule to select the dart d^+ to insert: among all candidates for inserting into T, select one that in the dual tree T^*, rooted at the infinite face, has no descendants that are candidates. Each operation of the dynamic-tree data structure requires $O(\log n)$ time, giving the $O(n \log n)$ bound.

Building on this work, Cabello, Chambers, and Erickson [8, 9] described a simpler $O(n \log n)$ algorithm and generalized it to graphs embedded on higher-genus surfaces. In their analysis, they assume *non-degeneracy*: between each pair of vertices there is a unique shortest path. (They suggest coping with degeneracy by introducing tiny random perturbations.)

The algorithm we present in this paper builds in turn on their approach. We give a method for finding pivots that does not use any sophisticated data structure.

Theorem 4.1. *There is an $O(n+L)$-time algorithm that, given a planar embedded graph with nonnegative integer dart lengths that sum to L, solves the MSSP problem, computing the pivots for all shortest-path trees rooted at the vertices on the boundary of the infinite face.*

The algorithm is obtained from that of Cabello et al. by two modifications. First, because we cannot address degeneracy by introducing tiny perturbations, we must re-introduce a *leafmost* rule in order to cope with degeneracy. Second, a more detailed analysis of the pivots allows us to show that, for darts with integer lengths that are small on average, the dynamic-tree data structure can be eliminated in favor of a much simpler data structure.

4.2 Lower bound for real edge-lengths

The following theorem shows that every $o(n \log n)$-time algorithm for MSSP operates on lengths and distances other than by addition and comparison. The proof technique is to reduce sorting to MSSP.

Theorem 4.2. *Every linear decision tree that computes multiple-source shortest paths has depth $\Omega(n \log n)$.*

Proof. Consider the family depicted in Figure 2 of graphs with parameter π, a permutation on $\{1, \ldots, n\}$. The infinite face has $\sqrt{n}+2$ vertices. For all i, the differences between the shortest path tree with root u_i and the shortest path tree with root u_{i+1} are to exchange, for all $i\sqrt{n} < j \leq (i+1)\sqrt{n}$, dart $u_0 v_{\pi^{-1}(j)}$ for dart $u_{\sqrt{n}} v_{\pi^{-1}(j)}$. Over all possible permutations, there are $N = \binom{n}{\sqrt{n}, \ldots, \sqrt{n}}$ possible outputs, so the decision tree has depth at least $\log N = \Omega(n \log n)$. $\square$

4.3 Computing boundary-to-boundary distances

Wee show that, when the boundary of the infinite face consists of $O(\sqrt{n})$ vertices, our MSSP algorithm can be extended to compute the dense distance graph in linear time.

Theorem 4.3. *There is an algorithm that, given a planar embedded graph with nonnegative integer dart lengths, computes distances between vertices on the boundary of the infinite face. The algorithm runs in $O(n + L + k^2)$ time where L is the sum of all dart lengths and k is the number of vertices on the boundary of the infinite face.*

Since the output size is k^2, the running time when $L = O(n)$ is optimal (to within a constant factor).

[2]The term *pivot* comes from an analogy to the network-simplex algorithm.

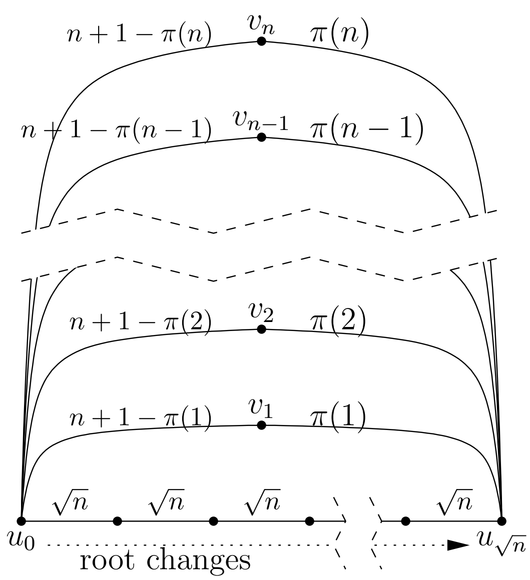


Figure 2. A family of graphs with parameter π, a permutation on $\{1, \ldots, n\}$, on which every linear decision tree for MSSP has depth $\Omega(n \log n)$.

