Faster shortest-path algorithms for planar graphs

Monika R. Henzinger[*]
Department of Computer Science
Cornell University

Philip Klein[†]
Department of Computer Science
Brown University

Satish Rao
NEC Research Institute

Sairam Subramanian[‡]
Bell Northern Research

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.

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. This algorithm can be used to obtain similar bounds for computing a feasible flow in a planar network, for finding a perfect matching in a planar bipartite graph, and for finding a maximum flow in a planar graph when the source and sink are not on the same face. We also give parallel and dynamic versions of these algorithms.*

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 [FrT].[1] For planar graphs, Frederickson [Freb] 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.

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

[*]Research supported by ONR/DARPA grant N00014-92-J-1989 Automatic Generation of Engineering Analysis, and by the NEC Research Institute.

[†]Department of Computer Science, Brown University, Providence, RI 02912-1910. Research partially supported by an NSF PYI award, CCR-9157620, together with PYI matching funds from Xerox Corporation and Honeywell Corporation. Additional support provided by the NEC Research Institute.

[‡]Research partially supported by an NSF PYI award, CCR-9157620, together with PYI matching funds from Xerox Corporation and Honeywell Corporation.

[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 using [AMO, FrW]. 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. In work subsequent to that discussed here, Thorup has developed an algorithm that requires $O(m\log\log m)$ time. [Tho].

known is $O(n^{1/2} m \log L)$ time, due to Goldberg [Gol], which yields $O(n^{3/2} \log L)$ time on sparse (e.g. planar) graphs. For planar graphs, Lipton, Rose, and Tarjan [LRT] showed how to solve the problem in $O(n^{3/2})$ time using planar separators. 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 too large in magnitude.

For planar graphs, shortest-path computation is closely related to network flow. Hassin [Has] 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 [MiN] (see also Johnson and Venkatesan [JoV]) 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.

The key to both our shortest-path algorithms is our use of graph-decompositions based on separators. Lipton and Tarjan showed [LiT] 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 [Freb] 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.[2]

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. We use roughly $\log^* n$ levels of divisions. 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.

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

[2]Goodrich has since given [Goo] a linear-time algorithm to find a separator decomposition in a planar graph. However, we will not need this result.

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. [KPS, PST].

The algorithm for arbitrary lengths first applies the shortest-path algorithm due to Lipton, Rose, and Tarjan [LRT] to each region, obtaining shortest-path distances between each pair of boundary nodes of 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 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 [GaM]. Cohen [Coh] gives a parallel version of the shortest-path algorithm of Lipton, Rose, and Tarjan. To compute shortest paths on the union of complete graphs, instead of using Goldberg's algorithm, which does not directly parallelize, we use 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 [KlS], an approach based in turn on that used in dynamic algorithms for a variety of problems in planar graphs [Frea, GaI, GIS, Sub]. 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 $O(n^{9/7}\log(nL))$.

In Section 2, we introduce some terminology concerning graph divisions. In Section 3, we give our linear-time algorithm for shortest paths with nonnegative edge-lengths. This algorithm assumes the graph is equipped with a recursive division. In Section 4, we describe our simple linear-time algorithm for finding a recursive decomposition.

In Section 5, we give our algorithm for computing shortest paths in the presence of negative lengths, and discuss the application of this algorithm to finding a feasible flow and a maximum flow in a planar network. We also observe that this algorithm can be implemented efficiently in parallel. Finally, we describe the dynamic version of this algorithm in Section 6.

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. For a function f, a class of graphs that is closed under taking subgraphs is said to be *f-separable* if for any $n > 1$, an n-node graph in the class has a separator of size $O(f(n))$. Examples of graph classes that have been shown to be f-separable for sublinear functions f are planar graphs [LiT], bounded-genus graphs [GHT], d-dimensional overlap graphs [MTV], and graphs excluding a fixed graph as a minor [AST].

Frederickson showed [Freb] that separators can be used to find a *division* of a graph, a special kind of partition of the edge-set into two or more 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. Note that each region has strictly fewer edges than the graph itself.

An r-*division* of an n-node graph is a division into $O(n/r)$ regions, each containing at most r nodes including $O(\sqrt{r})$ boundary nodes. Frederickson showed how to find an r-division of an n-node planar graph in $O(n \log n)$ time. In fact, his method applies to any subgraph-closed $\sqrt{n}$-separable class of graphs.

