Faster Shortest-Path Algorithms for Planar Graphs

Philip Klein[1]
Satish Rao[2]
Monika Rauch[3]
Sairam Subramanian[4]

Technical Report No. CS-94-12

March, 1994

[1]Brown University,Computer Science Department Box 1910, Providence, RI 02912
[2]NEC Research Institute
[3]Cornell University
[4]Brown University, Computer Science Department Box 1910, Providence, RI 02912

Faster shortest-path algorithms for planar graphs

to appear in Proceedings of STOC 94

Philip Klein[*] Brown University

Satish Rao NEC Research Institute

Monika Rauch[†] Cornell University

Sairam Subramanian[*] Brown University

Abstract

We give a linear-time algorithm for single-source shortest paths in planar graphs with nonnegative edge-lengths. Our algorithm also yields a linear-time algorithm for maximum flow in a planar graph with the source and sink on the same face. The previous best algorithms for these problems required $\Omega(n\sqrt{\log n})$ time where n is the number of nodes in the input graph.

For the case where negative edge-lengths are allowed, we give an algorithm requiring $O(n^{4/3}\log nL)$ time, where L is the absolute value of the most negative length. Previous algorithms for shortest paths with negative edge-lengths required $\Omega(n^{3/2})$ time. Our shortest-path algorithm yields an $O(n^{4/3}\log n)$-time algorithm for finding a perfect matching in a planar bipartite graph. A similar improvement is obtained for maximum flow in a directed planar graph.

1 Introduction

Computing shortest paths is a fundamental and ubiquitous problem in network analysis. Aside from the importance of this problem in its own right, often the problem arises in the solution of other problems (e.g. network flow and matching). In this paper we give improved algorithms for single-source shortest paths in planar networks.

The first algorithm handles the important special case of nonnegative edge-lengths. For this case, we obtain a linear-time algorithm. Thus our algorithm is optimal, up to constant factors. No linear-time algorithm for shortest paths in planar graphs was previously known. For general graphs the best bounds known in the standard model, which forbids bit-manipulation of the lengths, is $O(m+n\log n)$ time, due to Fredman and Tarjan [9].[1] For planar graphs, Frederickson [7] pioneered the use of *separators* to obtain faster shortest-path algorithms. His algorithm, the best known previously, runs in $O(n\sqrt{\log n})$ time on planar graphs. It depends on the fact that planar graphs have size-$O(\sqrt{n})$ separators.

One interesting aspect of our shortest-path algorithm for nonnegative edge-lengths is that, although it uses separators, they are not required to have size $O(\sqrt{n})$. For example, it is sufficient to use separators of size $O(n^{1-\epsilon})$. Thus our algorithm can in principle be applied to a much broader class of graphs than just planar graphs. Of course, in order for the whole computation to take only linear time, it must be possible to find the r-division in linear time. If this is not the case, our algorithm may still be useful if many shortest path computations are performed on the same graph, for in this case it may be worthwhile to precompute the decomposition. This is the case in various algorithms, e.g., approximate multicommodity flow computations.

The second algorithm handles negative edge-lengths. We obtain an algorithm that takes time $O(n^{4/3}\log nL)$, where the lengths are integers greater than $-L$. For general graphs, the best bound known is $O(n^{1/2}m\log L)$ time, due to Goldberg [14] which yields $O(n^{3/2}\log L)$ time on sparse (e.g. planar) graphs. For undirected planar graphs, Pan and Reif [25,26] showed how to achieve $O(n^{3/2})$ time using separators. Cohen [4] showed how to achieve the same bound for directed planar graphs. Previously no algorithm handling negative lengths was known that ran faster than $O(n^{3/2})$. Our algorithm overcomes this apparent barrier, improving the time by a (fractional) polynomial factor when the negative lengths are not

*Research supported by NSF PYI award CCR-9157620, together with PYI matching funds from Honeywell Corporation, Thinking Machines Corporation and Xerox Corporation. Additional support provided by ARPA contract N00014-91-J-4052 ARPA Order No. 8225.

†Research supported by ONR/DARPA grant N00014-92-J-1989 Automatic Generation of Engineering Analysis

[1] In less restricted models, bounds of $O(m+n\log n/\log\log n)$ and $O(m+n\sqrt{\log C})$ can be achieved [1,8]. Here m is the number of edges, n is the number of nodes, and C is the maximum magnitude of an edge-length assuming edge-lengths are integers.

too large in magnitude.

For planar graphs, shortest-path computation is closely related to network flow. Hassin [17] has shown that if a source s and a sink t are located on the same face of a planar graph, then a maximum st-flow can be found by computing single-source shortest-paths in the planar dual. Thus using our linear-time algorithm, one obtains a linear-time algorithm for maximum st-flow in this case.

In the case when s and t are not on the same face and the graph is directed, Miller and Naor [21] (see also Johnson and Venkatesan [19]) show how to solve max-flow by computing single-source shortest-path computation with negative lengths. They use this approach to find a maximum flow. They also use the approach in solving the following problem: given multiple sources and multiple sinks, each with a specified supply or demand, find a feasible flow in the network. Bipartite perfect matching in a planar graph, for example, can be formulated in this way. The previous best algorithms for all of these problems required $\Omega(n^{3/2})$ time. Our shortest-path algorithm thus yields polynomial-factor improvements for these problems. In particular, we obtain an $O(n^{4/3} \log n)$-time algorithm for planar bipartite perfect matching. We obtain an $O(n^{4/3} \log n \log C)$-time algorithm for maximum flow, where C is the sum of (integral) capacities.

The approach used in obtaining the shortest-path algorithm for arbitrary lengths also enables us to obtain parallel and dynamic algorithms for this problem and for max-flow and matching.

The key to both our shortest-path algorithms is our use of graph-decompositions based on separators. Lipton and Tarjan showed [23] that given an n-node planar graph one can in linear time find a set of nodes of size $O(\sqrt{n})$ whose removal breaks the graph into pieces each of size at most $\frac{2}{3}n$. Based on this result, Frederickson [7] developed the notion of an r-*division* of a graph, a division of the graph into regions of size $\Theta(r)$ with boundaries of size $O(\sqrt{r})$. Frederickson showed that an r-division could be found in $O(n \log n)$ time by recursive application of the separator-algorithm of Lipton and Tarjan. Goodrich [15] has recently given a linear-time algorithm to find a separator decomposition in a planar graph. His method yields a linear-time algorithm to find an r-division.