4.4 The MSSP algorithm

We give a high-level description of the MSSP algorithm. In structure it closely resembles that of Cabello et al. The algorithm consists of k iterations, one for each of the darts d_i on the boundary of the infinite face. At the beginning of Iteration i, the shortest-path tree T is rooted at tail(d_i). Let D be the distance in the graph from the tail of d_i to the head. To change the root from the tail of d_i to its head, the algorithm modifies the shortest-path tree T by (a) inserting rev(d_i) with a length of $-D$, and (b) removing the dart whose head is the head of d_i. This is the *special pivot*. The resulting tree is a shortest-path tree rooted at the head of d_i.

Next, the algorithm gradually increases the length of rev(d_i) until it reaches its original length. During this process, ordinary pivots are performed as necessary to maintain the property that the tree T is a shortest-path tree with respect to the current lengths. At the end of the process, T is a shortest-path tree with respect to the original lengths. By *as necessary*, we mean that increasing the length of rev(d_i) without performing a pivot would make T not a shortest-path tree.

Say a dart xy is *active* if the root-to-x path in T does not include rev(d_i) but the root-to-y path does (and $xy \neq$ rev(d_i)).

We use λ to denote the (increasing) length of rev(d_i). Define $\boldsymbol{c}_\lambda$ to be the vector assigning lengths as a function of λ, i.e.,

$$c_\lambda(d) = \begin{cases} \lambda & \text{if } d = \text{rev}(d_i) \\ \boldsymbol{c}[d] & \text{otherwise.} \end{cases}$$

For a length vector $\boldsymbol{\ell}$ and for each vertex v, define $dist_i(v, \boldsymbol{\ell})$ to be the head(d_i)-to-v distance with respect to $\boldsymbol{\ell}$. For each dart d, define the *slack* of dart xy as

$$\text{slack}_i(xy, \boldsymbol{\ell}) = dist_i(x, \boldsymbol{\ell}) + \boldsymbol{\ell}[xy] - dist_i(y, \boldsymbol{\ell}).$$

It follows that, as λ increases, the slack of a dart xy decreases if and only if xy is active.

We use T^* to denote the spanning tree of the planar dual G^* that consists of edges not in T. The following lemma is adapted from Cabello et al.

LEMMA 4.4. *Let f be the face to the right of rev(d_i). The active darts are the darts in the path from f to f^∞ in T^*.*

Now we give a high-level description of the MSSP algorithm.

```
T := tail(d_1)-rooted shortest-path tree for G
for i = 1, ..., k,
1 λ := −1 times the distance from tail(d_i) to head(d_i)
2 remove the dart of T entering head(d_i) and insert rev(d_i)
3 while λ < c[rev(d_i)],
4    while there is an active dart d with slack_i(d, c_λ) = 0,
5       d+ := the leafmost such dart in the dual tree T*
6       remove from T the dart d− whose head is head(d+),
7       insert d+ into T
8    λ := λ + 1
```

This procedure differs somewhat from that of Cabello et al. In their algorithm, the dart d_i is bisected into two and the root is the vertex in between, but this difference is not significant. More significantly, their algorithm finds an active dart d^+ whose slack is minimum, and increases λ to make that dart's slack minimum, then pivots the dart in. Our algorithm, taking advantage of the small weights, iteratively increments λ by one, always pivoting in the leafmost zero-slack dart. These differences do not affect the procedure's correctness.

4.5 Correctness

In this section, we briefly show that, at the end of each iteration of the for-loop, T is a shortest-path tree.

LEMMA 4.5. *Assume that, at the beginning of the procedure's execution, T is a shortest-path tree with respect to $\boldsymbol{c}$. Then throughout the execution, T is a shortest-path tree with respect to $\boldsymbol{c}_\lambda$.*

PROOF. The assumption implies that, after Line 2, T is a shortest-path tree with respect to $\boldsymbol{c}_\lambda$, since the special pivot increases the distance to every vertex by exactly the initial value of λ. In each ordinary pivot, the fact that the entering dart has slack zero shows that the resulting tree remains a shortest-path tree. □

COROLLARY 4.6. *At the end of the procedure, T is a shortest-path tree with respect to $\boldsymbol{c}$.*