We generalize the notion of an r-division in two ways. we allow the number of boundary nodes per region to be bigger than $O(\sqrt{r})$, and we allow the number of nodes in a region to be bigger than r. Specifically, we define an (r, s)-*division* of an n-node graph to be a division into $O(n/r)$ regions, each containing $r^{O(1)}$ nodes, each having at most s boundary nodes. It is a *strict* (r, s)-division if each region has at most r nodes.

A straightforward adaptation of Frederickson's algorithm [Freb] yields the following lemma.

Lemma 2.1 Suppose $f(n) = o(n)$. For any subgraph-closed f-separable class of graphs, there is a constant c such that every graph in the family has a strict $(r, cf(r))$-division for any r.

Now we describe a recursive construction based on the notion of divisions. It is used in our shortest-path algorithm for nonnegative lengths. 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 a nondecreasing positive integer function f and a positive integer sequence $\bar{r} = (r_0, r_1, r_2, \ldots, r_k)$, an $(\bar{r}, f)$-*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 and $\bar{r}$ is nonempty then in addition the recursive division contains an $(r_k, f(r_k))$-division of G and a $(\bar{r}', f)$-recursive division of each of its regions, where $\bar{r}' = (r_0, r_1, \ldots, r_{k-1})$. The following lemma is immediate from the definition and Frederickson's theorem.

Lemma 2.2 Suppose $f(n) = o(n)$. For some constant c, any graph from a subgraph-closed f-separable class has a $(\bar{r}, cf)$-recursive division for any $\bar{r}$.

For two regions R_1 and R_2 of different divisions of the recursive division, R_1 is an ancestor of R_2 if R_1 contains R_2. The immediate proper ancestor of a region R is called the *parent* of R. *Descendants* and *children* are defined analogously. We also use the term *subregion* to mean descendant. A region with no subregions is called an *atomic region*. In our linear-time algorithm, every atomic region has precisely one edge. 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.

A recursive division can be compactly represented by a rooted tree whose leaves are labeled by distinct edges of G. We call this tree the *recursive-division tree*. The root of the tree represents the region consisting of all of G, the children of the root represent the subregions into which that region is divided, and so on. Each leaf represents a region containing exactly one edge.

In our linear-time algorithm, we assume the input graph is equipped with a recursive-division tree. In Section 4 we describe a linear-time algorithm to find a recursive decomposition. Our decomposition algorithm can be adapted to work for any minor-closed f-separable class of graphs where it takes linear time to find a separator.

3 Shortest paths with nonnegative edge-lengths

3.1 Overview

Like Dijkstra's algorithm, our algorithm maintains a label $d(v)$ for each node v of G giving the length of some path from s to v.

We say an edge vw is *relaxed* if $d(w) \leq d(v) + \text{length}(vw)$. It is well-known that the labels give correct shortest-path distances if the following *shortest-path conditions* are satisfied:

Shortest-path condition 1: $d(s) = 0$,

Shortest-path condition 2: every label $d(v)$ is an upper bound on the distance, and

Shortest-path condition 3: every edge is relaxed,

To *relax* an edge vw is to update $d(w)$ according to the rule

$$d(w) := \min\{d(w), d(v) + \text{length}(vw)\}$$

Dijkstra's algorithm initializes $d(v)$ to infinity for every node, then sets $d(s)$ to 0, and proceeds to relax every edge. As a consequence of the order in which Dijkstra's algorithm relaxes edges, at the end every edge is relaxed. Hence the labels give correct distances.

Our algorithm cannot afford to relax edges in the same order as Dijkstra's algorithm. Our algorithm's order is inferior in that many edges need to be relaxed several times during the course of the algorithm. However, as described below, our algorithm can more cheaply select edges to relax, so the overall running time is lower.

Johnson's implementation of Dijkstra's algorithm maintains a priority queue containing the nodes; the key of a node v is the associated label $d(v)$. Since Dijkstra's algorithm performs $O(n)$ queue operations for a graph with n nodes and $O(n)$ edges, and each queue operation takes $O(\log n)$ time, the total time is $O(n \log n)$.