The algorithm for nonnegative edge-lengths starts with a recursive version of an r-division, i.e. an r-division each of whose regions has an r'-division (for r' much smaller than r), and so on. The algorithm maintains priority queues for each of the regions. It performs steps analogous to those of Dijkstra's algorithm for shortest paths, but moves between the queues of different regions in a way that takes into account the fact that operations on queues associated with larger regions are more expensive than those for queues associated with smaller regions (because the former queues have more elements). The labels thus assigned to nodes within a region are not guaranteed to be accurate because the labels of the boundary nodes of the region may not be accurate. Thus the movement between different regions must be carefully orchestrated to make progress towards obtaining accurate distance labels while at the same time ensuring that on average enough small-queue operations can be done per large-queue operation. Note that other problems such as bottleneck shortest path that can be solved using slight variants of Dijkstra's algorithm can also be solved in linear time using our approach.

The algorithm for arbitrary lengths first applies the algorithm of Cohen [4] to each region, obtaining (1) shortest-path distances between each pair of boundary nodes of the region and (2) an augmented version of the region that facilitates computing shortest-paths in the region. For each region, the algorithm constructs a complete directed graph on the boundary nodes, where the length of an edge from u to v is defined to be the distance from u to v within the region. The algorithm then applies the algorithm of Goldberg to the union of these complete graphs, obtaining distances from the source to each of the boundary nodes. Finally, the algorithm operates once again on each region, using the distances computed for the boundary nodes together with the augmented version of the region to compute distances for all the nodes in the region.

To obtain a parallel algorithm, we simply use a parallel algorithm to carry out each step. Finding the r-division can be done by repeated application of the parallel planar-separator algorithm of Gazit and Miller [12]. Cohen gives a parallel version of her algorithm. To compute shortest paths on the union of complete graphs, we use not Goldberg's algorithm, which does not directly parallelize, but Gabow and Tarjan's parallel algorithm for the assignment problem; there is an efficient reduction from single-source shortest-paths to the assignment problem. We obtain a parallel algorithm requiring $O(n^{2/3} \log nL \log^3 n)$ time and $O(n^{4/3} \log nL \log n)$ work, where L is the maximum magnitude of lengths. The same bounds hold for perfect matching; the bounds are higher by a logarithmic factor for max flow.

To obtain a dynamic algorithm, we use an approach used previously for dynamically approximating shortest paths in planar undirected graphs [20], an approach based in turn on that used in dynamic algorithms for a variety of problems in planar graphs [6, 10,11,27]. To compute the shortest path from a given source to a given sink, one operates on the union of complete graphs with two of the complete graphs replaced by the regions they represent, one for the source and one for the sink. To update the cost of an edge, for example, one recomputes the complete graph for the region containing the edge. Some difficulties arise in handling addition of edges to the network; the solution involves bounding the time per addition only in an amortized sense. The time per operation is $O(n^{9/7} \log nL)$.

2 Graph decompositions

For a graph G we use $V(G)$ to denote the node-set of G and $E(G)$ to denote the edge-set. For a graph G and a node-subset S, an *S-balanced node-separator* is a set of nodes whose removal breaks G into two pieces such that each piece contains at most an α fraction of the nodes of S (for some suitable constant $1/2 \leq \alpha < 1$). The *size* of the separator is the number of nodes it contains. A class of graphs has an *f-separator theorem* if any k-node member of the class has a size-$f(k)$ separator. Examples of classes for which such theorems have been proved are planar graphs [23], bounded-genus graphs [13], two-dimensional overlap graphs [22], and graphs excluding a fixed graph as a minor [2].

Separators can be used to find a *division* [6] of a graph, a partition of the edge-set into subsets, called *regions*. A node is said to be *contained* in a region if some edge of the region is incident to the node. A node contained in more than one region is called a *boundary node* of the regions containing it. An (r, s)-division of an n-node graph is a division into $O(n/r)$ regions, each containing at most r nodes including $O(s)$ boundary nodes.

Theorem 2.1 [Frederickson] Any class of graphs with an f-separator theorem has an $(r, f(r))$-division for any r.

If we repeatedly divide the regions of an (r, s)-division to get smaller and smaller regions, we get a recursive division. More formally, for integer nondecreasing functions g and f where $g(k) \leq k$ for all k, a *(g, f)-recursive division* of an n-node graph G is defined recursively as follows. The recursive division contains one region R_G consisting of all of G. If G has more than one edge then in addition the recursive division contains a (g, f)-recursive division of each region of a $(g(n), f(g(n)))$-division of G. Our shortest-path algorithm for the case when edges have nonnegative weights makes use of a *(g, f)-recursive division* for suitable values of g and f.

The regions of the $(g(n), f(g(n)))$-division are called the *children* of R_G, and R_G is called their *parent*. The *ancestors* and *descendants* of a region are defined analogously. We also use the term *subregion* to mean descendant. A region with only one edge has no subregions and is hence called an *atomic region*. For an edge xy, the atomic region consisting of xy is denoted $R(xy)$.

The *level* of an atomic region is zero, and the level of a nonatomic region is one more than the maximum level of its immediate subregions. We use parent(R) to denote the parent of a region R, and we use $\ell(R)$ to denote its level.

Theorem 2.2 Any class of graphs with an f-separator theorem has a (g, f_c)-division for any g where $g(k) = o(k)$.

3 Shortest paths with nonnegative edge-lengths

For our shortest-path algorithm, we assume without loss of generality that the input graph G is directed, that each node has at most two incoming and outgoing edges, and that there is a finite-length path from s to each node. The algorithm maintains a label $d(v)$ for each node v of G. For each region of the recursive division of G, the algorithm maintains a priority queue $Q(R)$. If R is nonatomic, the items stored in $Q(R)$ are the immediate subregions of R. If R is an atomic region, $Q(R)$ consists of only one item, the single edge contained in R; in this case the key associated with the edge is either the label of the tail of the edge or infinity, depending on whether the edge needs processing.