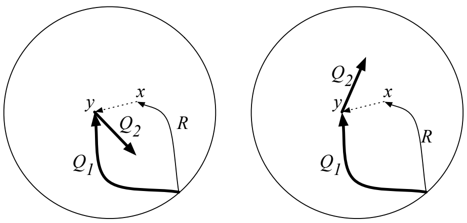
PROOF. The procedure terminates when $\lambda = \boldsymbol{c}[\text{rev}(d_i)]$ so $\boldsymbol{c}_\lambda$ is identical to $\boldsymbol{c}$. □

4.6 Noncrossing

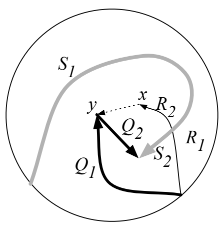
The following theorem, analogous to Theorem 3.4, is the basis for the running-time analysis. Like that theorem, this one is trivial in the absence of degeneracy.

THEOREM 4.7. *Let v be any vertex. For $j = 0, 1, 2, \ldots$, let P_j^v be the root-to-v path in the shortest-path tree T after j pivots. The paths $P_0^v, P_1^v, \ldots$ are mutually noncrossing.*

PROOF. Assume the theorem is not true. Let j be the minimum integer such that, for some vertex v and some integer $i < j$, P_j^v crosses P_i^v. The path P_j^v is obtained from P_{j-1}^v by a pivot. The pivot must be ordinary since a special pivot cannot create a crossing. Let T be the shortest-path tree just before the pivot, and let xy be the dart inserted into T by the pivot. Then P_{j-1}^v is the path to v in T. Write $P_{j-1}^v = Q_1\ Q_2$ where the end of Q_1 (and start of Q_2) is y. Let R be the path in T to x after $j-1$ pivots. Then $P_j^v = R\ xy\ Q_2$.

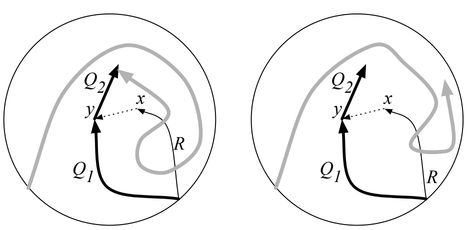


Let $S = P_i^v$. By our choice of j, S cannot cross Q_1 or Q_2, so it must cross R. First suppose S crosses R an odd number of times. Write $S = S_1\ S_2$ where the start of S_2 is the last vertex of S on R.



Then S_2 is a shortest path, so all of its darts have slack zero. Write $R = R_1\ R_2$ where the end of R_1 is the start of S_2. At least one of the darts of S_2 is not in T (or else T contains a cycle). Therefore at least one these darts is a proper descendant in T^* of xy. Therefore the leafmost rule does not select xy to pivot in, which is a contradiction.

Now suppose S crosses R an even number of times. Write $R = R_1\ R_2$, where the end of R_1 is the final crossing of S over R. One of the previous crossings must be an internal vertex of R_1,



or else there is no way for R to reach v without crossing itself. Therefore R_1 is a contradiction to the choice of j. $\square$

4.7 Implementation of pivot-finding

The challenge in implementing the MSSP algorithm is in finding the leafmost active dart with zero slack. The strategy we use is exactly analogous to that for max st-flow. The algorithm searches up the path in T^*, searching for a dart with zero slack, using a finger to keep track of which vertex it has reached. When it finds a zero-slack dart, it performs a pivot, reverses the orientation of darts to maintain that T^* is oriented towards the root f^∞, and resets the finger to the rootmost vertex on the path such that all darts on the path leafward of that vertex are known to have nonzero slack. The implementation uses the following simple data structures:

- *to represent the shortest-path tree T:* an array pred[·] mapping vertices of G (other than T's root) to darts (pred[f] is the dart of T whose head in G is f)
- *to represent the dual spanning tree T^*:* an array succ[·] mapping vertices of G^* (other than f^∞) to darts (succ[v] is the dart of T^* whose tail is v);
- *to represent the dart slacks:* an array slack[·] mapping darts to integers;
- *to represent the set of those vertices on the s-to-t path that the finger has visited:* a boolean array visited[·] indexed by vertices;
- *to represent the finger:* a pointer ptr to a vertex on the path.