Frederickson took the first step toward beating this time bound, using the idea of multiple priority queues of different sizes. Frederickson's algorithm decomposes the graph into small regions, then further decomposes those regions into smaller regions. In preprocessing and postprocessing steps, his algorithm does Dijkstra calculations within each region. Because each of these regions is small, the priority queue used in the Dijkstra calculation is small (i.e. contains few elements) and so the queue operations are cheap. The main thrust of his algorithm performs a Dijkstra calculation on the graph consisting of the boundary nodes of the regions. For this calculation, the priority queue is fairly large, but the number of boundary nodes is significantly smaller than n, so the number of queue operations is also much smaller than n. For the main thrust, the algorithm makes use of a data structure developed by Frederickson, called a topology-based heap.

Our algorithm is similar to Frederickson's in that it divides the graph into regions and subregions, and performs relax operations. Like his algorithm, our algorithm runs quickly because most queue operations are performed on smaller queues. Our linear-time algorithm differs in that it makes use of many (roughly $\log^* n$) levels of division: the subregions themselves have subregions, and so on, whereas Frederickson's method uses only three levels. The more fundamental difference, however, is in strategy.

Frederickson's algorithm consists of a series of phases in each of which Dijkstra's algorithm is run to completion on a given graph. In contrast, our algorithm has a limited "attention span." It chooses a region, then performs a number of steps of Dijkstra's algorithm (the number depends on the level of the region), then abandons that region until later. Thus it skips around between the regions.

To provide intuition, we briefly describe a simplified version of our algorithm. The simplified version runs in $O(n \log\log n)$ time. Divide the graph into $O(n/\log^4 n)$ regions of size $O(\log^4 n)$ with boundaries of size $O(\log^2 n)$.[3] We associate a status, *active* or *inactive*, with each edge. Initialize by deactivating all edges and setting all node labels $d(v)$ to infinity. Then set the label of the source to 0, and activate its outgoing edges. Now repeat the following three steps:

Step 1: Select the region containing the lowest-labeled node that has active outgoing edges in the region.

Step 2: Repeat $\log n$ times:

Step 2A: Select the lowest-labeled node v in the current region that has active outgoing edges in the region. Relax and deactivate all its outgoing edges vw in that region. For each of the other endpoints w of these edges, if relaxing the edge vw resulted in decreasing the label of w, then activate the outgoing edges of w.

Note that applying Dijkstra's algorithm to the region would entail repeating Step 2A as many times as there are nodes in the region. Every node would be selected exactly once. We cannot afford that many executions of Step 2A, since a single region is likely to be selected more than once in Step 1. In contrast to Dijkstra's algorithm, when a node is selected in Step 2A, its current label may not be the correct shortest-path distance to that node; its label may later be decreased, and it may be selected again. Since the work done within a region during a single execution of Step 2 is speculative, we don't want to do too much work. On the other hand, we don't want to execute Step 1 too many times. In our analysis of this algorithm, we show how to "charge" an execution of Step 1 to the $\log n$ iterations of Step 2A.

There is an additional detail. It may be that Step 2A cannot be repeated $\log n$ iterations because after fewer than $\log n$ times there are no active outgoing edges left in the region. In this case, we say the execution of Step 2 is *truncated.* Since we cannot charge a truncated execution of Step 2 to $\log n$ iterations of Step 2A, we need another way to bound the number of such executions. One might think that after a region R underwent one such truncated execution, since all its edges were inactive, the same region would never again be selected in Step 1. However, relax-steps on edges in another region R' might decrease the label on a node w on the boundary between R and R', which would result in w's outgoing edges being activated. If w happens to have outgoing edges within R, since these edges become active, R will at some later point be selected once again in Step 1.

This problem points the way to its solution. If R "wakes up" again in this way, we can charge the subsequent truncated execution involving R to the operation of updating the label on the boundary node w. Our analysis makes use of the fact that there are relatively few boundary nodes to bound the truncated executions. Indeed, this is where we use the fact that the regions have small boundaries.

In Subsection 3.2, we give the two procedures that comprise our algorithm, and we prove that when the algorithm terminates, it has calculated correct shortest-path distances. We present the analysis of the algorithms in Subsections 3.3 through 3.8.

The basis for the analysis is a charging scheme described briefly in Subsection 3.3 and detailed in Subsection 3.5. The properties of this scheme are proved in Subsection 3.6. In Subsection 3.4, we analyze the simplified version outlined above, showing that this algorithm runs in $O(n \log\log n)$ time. The analysis contains the rudiments of the ideas used in the multilevel version. Thus intuition gained by studying the simplified verison will help the reader to understand the analysis of the multilevel scheme,

[3]We omit discussion of how to do this within the required time bound. A division with slightly different parameters can be found in linear time; see Section 4.

which is presented in Subsections 3.7 and 3.8. It is the multilevel version that achieves linear running time.