We assume the priority queue Q supports the operations

- $updateKey(Q, x, k)$, which updates the key of x to k,
- $minItem(Q)$, which returns the item in Q with the minimum key, and
- $minKey(Q)$, which returns the key associated with $minItem(Q)$

The effect of removing an item from the queue is achieved by updating the item's key to infinity.

For each level ℓ of the recursive division, we have an associated parameter α_ℓ. We determine the values of these parameters later.

The algorithm is based on the following two procedures.

Activate(R)
Comment: R is a region.

A1 If R contains a single edge uv then

A2 if $d(v) > d(u) + \mathrm{length}(uv)$ then set $d(v) := d(u) + \mathrm{length}(uv)$ and, for each outgoing edge vw of v, call GlobalUpdate($R(vw), vw, d(v)$).

A3 $updateKey(Q(R), uv, \infty)$.

A4 Return $minKey(Q(R))$.

A5 Else (R is nonatomic)

A6 Repeat $\alpha_{\ell(R)}$ times or until $minKey(Q(R))$ is infinity:

A7 Let $R' := minItem(Q(R))$.

A8 Let k := Activate(R'), and call $updateKey(Q(R), R', k)$.

A9 Return $minKey(Q(R))$.

GlobalUpdate(R, x, k)
Comment: R is a region, x is an item of $Q(R)$, and k is a key value.

G1 $updateKey(Q(R), x, k)$

G2 If the $updateKey$ operation reduced the value of $minKey(Q(R))$ then

G3 GlobalUpdate(parent(R), R, k).

To compute shortest paths from a source s, proceed as follows. Initialize all labels and keys to infinity. Then assign the label $d(s) := 0$, and for each outgoing edge sw, call GlobalUpdate($R(sw), sw, 0$). Then call Activate(R_G), where R_G is the region consisting of all of G. When it returns, the labels $d(v)$ are the shortest distances from s, as we prove in the next subsection.

3.1 Correctness

We say an edge vw is *relaxed* if $d(v) \leq d(w) + \mathrm{length}(vw)$. It is well-known that if

1. $d(s) = 0$,
2. every label $d(v)$ is an upper bound on the distance from s to v, and
3. every edge is relaxed,

then the labels give correct distances. The algorithm immediately assigns $d(s) := 0$ and never changes that label, so the first condition holds. We show that the second condition also holds throughout the algorithm and that the third condition holds if the algorithm terminates. In the next subsection, we bound the time until termination.

Lemma 3.1 For each node v, throughout the algorithm the label $d(v)$ is an upper bound on the distance from s to v.

Proof: By induction on the number of steps of the algorithm that have been executed. Initially every label except that of s is infinity. We only change labels in step A2. Assuming inductively that $d(u)$ and the old value of $d(v)$ are upperbounds on the distance to u and v, it follows that the new value of $d(v)$ is also an upper bound. □

We say that an edge uv is *pending* if the key of uv in $Q(R(uv))$ is finite.

Lemma 3.2 If an edge uv is not pending then it is relaxed.

Proof: The lemma holds before the first call to Activate since at that point every node but s has label infinity, and every outgoing edge of s is pending. The algorithm only changes an edge uv to not pending, i.e., uv is assigned a key of ∞ in step A3, just after the edge is relaxed.

An edge vw could become unrelaxed when the labels of its endpoints change. Note that labels never go up. The label of v might go down in step A2, but in the same step the algorithm calls GlobalUpdate($R(vw), vw, d(v)$) for each outgoing edge vw of v. In step G1 of GlobalUpdate, the key of vw is updated to a finite value, so vw is again pending. □

Lemma 3.3 The key of a pending edge vw is $d(v)$.

Proof: Initially all labels and keys are ∞. Whenever a label $d(v)$ is assigned a value k (either in the initialization, where $v = s$, or in step A2), GlobalUpdate($R(vw), vw, k$) is called for each outgoing edge vw. The first step of GlobalUpdate(R, v, k) is to update the key of vw to k. □

Next we show that the queues are "consistent," in a sense to become apparent. During the execution of the algorithm, the most recent invocation of Activate that is still active is called the *current invocation*, and the region to which that invocation was applied is called the *current region*.

Lemma 3.4 For any region R that is not an ancestor of the current region, the key associated with R in $Q(\mathrm{parent}(R))$ is the min key of $Q(R)$.

Proof: At the very beginning of the algorithm, all keys are infinity. Thus in this case the lemma holds trivially. Every time the minimum key of some queue

$Q(R)$ is changed in step G1, a recursive call to GlobalUpdate in step G3 ensures that the key associated with R in $Q(\text{parent}(R))$ is updated.

We must also consider the moment when a new region becomes the current region. This happens upon invocation of the procedure Activate, and upon return from Activate. When Activate(R) is newly invoked, the new current region R is a child of the old current region, so Lemma 3.4 applies to even fewer regions than before; hence we know it continues to hold. When Activate(R) returns, the parent of the previous current region R becomes current. Hence at that point Lemma 3.4 applies to R. Note, however, that Activate(R) returns the value $minKey(Q(R))$, and the calling invocation, which is Activate(parent(R)), updates the key of R in $Q(\text{parent}(R))$ to this returned value. □

Corollary 3.5 For any region R that is not an ancestor of the current region,

$$minKey(Q(R)) = \min\{d(v) \; : \; vw \text{ is a pending edge contained in } R\} \quad (1)$$

Proof: By induction on the level of R, using Lemma 3.4. □

Recall that the algorithm terminates when Activate(R_G) returns, where R_G consists of the whole graph. We shall assign infinity to the parameter α_i associated with this invocation, so Activate(R_G) returns only when $minKey(Q(R_G)) = \infty$. It follows from Corollary 3.5 that at that point no edge is pending, so by Lemma 3.2 every edge is relaxed. Hence the labels $d(v)$ give shortest-path distances.

3.2 Analysis

In this subsection, we prove a bound on the running time of the algorithm. The bound depends on the quality of the recursive division and on the parameters α_ℓ used in the algorithm. We show that for, e.g., planar graphs, the running time is linear.

Consider an invocation A of the procedure Activate.[2] The *region* of A is the region R to which Activate is applied. The *level* of A is the level of R. The *start key* of A is the min key of $Q(R)$ just before the invocation begins. The node v achieving the minimum in (1) at this moment is called the *start node* of A. We say that A *starts with* v.