4.8 Running time

The proof of the following theorem is analogous to that of Theorem 3.6.

THEOREM 4.8. *Let L be the total length of all darts. The running time of* MSSP *is $O(n + L)$.*

4.9 Implementation of distance-finding

Now we prove Theorem 4.3. Recall that, for every dart xy and iteration i, we define

$$\mathrm{slack}_i(xy, \boldsymbol{\ell}) = dist_i(x, \boldsymbol{\ell}) + \boldsymbol{\ell}[xy] - dist_i(y, \boldsymbol{\ell}).$$

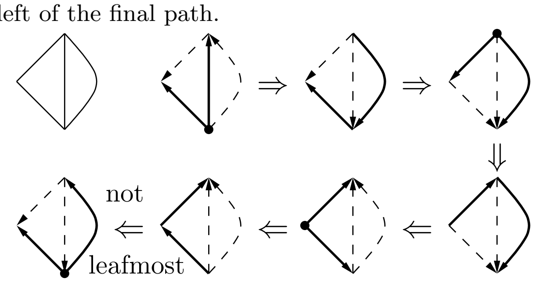
By solving for $dist_i(y, \boldsymbol{\ell})$, we obtain the equation

$$dist_i(y, \boldsymbol{\ell}) = dist_i(x, \boldsymbol{\ell}) + \boldsymbol{\ell}[xy] - \mathrm{slack}_i(xy, \boldsymbol{\ell}).$$

To obtain distances from the root to the other $k-1$ vertices on the boundary of the infinite face in time $O(k)$, compute the cumulative sums of $\boldsymbol{\ell}[xy] - \mathrm{slack}_i(xy, \boldsymbol{\ell})$ for darts xy in order on the oriented boundary of the infinite face.

4.10 Necessity of leafmost pivots

Theorem 4.7 does not hold without the leafmost selection rule. Here is an execution of MSSP where the leafmost rule is not obeyed and the initial path into the top vertex is to the left of the final path.



References

[1] Glencora Borradaile, Erik Demaine, and Siamak Tazari. Polynomial-time approximation schemes for subset-connectivity problems in bounded-genus graphs. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science (STACS)*, 2009.

[2] Glencora Borradaile and Philip N. Klein. The two-edge connectivity survivable network problem in planar graphs. In *Proceedings of the Thirty-fifth International Colloquium on Automata, Languages and Programming*, 2008.

[3] Glencora Borradaile and Philip N. Klein. An $O(n \log n)$ algorithm for maximum *st*-flow in a directed planar graph. *Journal of the ACM*, 56(2), 2009.

[4] Glencora Borradaile, Philip N. Klein, and Claire Mathieu. A polynomial-time approximation scheme for steiner tree in planar graphs. *ACM Transactions on Algorithms*, 5, 2009. Special Issue on SODA 2007.

[5] Glencora Borradaile, Philip N. Klein, Shay Mozes, Yahav Nussbaum, and Christian Wulff-Nilsen. Multiple-source multiple-sink maximum flow in directed planar graphs in near-linear time. In *Proceedings of the 52nd Annual IEEE Symposium on Foundations of Computer Science*, pages 170–179, 2011.

[6] Ulrik Brandes and Dorothea Wagner. A linear time algorithm for the arc disjoint menger problem in planar directed graphs. *Algorithmica*, 28(1):16–36, 2000.

[7] S. Cabello. Many distances in planar graphs. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1213–1220, 2006.

[8] Sergio Cabello and Erin W. Chambers. Multiple source shortest paths in a genus g graph. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 89–97, 2007.

[9] Sergio Cabello, Erin W. Chambers, and Jeff Erickson. Multiple-source shortest paths in embedded graphs. *CoRR*, abs/1202.0314, 2012.

[10] Erin W. Chambers, Jeff Erickson, and Amir Nayyeri. Homology flows, cohomology cuts. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing*, pages 273–282, 2009.

[11] Hsien-Chih Chang and Hsueh-I Lu. Computing the girth of a planar graph in linear time. In *Proceedings of the 17th Annual International Conference on Computing and Combinatorics (COCOON)*, pages 225–236, 2011.