We assume in this section that the graph is equipped with a recursive division. For the simplified version of the algorithm, the recursive division is very simple: the level 2 division consists of a single region comprising the whole graph, the level 1 division consists of regions of size $O(\log^4 n)$ with boundaries of size $O(\log^2 n)$, and, of course, the level 0 division consists of atomic regions. For the multilevel version, we use roughly $\log^* n$ levels of division. The top level division consists of a single region comprising the whole graph, the next level down consists of roughly $n/\log^2 n$ regions of size slightly more than $\log^2 n$, and so on; the precise parameters are specified in Section 4. In Subsection 3.7, we give an inequality involving these parameters, (19), that is sufficient for the running time to be linear.

One of the procedures, PROCESS, refers to parameters α_ℓ, one for each of the levels ℓ of the division except the atomic level. These parameters are set differently for the simplified version and the multilevel version. For the simplified scheme, the middle level, level 1, has the associated parameter $\alpha_1 = \log n$. This value is the number of iterations of the loop in Step 2 in the outline given above. For both schemes, the value 1 is assigned to the parameter associated with the highest level, the level in which there is a single region containing the entire graph. Thus in the simplified scheme, $\alpha_2 = 1$. The settings of the parameters for the multilevel scheme are given in Subsection 3.7.

3.2 The algorithm and its proof of correctness

We assume without loss of generality that the input graph G is directed, that each node has at most two incoming and two outgoing edges, and that there is a finite-length path from s to each node. We assume that the graph is equipped with a recursive division.

For each region R 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.

The algorithm is intended to ensure that for any region R, the minimum key in the queue $Q(R)$ is the minimum distance label $d(v)$ over all edges vw in R that need to be processed. We make this precise in Corollary 3.4. We use the fact that the regions are nested to help us maintain this property. This idea of maintaining priority queues for nested sets is not new, and has been used, e.g. in finding the k^{th} smallest element in a heap [Frec].

We assume the priority queue data structure 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 $minItem(Q)$

We indicate an item is inactive by setting its key to infinity. Items go from inactive to active and back many times during the algorithm. We never delete items from the queue. This convention is conceptually convenient because it avoids the issue of having to decide, for a given newly active item, into which of the many priority queues it should be re-inserted. In an implementation, one might remove inactive items from their queues.

The two procedures are as follows.

Process(R)
Comment: R is a region.

A1 If R contains a single edge uv then

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

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

A4 Else (R is nonatomic)

A5 Repeat $\alpha_{\ell(R)}$ times or until $minKey(Q(R))$ is infinity:

A6 Let $R' := minItem(Q(R))$.

A7 Call Process(R'), and then call $updateKey(Q(R), R', minKey(Q(R')))$.

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 repeatedly call Process(R_G), where R_G is the region consisting of all of G, until the call results in $minKey(Q(R_G))$ being infinity. At this point, the labels $d(v)$ are the shortest distances from s, as we now prove.

Recall that an edge vw is *relaxed* if $d(w) \leq d(v) + \text{length}(vw)$. Our goal is to prove that when the algorithm terminates, the shortest-path conditions hold.

The algorithm immediately assigns $d(s) := 0$ and never changes that label, so shortest-path condition 1 holds. We show that shortest-path condition 2 also holds throughout the algorithm and that shortest-path condition 3 holds if the algorithm terminates. In the next subsection, we bound the time until termination.

Our goal in this subsection is to show the following invariant holds during the algorithm. During the execution of the algorithm, the most recent invocation of Process that is still active is called the *current invocation*, and the region to which that invocation was applied is called the *current region*. If Process(R') was invoked during step A7, the region R' remains the current region until the end of that step The invariant states essentially that for any region R that is not an ancestor of the current region, $minKey(Q(R))$ is the minimum label v of the tail of an edge that may not relaxed. The algorithm terminates when a call Process(R_G) results in $minKey(Q(R_G))$ being infinity. At this point, it follows from the invariant that every edge is relaxed. Hence the labels $d(v)$ give shortest-path distances.

We start by addressing shortest-path condition 2.