The *end key* of A is the min key of $Q(R)$ just after the invocation ends. We denote the start key of A by start(A), and the end key of A by end(A).

If invocation B is the result of executing step A8 during invocation A, then B is called the *child* of A. The children of an invocation A are *ranked* according to when they occur in time. The *parent* of an invocation, and the *descendants* and *ancestors* of an invocation are similarly defined.

[2] Henceforth, unless otherwise stated, an "invocation" refers to an invocation of Activate.

We define the partial order $\leq$ on the set of invocations as follows: $A \leq B$ if A and B have the same region R, and A occurs before B. We say A is B's immediate predecessor if $A < B$ and there is no invocation C with region R such that $A < C < B$.

Lemma 3.6 Consider a invocation A with region R, and a child A' of A. At the time when A' starts, $minKey(Q(R))$ is the start key of A'.

Proof: Let R' be the region of A'. By choice of R' in step A7, R' is the item in $Q(R)$ whose key is minimum. Furthermore, by Lemma 3.4, the key of R' is the start key of A'. □

Lemma 3.7
1. For an invocation A, every key assigned during A has value at least the start key of A, and
2. If $A_1, \ldots, A_p$ are the children of A ordered by rank, then start(A) = start(A_1),

$$\text{start}(A_1) \leq \text{start}(A_2) \leq \cdots \leq \text{start}(A_p)$$

and start(A_p) $\leq$ end(A).

Proof: By induction on the level of A. If the region of A is $R(uv)$ for some edge uv, then the start key of A is $d(u)$. If GlobalUpdate is called in step A2, it is called with a key $k = d(u) + \text{length}(vw)$, which is at least $d(u)$ by the nonnegativity of the edge-lengths. Thus every key assigned during the call to GlobalUpdate is at least the start key of A. The only other key assigned during A is ∞, which is assigned in step A3.

Suppose now that the argument of A is a nonatomic region R. Let $A_1, \ldots, A_p$ be the children of A. For $i = 1, \ldots, k$, let k_i be the min key of $Q(R)$ at the time when A_i starts, and let k_{p+1} be the min key of $Q(R)$ when A_p ends. Since A_1 starts just after A starts, k_1 is the start key of A. By Lemma 3.6, k_i is the start key of A_i. By definition, k_{p+1} is the end key of A.

We show that $k_{i+1} \geq k_i$ for $i = 1, \ldots, p$, which proves Statement 2. By Statement 1 of the inductive hypothesis, every key assigned during A_i has value at least k_i. In particular, every key assigned to an item of $Q(R)$ has value at least k_i, so in particular the min key of $Q(R)$ at the end of A_i, which is k_{i+1}, does not go below k_i. We have proved Statement 2. We have shown moreover that every key assigned during the children of A has value at least k_1, proving Statement 1 holds for A. This completes the induction step. □

We say a boundary node v of a region R is an *entry node* of R if v has an outgoing edge in R. We also define the source node s to be an entry node of each of the regions containing it.

Lemma 3.8 Let k_0 be the key associated with R at some time t, and let A be the first invocation with region R occurring after time t. If start$(A) < k_0$ then A starts with an entry node.

Proof: Let v be the start node of A. Since start$(A) < k_0$, it follows from Corollary 3.5 and Lemma 3.3 that the key of some edge vw in R must have been lowered to less than k_0 by an *updateKey* operation between time t and the time A started. During this time no invocation acted on the region R or its subregions. The *updateKey* operation must have been called by an invocation GlobalUpdate$(R(vw), vw, k)$ occurring either during the initialization of the algorithm or during an invocation of Activate with an atomic region $R(uv)$.

In the former case, $v = s$. In the latter case, since $R(uv)$ is not a subregion of R, v must be a boundary node of R. Hence in either case v is an entry node of R. □

We immediately obtain two corollaries.

Corollary 3.9 Let A and B be invocations with region R such that A is B's immediate predecessor. If end$(A) >$ start(B) then B starts with a entry node of R.

Corollary 3.10 For each region R, the first invocation with region R starts with an entry node of R.

Lemma 3.11 If $A < B$ are two invocations with the same start node then start$(A) >$ start(B).

Proof: Let v be the common start node. By Corollary 3.5, the start key of A is the label of v at the time A begins. The first atomic regions to be processed during A are those for outgoing edges of v. After these are processed, the keys associated with these atomic regions are set to infinity in step A3. Step A2 ensures that these keys are not further updated unless there is a reduction in the label of v. Since the start node of B is v, it follows by Corollary 3.5 that some outgoing edge of v is once again pending, so such a reduction in the label of v must have taken place before B begins. Since the start key of B is the label of v, it follows that start$(A) >$ start(B). □

We say an invocation A is *stable* if, for every $B > A$, the start key of B is at least the start key of A. Corollary 3.9 essentially shows that the source of instability is start nodes that are entry nodes.

For an invocation A, let *entryPred*(A) be the latest invocation $E \leq A$ such that E starts with an entry node. It follows by Corollary 3.10 that there is such an invocation E. Note that *entryPred*(A) is a predecessor of A (not necessarily a proper predecessor) and hence has the same region as A.

Lemma 3.12 For every invocation A, start$(A) \geq$ start(*entryPred*(A)).

Proof: Let $A_0 =$ *entryPred*(A), and let $A_0 < A_1 < \cdots < A_p = A$ be the invocations with the same region as A that occur between A_0 and A. By definition of *entryPred*, none of of $A_1, \ldots, A_p$ starts with an entry node, so, by Corollary 3.9, start$(A_0) \leq$ start$(A_1) \leq \cdots \leq$ start(A). □

Lemma 3.13 Suppose $A < B$ are two invocations such that *entryPred*$(A) =$ *entryPred*(B). If B is stable then every child of A is stable.

Proof: Let $A = C_0 < C_1 < \ldots < C_p = B$ be the invocations with region R that occur between A and B. Since *entryPred*$(A) =$ *entryPred*(B), the start nodes of $C_1, \ldots, C_p$ are not entry nodes of R. Hence it follows by Corollary 3.9 that end$(C_{i-1}) \leq$ start(C_i) for $i = 1, \ldots, p$. Moreover, it follows from Statement 2 of Lemma 3.7 that start$(C_i) \leq$ end(C_i) for $i = 1, \ldots, p$. We infer that

$$\text{end}(A) \leq \text{start}(C_i) \tag{2}$$

for $i = 1, \ldots, p$.