[12] David Eisenstat, Philip N. Klein, and Claire Mathieu. An efficient polynomial-time approximation scheme for Steiner forest in planar graphs. In *Proceedings of the 23nd ACM-SIAM Symposium On Discrete Algorithms*, pages 626–638, 2012.

[13] Jeff Erickson. Maximum flows and parametric shortest paths in planar graphs. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 794–804, 2010.

[14] Jeff Erickson and Amir Nayyeri. Computing replacement paths in surface embedded graphs. In *Proceedings of the 22nd ACM-SIAM Symposium On Discrete Algorithms*, pages 1347–1354, 2011.

[15] 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.

[16] G. Frederickson. Fast algorithms for shortest paths in planar graphs with applications. *SIAM Journal on Computing*, 16:1004–1022, 1987.

[17] Harold N. Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. *J. Comput. Syst. Sci.*, 30(2):209–221, 1985.

[18] R. Hassin. Maximum flow in (s, t) planar networks. *Information Processing Letters*, 13:107, 1981.

[19] M. R. Henzinger, Philip 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.

[20] Giuseppe F. Italiano, Yahav Nussbaum, Piotr Sankowski, and Christian Wulff-Nilsen. Improved algorithms for min cut and max flow in undirected planar graphs. In *Proceedings of the 43rd ACM Symposium on Theory of Computing*, pages 313–322, 2011.

[21] Haim Kaplan and Yahav Nussbaum. Maximum flow in directed planar graphs with vertex capacities. *Algorithmica*, 61(1):174–189, 2011.

[22] Ken-ichi Kawarabayashi, Philip N. Klein, and Christian Sommer. Linear-space approximate distance oracles for planar, bounded-genus, and minor-free graphs. In *Proceedings of the 38th International Colloquium on Automata, Languages and Programming, ICALP 2011*, pages 135–146, 2011.

[23] Philip N. Klein. Multiple-source shortest paths in planar graphs. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium On Discrete Algorithms*, pages 146–155, 2005.

[24] Philip N. Klein. A subset spanner for planar graphs, with application to subset TSP. In *Proceedings of the 38th Annual ACM Symposium on Theory of Computing*, pages 749–756, 2006.

[25] Philip N. Klein and Shay Mozes. Multiple-source single-sink maximum flow in directed planar graphs in $O(\text{diameter } n \log n)$ time. In *Proceedings of the 12th Algorithms and Data Structures Symposium, WADS 2011*, pages 571–582, 2011.

[26] Philip N. Klein, Shay Mozes, and Oren Weimann. Shortest paths in directed planar graphs with negative lengths: A linear-space $O(n \log^2 n)$-time algorithm. *ACM Trans. Algorithms*, 6(2):1–18, 2010.

[27] Jakub Lacki, Yahav Nussbaum, Piotr Sankowski, and Christian Wulff-Nilsen. Single source - all sinks max flows in planar digraphs. In *Proceedings of the 53rd Annual IEEE Symposium on Foundations of Computer Science*, 2012.

[28] Shay Mozes and Christian Sommer. Exact distance oracles for planar graphs. In *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 209–222, 2012.

[29] Shay Mozes and Christian Wulff-Nilsen. Shortest paths in planar graphs with real lengths in $O(n \log^2 n/ \log \log n)$ time. In *Proceedings of the 18th Annual European Symposium on Algorithms*, pages 206–217, 2010.

[30] H. Ripphausen-Lipa, D. Wagner, and K. Weihe. The vertex-disjoint Menger problem in planar graphs. *SIAM J. Comput.*, 24(5):1002–1017, 1995.

[31] F. R. Schmidt, E. Toeppe, and D. Cremers. Efficient planar graph cuts with applications in computer vision. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Miami, Florida, June 2009.

[32] A. Schrijver. On the history of the transportation and maximum flow problems. *Mathematical Programming*, 91(3):437–445, 2002.

[33] Karsten Weihe. Edge-disjoint (s, t)-paths in undirected planar graphs in linear time. *J. Algorithms*, 23(1):121–138, 1997.

[34] Christian Wulff-Nilsen. Solving the replacement paths problem for planar directed graphs in $O(n \log n)$ time. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 756–765, 2010.