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 upper bounds 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 *active* if the key of uv in $Q(R(uv))$ is finite. To prove that every edge is relaxed at termination, we show that (a) if an edge is inactive, then it is relaxed, and (b) at termination all edges are inactive.

Lemma 3.2 If an edge uv is inactive then it is relaxed (except during step A2).

Proof: The lemma holds before the first call to Process since at that point every node but s has label infinity, and every outgoing edge of s is active. The algorithm only deactivates an edge uv, 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 active. $\Box$

Lemma 3.3 The key of an active edge vw is $d(v)$ (except during step A2.

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. $\Box$

Next we show that the queues are "consistent," in a sense to become apparent.

Lemma 3.4 For any region R that is not an ancestor of the current region, the key associated with R in $Q(\text{parent}(R))$ 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 Process, and upon return from Process. When Process(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 Process(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 immediately after the call to Process(R), the calling invocation, which is Process(parent(R)), updates the key of R in $Q(\text{parent}(R))$ to the value $minKey(Q(R))$. $\Box$

Finally, we prove the correctness invariant for the algorithm. As described at the beginning of this subsection, this invariant implies that, when the algorithm terminates, the labels are correct.

Corollary 3.5 For any region R that is not an ancestor of the current region,
$minKey(Q(R)) =$

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

Proof: By induction on the level of R, using Lemma 3.4. $\Box$

3.3 Invocations, and the Charging Invariant

In this subsection, we introduce some terminology used for the analysis of the algorithm, and we state the invariant satisfied by our charging scheme. In Subsection 3.4, we show that, assuming the invariant holds, the simplified algorithm runs in $O(n \log\log n)$ time. In Subsection 3.5, we introduce more terminology, and define our charging scheme. In Subsection 3.7, we formulate a recurrence and obtain an expression for the running time of the many-level scheme. Finally, in Subsection 3.8, we show that the time bound is linear.

The terminology we introduce in this section is used in the analysis of both the simplified and the linear-time algorithms. Consider an invocation A of the procedure Process.[4] The *region* of A is the region R to which Process is applied. The *level* of A is the level of its region R. If invocation B is the result of executing step A7 during invocation A, then B is called the *child* of A and A is the *parent* of B. The children of an invocation A are *ranked* according to when they occur in time. The *descendants* and *ancestors* of an invocation are similarly defined. The *end key* of A, denoted $\mathrm{end}(A)$, is the min key of $Q(R)$ just after the invocation ends.

Recall the informal description of the simplified algorithm in Subsection 3.1. As we explained in that subsection, we ordinarily charge each execution of Step 1 to $\log n$ executions of Step 2A. However, sometimes Step 2A cannot be repeated $\log n$ times because we run out of active edges to select. Because in such a case the loop of Step 2A runs for less time than usual, we say the loop has been truncated. Applying this terminology to the procedure Process, we say an invocation A of Process is *truncated* if $\mathrm{end}(A)$ is infinity, i.e. if at the end of the invocation the queue associated with the region of A has no more active elements. Note that every level-0 invocation is truncated. Furthermore, when a highest-level invocation is truncated, the algorithm terminates.

We define an *entry node* of a region. For each level-0 region $R(uv)$, u is an entry node of the region. For an intermediate-level region R (a level-1 region in the simplified algorithm), each boundary node v of R with an outgoing edge in R is an entry node of R. For the top-level region R_G, s is the only entry node.

In Subsection 3.5, we describe a charging scheme in which each truncated invocation C is charged to a pair (R, v) where R is an ancestor of the region of C and v is an entry node of R. We call C a *charger* of (R, v). Later we show that our charging scheme satisfies the following *charging scheme invariant*:

> For any pair (R, v), there is an invocation B whose region is R such that all chargers of (R, v) are descendents of B.

We can use the invariant to bound the number of chargers of (R, v) at a given level. By the invariant, all such chargers are descendents of B, an invocation with region R. Clearly there is only one charger whose region is R, namely B itself. Let i be the level of R, and let $j > i$. To count the number of level-j chargers of (R, v), note that B has at most α_i children, and each of them has at most α_{i-1} children, and so on, so the number of descendents of B at level j is $\alpha_i \alpha_{i-1} \ldots \alpha_{j+1}$. Thus the number of level-j chargers of (R, v) is at most this number.