Let A' be a child of A, and let C' be any invocation such that $C' > A'$. Our goal is to show

$$\text{start}(A') \leq \text{start}(C') \tag{3}$$

Let C be the parent invocation of C' (so the region of C' is R). If $C = A$, then (3) follows from Statement 1 of Lemma 3.7.

Assume therefore that $C > A$. It follows from Statement 1 of Lemma 3.7 that start$(A') \leq$ end(A) and that start$(C) \leq$ start(C'), so it suffices to show end$(A) \leq$ start(C). There are two cases.

Case 1: $C = C_i$ for some $i > 1$. In this case, end$(A) \leq$ start(C) by (2).

Case 2: $C > B$. In this case, end$(A) \leq$ start(B) follows by (2), and start$(B) \leq$ start(C) follows by the stability of B, so we obtain end$(A) \leq$ start(C). □

Recall that R_G is the region consisting of the whole graph.

For an invocation A, define *stableAnc*(A) to be the lowest stable ancestor of A. Since there is only one invocation of Activate whose region is R_G, the region consisting of the whole graph, that activation is trivially stable. Hence every invocation has a stable ancestor, so *stableAnc*(A) is well-defined.

We define an invocation A to be *truncated* if $\mathrm{end}(A)$ is infinity. Our first goal is to upper-bound the time spent in truncated invocations. We will then bound the amount of time spent in nontruncated invocations. We leave for later the time spent in the procedure GlobalUpdate.

Define the *chargee* of a truncated invocation C to be the pair (v, R) where v and R are, respectively, the start node and the region of *entryPred*(*stableAnc*(C)). Note that by definition of *entryPred*, the start node v is an entry node of R, and R is also the region of *stableAnc*(C). Hence the region of C is a subregion of R. A truncated invocation whose chargee is (v, R) is called a *charger* of (v, R).

Lemma 3.14 For each pair (v, R) there is an invocation A with region R such that every charger of (v, R) is a descendant of A.

Proof: The lemma is trivial if (v, R) has no chargers. Otherwise, let C be the earliest charger. Let $A =$ *stableAnc*(C), and let $E =$ *entryPred*(A). Then R is the region of A and E, and v is the start node of E. Moreover, by Lemma 3.12,

$$\mathrm{start}(E) \leq \mathrm{start}(A) \tag{4}$$

Let B be any invocation such that $B > A$. Our goal is to show that no descendant of B is a charger. Suppose for a contradiction that C' is a descendant of B and is a charger of (v, R). It follows that $B =$ *stableAnc*(C') and in particular that B is stable. Let $E' =$ *entryPred*(B), and note that v is the start node of E'. Since B occurs after A, E' occurs no earlier than E. Suppose that E and E' are distinct. In this case E' must occur after A (else *entryPred*(A) would be E'). Moreover, by Lemma 3.11, $\mathrm{start}(E') < \mathrm{start}(E)$. Combining this inequality with (4), we obtain $\mathrm{start}(E') < \mathrm{start}(A)$, contradicting the stability of A. Thus we may assume $E =$ *entryPred*(B).

Case 1: $C = A$. In this case, A is truncated, so $\mathrm{end}(A)$ is infinity. Let A' be the immediate successor of A, and note that since E comes no later than A and B comes after A, $E < A' \leq B$. By Corollary 3.9, A' starts with an entry node of R, contradicting the fact that $E =$ *entryPred*(B).

Case 2: $C \neq A$. By Lemma 3.13, the child of A that is an ancestor of C is stable, contradicting the fact that A is the lowest stable ancestor of C. □

For levels $i \geq j$, let $\beta_{ij} = \alpha_i \alpha_{i-1} \cdots \alpha_{j+1}$. ($\beta_{ii}$ is defined to be 1, and β_{ij} with $i < j$ is defined to be 0.)

Corollary 3.15 For a level-i region R and an entry node v of R, (v, R) has at most β_{ij} level-j chargers.

Proof: By Lemma 3.14, all chargers are descendants of some invocation A with region R. The invocation A results in at most α_i invocations at level $i-1$, each of which results in at most α_{i-1} invocations at level $i-2$, and so on. Thus the number of level-j descendants of A is at most β_{ij}. □

We can use Corollary 3.15 to count the number of level-j truncated invocations.

First, we define r_i to be the maximum size of any level i region. We note that $r_{i-1} \leq g(r_i)$. Also there are $O(n/r_i)$ regions at level i, each consisting of at most r_i nodes and each having $O(f(r_i))$ boundary nodes. This follows from the fact that each level is composed of a subset of the $O(n)$ edges of the original graph. Thus the number of boundary nodes at level i is $O(nf(r_i)/r_i)$. Each is an entry node of at most two regions at that level, so there are $O(nf(r_i)/r_i)$ pairs (v, R) where R is a level-i region and v is an entry node of R. Each pair has at most β_{ij} level-j chargers by Corollary 3.15, for a total of

$$\sum_i O(nf(r_i)\beta_{ij}/r_i) \tag{5}$$

Next we formulate a recurrence for the total number s_j of level-j invocations, including both truncated and non-truncated. Since every level-0 invocation is truncated, $s_0 = \sum_i O(nf(r_i)\beta_{i,0}/r_i)$. Since each nontruncated level-j invocation results in α_j invocations at level $j-1$, the number of such invocations is s_{j-1}/α_j. Hence the total number of level-j invocations is

$$s_j \leq s_{j-1}/\alpha_j + \sum_i O(nf(r_i)\beta_{ij}/r_i) \tag{6}$$

The time required for a single level-j invocation, not including further calls made to Activate or GlobalUpdate, is dominated by the number α_j of iterations of the loop multiplied by the time to obtain the minimum item from the queue and to update its key. Since the queue of a level-j region has $O(r_j/r_{j-1})$ items in its queue, the time for these operations is $O(\log r_j)$. Thus the time for such an invocation is $O(\alpha_j \log r_j)$. It follows that the total time for the algorithm, not including time spent in GlobalUpdate, is

$$\sum_j O(s_j \alpha_j \log r_j)$$

Next we analyze the time spent in GlobalUpdate. Consider a level-0 invocation A_0 of Activate, and let $R(uv)$ be its (atomic) region. It makes at most two calls GlobalUpdate($R(vw), vw, d(v)$), each of which leads to a series of recursive calls to GlobalUpdate, involving a corresponding series of regions $R(vw){=}R_0, R_1, \ldots, R_p$, where R_{i+1} is the parent region of R_i. The call involving region R_i requires $O(\log r_i)$ time (not including recursive calls), so the total time for the series of calls is

$$\sum_{j=1}^{p} O(\log r_j) \tag{7}$$

The key to the analysis is bounding the length of such a series. Lemma 3.16 below shows that all but the last call in the series involves a region R_i of which the node v is a boundary node. Let $level(v) = \max\{i : v \text{ is a boundary node of a level-}i \text{ region}\}$. It follows that the length p of the series is at most $1 + level(v)$.