Define $\beta_{ij} = \alpha_i \alpha_{i-1} \ldots \alpha_{j+1}$ for $j > i$. β_{ii} is defined to be 1, and for $i < j$, we define β_{ij} to be zero.) We state our bound as a lemma.

Lemma 3.6 For a level-i region R and an entry node v of R, (R, v) has at most β_{ij} level-j chargers.

3.4 Analysis of the simplified algorithm

In this subsection, we show that the charging scheme invariant enables us to show that the simplified algorithm takes $O(n \log\log n)$ time. We start by bounding the number of invocations charging to a pair (R, v), using Lemma 3.6 and the fact that $\alpha_1 = \log n$ and $\alpha_2 = 1$.

- if R has level 0, then (R, v) is charged by at most one level-0 invocation;
- if R has level 1, then the pair is charged by at most one level-1 invocation and at most α_1 level-0 invocations; and

[4]Henceforth, unless otherwise stated, an "invocation" refers to an invocation of Process.

- if R has level 2, then the pair is charged by at most one level-2 invocation, at most $\alpha_2 = 1$ level-1 invocations and at most $\alpha_2\alpha_1 = \log n$ level-0 invocations.

Since there are $O(n)$ such pairs (R, v) where R has level 0, it follows that $O(n)$ truncated level-0 invocations are charged to level-0 regions. There are $O(n/\log^4 n)$ level-1 regions, and each has $O(\log^2 n)$ boundary nodes, so there are $O((n/\log^4 n)(\log^2 n))$ such pairs (R, v) where R has level 1. Each such pair is charged by at most $\log n$ level-0 invocations, for a total of $O(n/\log n)$. Finally, there is only one pair (R, v) where R has level 2 and v is an entry node of R, namely (R_G, s), and it is charged by at most $\log n$ level-0 invocations. Thus there are $O(n)$ truncated level-0 invocations. Similarly, the number of truncated level-1 invocations is $O(O(n/\log^4 n)(\log^2 n))$, which is $O(n/\log^2 n)$. There is at most 1 truncated level-2 invocation.

Let s_j be the total number of of level-j invocations, including both truncated and non-truncated. Since every level-0 invocation is truncated,

$$s_0 = O(n). \tag{2}$$

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-1 invocations is

$$s_1 \leq s_0/\alpha_1 + O(n/\log^2 n) = O(n/\log n), \tag{3}$$

and the total number of level-2 invocations is

$$s_2 \leq s_1/\alpha_2 + 1 = s_1 + 1 = O(n/\log n). \tag{4}$$

Next we analyze the time t_j required for a single level-j invocation, not including further calls made to Process or GlobalUpdate. Obviously, $t_0 = O(1)$. Each level-1 invocation results in $\log n$ operations on a queue of size $\log^4 n$. Each such operation takes $O(\log\log n)$ time, so $t_1 = O(\log n \log\log n)$. Each level-2 invocation results in one operation on a queue of size $O(n/\log^4 n)$, so $t_2 = O(\log n)$. It follows that the total time for the algorithm, not including time spent in GlobalUpdate, is $\sum_j s_j t_j = O(n \log\log n)$.

Next we analyze the time spent in GlobalUpdate. Each initial call to GlobalUpdate is made in a level-0 invocation, which is necessarily a truncated invocation. Consider a level-0 invocation A_0 of Process, 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 $R(vw)=R_0, \ldots, R_p$, for $p \leq 2$ where R_{i+1} is the parent region of R_i. The queue operation done on $Q(R_0)$ takes $O(1)$ time, the queue operation done on $Q(R_1)$ takes $O(\log\log n)$ time, and the queue operation done on $Q(R_2)$ takes $O(\log n)$ time. Thus if the recursion goes no higher than R_1 (i.e. if $p \leq 1$), it takes $O(\log\log n)$ time. Since there are $O(n)$ level-0 invocations, the total time for GlobalUpdate calls with $p \leq 1$ is $O(n \log\log n)$.

To bound the case $p = 2$ we distinguish two cases. Remember that $O(n/\log n)$ level-0 invocations are charged to each level-1 or level-2 region. (These correspond to *type 1* invocations in Section 3.) Their calls to GlobalUpdate require $O((n/\log n)\log n) = O(n)$ time.