Suppose the chargee of the level-0 invocation A_0 is (x, R). Let us say A_0 is *type 1* if the level of R is at least $level(v)$, and *type 2* otherwise. First we bound the time spent in type 1 invocations. As in the previous analysis, there are $O(nf(r_j)/r_j)$ pairs (v, R) where R is a level-j region and v is an entry node of R. Each is the chargee of at most $\beta_{j,0}$ level-0 invocations. If such an invocation is type 1 then the time required for the resulting series of calls to GlobalUpdate is $\sum_{i=1}^{j+1} O(\log r_i)$. Thus the total time spent in GlobalUpdates called from type 1 invocations is

$$O(\sum_j (nf(r_j)/r_j)\beta_{j,0} \cdot \sum_{i=1}^{j+1} \log r_i) \tag{8}$$

Now we bound the time for type 2 invocations. Fix an atomic region $R(uv)$ and let $j = level(v)$. We count the type 2 invocations with region $R(uv)$ as follows. By definition of type 2, the chargee of each such invocation is a pair (v, R) where R is an ancestor of $R(uv)$ and v is an entry node of R. If such a region R has level i, it has at most β_{i0} chargees with region $R(uv)$ by Corollary 3.15. Hence the number of type 2 invocations with region $R(uv)$ is at most $\sum_{i<j} f(r_i)\beta_{i0}$. For any j, the number of atomic regions $R(uv)$ with $level(v) = j$ is $O(nf(r_j)/r_j)$ (using the fact that each node has indegree at most 2), and the time required by GlobalUpdate for such a region is $\sum_{i=1}^{j+1} O(\log r_i)$. Hence the total time of the GlobalUpdates caused by type 2 invocations is

$$\sum_{j>1} O(nf(r_j)/r_j) \left(\sum_{i=1}^{j-1} f(r_i)\beta_{i0}\right) \left(\sum_{i=1}^{j+1} \log r_i\right) \tag{9}$$

We now proceed with lemma 3.16.

Lemma 3.16 Suppose that in step A2 a call GlobalUpdate($R(vw), vw, d(v)$) occurs. Let $R(vw){=}R_0, R_1, \ldots, R_p, R_{p+1}$ be the regions to which the resulting recursive invocations of GlobalUpdate are applied. Then v is an entry node of R_p.

Proof: Let A_0 be the invocation of Activate during which the initial call to GlobalUpdate was made. For $i = 0, \ldots, p+1$, let G_i be the invocation of GlobalUpdate with region R_i, and note that G_i calls G_{i+1} in step G3 for $i = 1, \ldots, p$. Since G_{p+1} takes place, it must be that during G_p the condition in step G2 must have been true: the *updateKey* operation in step G1 must have resulted in a reduction in the value of $minKey(Q(R_p))$.

Since R_p is an ancestor of $R_0 = R(vw)$, the edge vw belongs to R_p. To show that v is a boundary node of R_p, we show that the edge uv does not belong to R_p. Suppose for a contradiction that $uv \in R_p$, so $R(uv)$ is a subregion of R_p. Hence the invocation A_0 is a descendant of some invocation A whose region is R_p. If $A = A_0$ then the lemma is trivial. Otherwise, let A' be the child of A, that is an ancestor of A_0. By Lemma 3.6, the start key of A' is the value of $minKey(Q(R_p))$ at the time when A' starts. By Lemma 3.7, every key assigned during A' has value at least the start key of A', contradicting our earlier assertion that step G1 resulted in a reduction in the value of $minKey(Q(R_p))$. □

We now specify an algorithm by setting $\alpha_0 = 1$ and $\alpha_i = 4 \log r_{i+1}/\log r_i$ for $i > 0$. Our algorithm runs in linear time provided that

$$\frac{r_i}{f(r_i)} \geq 8^i f(r_{i-1}) \log r_{i+1} \left(\sum_{j=1}^{i+1} \log r_j\right) \tag{10}$$

We omit the proof of this claim due to lack of space.

Theorem 3.17 For any f such that $f(k) = O(k/2^{\log^\epsilon k})$ where ϵ is a constant, for any class of graphs with an f-separator theorem, there is a shortest-path algorithm that runs in linear time not including the time required for building a recursive division.

Now we address planar graphs specifically. Goodrich [3] has given a linear-time algorithm to find a region decomposition of a planar graph using $\sqrt{n}$-separators. His method can be used to obtain a $(g, \sqrt{n})$-recursive division.in linear time. Alternatively, we can use the fact that a $(g, n^{2/3})$-recursive division is sufficient to achieve the linear-time bound. In the complete version of this paper we will show how to find such a recursive division in linear time.

Corollary 3.18 Single-source shortest paths in planar graphs with nonnegative lengths can be computed in linear time.

Proof sketch: We find a (g, f) recursive division of G where $f(k) = ck/2^{\log^{\epsilon} k}$ and $g(k) = k/2^{c' \log^{\epsilon} k}$. We can show that inequality 10 holds for such a recursive division for appropriate choices of c and c'. □

4 Single-source shortest paths with negative edge-lengths

In this section we consider the shortest-path problem when the edges are allowed to have arbitrary weights. We will use the term *size* (as opposed to length) of a path to refer to the number of edges on the path.

Theorem 4.1 Let G be a n-node planar directed graph such that the sum of the absolute values of the edge-weights is at most D. Given a source node s in G the shortest paths from s to all the other nodes in G can be found in time $O(n^{4/3} \log^{2/3} D)$.

Suppose an r-division has been computed for the input graph. Define the *splitting set* of each region to be the set of boundary nodes of the region. We also use a *separator decomposition* for each region, as explained below. The algorithm consists of three phases.

Phase I: We apply a nested-dissection algorithm [25] to compute, for each region R of the division, two auxiliary graphs H_R and G_R. The nodes of H_R are the splitting nodes, and for every pair of nodes there is an edge whose length is the shortest-path distance in R. The nodes of G_R are the nodes of R, each edge of G_R corresponds to a path in R, and

1. $|E(G_R)| = |E(R)| \log(|E(R)|)$,
2. for any pair of nodes $u, v \in R$ there exists a shortest u-to-v path in G_R that has size $O(\log |V(R)|)$.

This is done using a straightforward divide-and-conquer. The time required is $O(r^{3/2})$ time [4,25,26], but for ease of exposition we describe a slightly simplified version that requires a log factor more time. Given a graph R with splitting set S, we proceed as follows. Suppose R has r nodes. Let X be a size $O(\sqrt{r})$ separator for R that breaks R into up to three pieces R_1, R_2, R_3 such that each piece contains at most a $2/3$ fraction of the nodes of G and a $2/3$ fraction of the nodes of S.[3] Let the splitting set for R_i consist of the nodes of S that lie in R_i, together with the nodes of X. Suppose we have recursively computed H_{R_i} and G_{R_i} for $i = 1, 2, 3$. Then H_R can be computed from the H_{R_i}'s by computing all-pairs shortest paths in the union of these graphs; the time required is $O((\sqrt{r})^3)$. Finally, G_R consists of the union of H_R with the union of the G_{R_i}'s. A simple induction proof shows that this construction has the desired properties.

[3] Such a separator can be obtained by first finding a $V(R)$-balanced separator and then finding an S-balanced separator in the piece that contains more than half the nodes of S.

Phase II: Let H be the union of the H_R's for all the regions R of the division. We use the single-source shortest path algorithm due to Goldberg [14] in H to compute shortest paths from s to all the boundary nodes of the regions.

Phase III: For each region R, we obtain G'_R from H_R by adding a new source node s_R with edges to each of the boundary nodes of R; the lengths of the edges are the distances to the corresponding nodes as computed in Phase II. We then compute shortest paths in G'_R using the Bellman-Ford shortest-path algorithm. Because of property (2) above, we only need $O(\log n)$ iterations of that algorithm (as observed, e.g., by Cohen [5]).

It is easy to verify that the distances computed in Phase III are the correct shortest-path distances.

Now we analyze the algorithm. As remarked earlier, the first phase can be done in $O(r^{3/2})$ time for each region in the division [4,25,26]. Therefore the total time for this phase is $O((n/r)r^{3/2})$. To bound the time required for the single-source shortest-path computation on the auxiliary graph H, we note that H has $O(n)$ edges and $O(n/r^{1/2})$ nodes. Therefore running Goldberg's algorithm on H requires $O(\frac{n^{3/2}}{r^{1/4}} \log D)$ time. The time required to run a Bellman-Ford computation on one region for $O(\log r)$ iterations is $O(r \log r)$ (see for instance [16]). Thus, the time required to run the third phase on all the regions is $O(n \log r)$. Choosing r equal to $n^{2/3} \log^{4/3} D$, we see that the entire algorithm requires $O(n^{4/3} \log^{2/3} D)$ time. We thus get the bounds of Theorem 4.1.

A problem related to maximum flow is that of computing circulations. In this problem we are given a planar directed network with demands (both positive and negative) on the nodes such that the sum of the demands is zero. Our task is to compute a feasible flow assignment to the edges of the network (if one exists) such that the sum of the flow into any node is equal to the demand at that node. Miller and Naor [21] show that a legal-circulation can be computed in a planar graph G by doing a single-source shortest-path computation in the dual graph. The circulation problem is also related to the problem of computing perfect matchings in planar bipartite graph. We can also use our algorithm to compute maximum flow from one or more sources. Using our shortest path algorithm we get the following bounds for these problems:

Corollary 4.2 A legal circulation can be computed in a planar graph in time $O(n^{4/3}\log^{2/3} D)$ time where D is the sum of all the edge-capacities. A perfect matching can be found in $O(n^{4/3}\log^{2/3} n)$ time in an n-node planar bipartite graph.

Given a single source and sink the maximum flow from source to sink can be computed in $O(n^{4/3}\log n\log^{2/3} D)$ time. A minimum cut can also be computed within the same time bounds. If there are multiple sources and sinks that lie on at most k faces then a maximum flow in G can be computed in $O(k^2 n^{4/3}\log n\log^{2/3} D)$ time.

Computing shortest paths in parallel To get an efficient parallel algorithm we follow the same three-phase strategy but instead of using Golberg's algorithm in the second phase we use an algorithm due to Gabow and Tarjan [24]. Given an x-node e-edge graph H with integral edge-lengths that are at most N in magnitude, the algorithm in [24] can compute single-source shortest-paths in H in time $O(\sqrt{x}e\log(xN)(\log 2p)/p)$ using $p \leq e/(\sqrt{x}\log^2 x)$ processors. Using this algorithm in the boundary-path stage and setting r to be $n^{2/3}log^{4/3}n$ and p to be $n^{2/3}/log^{5/3}n$ we can get a parallel algorithm with the following bounds:

Theorem 4.3 Let G be a n-node planar directed graph such that the sum of the absolute values of the edge-lengths is at most D. The single-source shortest-path problem in G from a given source-node s can be solved in time $O(n^{2/3}\log^{7/3} n(\log n + \log D))$ and work $O(n^{4/3}\log n(\log n + \log D))$.

Using this shortest-path algorithm we can also get parallel algorithms for computing a legal-circulation, the maximum st-flow, and a perfect matching in a planar bipartite graph with the bounds shown in the following corollary:

Corollary 4.4 A legal circulation in G can be computed in time $O(n^{2/3}\log^{7/3} n(\log n + \log D))$ and work $O(n^{4/3}(\log n + \log D))$, while a maximum st-flow can be computed in time $O(n^{2/3}\log^{10/3} n(\log n + \log D))$ and work $O(n^{4/3}\log^2 n(\log n + \log D))$.

If G is a planar bipartite graph then we can find a perfect matching in time $O(n^{2/3}\log^{13/3} n)$ with $O(n^{4/3}\log^3 n)$ work.

Dynamic maintenance of shortest paths These partitioning techniques can also be used to derive efficient dynamic algorithms for shortest paths and other related problems. The ideas are similar to a number of other fully dynamic algorithms for graph problems [6, 10,11,20,27].

The basic idea is to divide G into suitable sized pieces and precompute all-boundary pair shortest paths in each piece. These precomputed answers are used to answer any given query quickly. At the same time when edges are added or removed we need only recompute the shortest paths in a few pieces. Our results for the dynamic maintenance of shortest paths are as follows:

Theorem 4.5 Given an n-node planar directed graph G, such that the sum of the absolute-values of the edge-lengths is at most D, there exists a $O(n)$-space fully dynamic data structure to maintain the all-pairs shortest-path information in G. The time per operation is $O(n^{9/7}\log D)$. The time for queries, edge-deletion, and changing lengths is worst-case while the time for adding edges is amortized.

As with the sequential and parallel algorithms our shortest path algorithm can be used to derive dynamic algorithms for maintaining circulations and perfect matchings in planar graphs. The amortized time per operation for maintaining a circulation is $O(n^{9/7}\log D)$, while the amortized time per operation for maintaining a perfect matching is $O(n^{9/7}\log n)$. The details will be given in the full version of the paper.

References

[1] R. Ahuja, K. Mehlhorn, J. Orlin, and R. E. Tarjan, "Faster algorithms for the shortest path problem," *Journal of the Association for Computing Machinery* 37 (1990), 213–223.

[2] N. Alon, P. Seymour, and R. Thomas, "A separator theorem for graphs with an excluded minor and its applications," *Proc. 22nd Annual ACM Symposium on Theory of Computing* (1990), 293–299.

[3] M.J. Atallah, R. Cole, and M.T. Goodrich, "Cascading Divide-and Conquer: A Technique for Designing Parallel Algorithms," *Proc. 28th IEEE Symp. on Foundations of Computer Science* (1987), 151–160.

[4] E. Cohen, "Efficient parallel shortest-paths in digraphs with a separator decomposition," *Proc. 5th Annual Symposium on Parallel Algorithms and Architectures* (1993), 57–67.

[5] E. Cohen, "Parallel algorithms with improved work for shortest-paths from multiple sources," *Proc. 2nd Israel Symposium on Theory of Computing and Systems* (1993), 57–67.

[6] G.N. Frederickson, "Data Structures for On-Line Updating of Minimum Spanning Trees," *Proc. 15th ACM Symp. on Theory of Computing* (1983), 252–257.

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

[8] M. L. Fredman and D. E. Willard, "Trans-dichotomous algorithms for minimum spanning trees and shortest paths," *Proc. 31st Annual IEEE Symposium on Foundations of Computer Science* (1990), 719–725.

[9] M.L. Fredman and R.E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *Journal of the Association for Computing Machinery* 34 (1987), 596–615.

[10] Z. Galil and G. F. Italiano, "Maintaining biconnected components of dynamic planar graphs," *Proc. 18th Int. Colloquium on Automata, Languages, and Programming.* (1991), 339–350.

[11] Z. Galil, G.F. Italiano, and N. Sarnak, "Fully Dynamic Planarity Testing," *Proc. 24th Annual ACM Symposium on Theory of Computing* (1992), 495–506.

[12] H. Gazit and G. L. Miller, "A parallel algorithm for finding a separator in planar graphs," *Proc. 28th Annual IEEE Symposium on Foundations of Computer Science* (1987), 238–248.

[13] J. R. Gilbert, J. P. Hutchinson, and R. E. Tarjan, "A separation theorem for graphs of bounded genus," Journal of Algorithms, 1984.

[14] A.V. Goldberg, "Scaling Algorithms for the Shortest Path Problem," *SIAM Symposium on Discrete Algorithms.* (1993).

[15] M. Goodrich, "Planar Separators and Parallel Polygon Triangulation," *Proc. 24th Annual ACM Symposium on Theory of Computing* (1992), 507–516.

[16] T. H. Cormen, C. E. Leiserson, and R. E. Rivest, *Introduction to Algorithms*, MIT Press, 1990.

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

[18] D.B. Johnson, "Parallel algorithms for minimum cuts and maximum flows in planar networks," *Journal of the Association for Computing Machinery* 34 (1987), 950–967.

[19] D.B. Johnson and S.M. Venkatesan, "Using divide and conquer to find flows in directed planar networks in $O(n^{1.5}\log n)$ time," *Proc. 20th Annual Allerton Conf. on Communication, Control, and Computing* (1982), 898–905.

[20] P. N. Klein and S. Subramanian, "A fully dynamic approximation scheme for all-pairs shortest paths in planar graphs," *Proc. 1993 Workshop on Algorithms and Data Structures* (1993), 442–451.

[21] G. L. Miller and J. Naor, "Flows in planar graphs with multiple sources and sinks," *Proc. 30th IEEE Symposium on Foundations of Computer Science* (1989), 112–117.

[22] G. L. Miller, S. Teng, and S. Vavasis, "A unified geometric approach to graph separators," *Proc. 31st Annual IEEE Symposium on Foundations of Computer Science* (1991), 538–547.

[23] R.J. Lipton and R.E. Tarjan, "A separator theorem for planar graphs," *SIAM Journal of Applied Mathematics* 36 (1979), 177–189.

[24] H. N. Gabow and R. E. Tarjan, "Almost-optimum speed-ups of algorithms for bipartite matching and related problems," *Proc. 20th Annual ACM Symposim on Theory of Computing* (1988), 514–527.

[25] V. Pan and J. H. Reif, "Fast and efficient solution of path algebra problems," *Journal of Computer and System Sciences* 38 (1989), 494–510.

[26] V. Pan and J. H. Reif, "The parallel computation of minimum cost paths in graphs by stream contraction," *Information Processing Letters* 40 (1991), 79–83.

[27] S. Subramanian, "A Fully Dynamic Data Structure For Reachability in Planar Digraphs," *Proc. 1993 European Symposium on Algorithms* (1993).