The remaining case are the level-0 invocations A_0 that are charged to their own level-0 region $R(uv)$. (These correspond to the *type 2* invocations in Section 3.) Note that there are $\Theta(n)$ of them. We have to show that only $O(n/\log n)$ of them result in a GlobalUpdate call with $p = 2$. To show this bound we associate each level-0 invocation A_0 that is charged to its own level-0 region $R(uv)$ with the node v whose label is being updated. Since the level-0 region $R(uv)$ is charged at most once by a level-0 invocation and v has indegree at most 2, at most two level-0 invocations are associated with a given node v. Note that $p = 2$ implies that v is an entry node for a level-1 region and that there are at most

$O(n/\log^2 n)$ entry nodes for level-1 regions. Thus there are $O(n/\log^2 n)$ level-0 invocations that charge level-0 regions and have $p = 2$. Their total time is $O(n/\log n)$.

This shows that the total time of the algorithm is $O(n \log\log n)$ provided that the charging invariant holds.

3.5 The charging scheme

The *start key* start(A) 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 following lemma is analogous to the fact that in Dijkstra's algorithm the labels are assigned in nondecreasing order.

Lemma 3.7 For any invocation A, and for the children $A_1, \ldots, A_p$ of A,

$$\mathsf{start}(A) \leq \mathsf{start}(A_1) \leq \mathsf{start}(A_2) \leq \cdots \leq \mathsf{start}(A_p) \leq A. \tag{5}$$

Moreover, every key assigned during A is at least $\mathsf{start}(A)$.

Proof: The proof is by induction on the level of A. If A's level is 0 then it has an atomic region $R(uv)$. In this case the start key is the value of $d(u)$ at the beginning of the invocation. The end key is infinity. The only finite key assigned (in GlobalUpdate) is $d(u) + \text{length}(uv)$, which is at least $d(u)$ by nonnegativity of edge-lengths. There are no children.

Suppose A's level is more than 0. In this case it invokes a series $A_1, \ldots, A_p$ of children. Let R be the region of A. For $i = 1, \ldots, p$, let k_i be the value of $minKey(Q(R))$ at the time A_i is invoked, and let k_{p+1} be its value at the end of A_p. Step A6 of the algorithm ensures $k_i = \text{start}(A_i)$ for $i \leq p$. By the inductive hypothesis, every key assigned by A_i is at least start(A_i), so $k_{i+1} \geq k_i$. Putting these inequalities together, we obtain

$$k_1 \leq k_2 \leq \cdots \leq k_{p+1}$$

Note that $k_1 = \text{start}(A)$ and $k_{p+1} = \text{end}(A)$. Thus (5) holds and every key assigned during A is at least start(A). $\Box$

We define the partial order $\leq$ on the set of invocations as follows: $A \leq B$ if A and B have the same region, and A occurs no later than B. We say in this case that A is a predecessor of B. We write $A < B$ if $A \leq B$ and $A \neq B$.

We say A is B's immediate predecessor if $A < B$ and there is no C such that $A < C < B$. Note that because of the definition of the partial order, A need not occur immediately before B; we require only that there be no intervening invocation with the same region.

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. Consider how instability could arise. By Lemma 3.7, computation just within the region of A cannot result in a key within that region being assigned a value less than the start key of A. Thus if A is not stable, computation on some neighboring region must have been the cause. This idea is illustrated by two examples, the case where the invocation's region is the whole graph and the case where it is an atomic region.

If the region R is the top-level region R_G, there are no regions outside R, so such invocations are all stable.

Corollary 3.8 Every top-level invocation is stable.

Proof: Consider two successive such invocations $D < E$. By Lemma 3.7, every key assigned during D is at least start(D), which is the value of $minKey(Q(R_G))$ at the beginning of D. Hence the value of $minKey(Q(R_G))$ at the end of D is at least its value at the beginning. But start(E) is the value of $minKey(Q(R_G))$ at the end of A, so start$(E) \geq$ start(D). □

Next consider a level-0 region $R(vw)$ and two invocations $A < B$ on $R(vw)$. Immediately after A, the edge vw is inactive, i.e. the min key of $Q(R(vw))$ is infinity. The edge can only become active again if the label $d(v)$ is decreased. When B occurs, the edge is active, so it follows that start$(B) <$ start(A), and that A is not stable. Hence if a level-0 invocation of $R(vw)$ is stable, then no further level-0 invocations of $R(vw)$ occur, i.e. $R(uv)$ is "finished."

For an invocation A, define stableAnc(A) to be the lowest stable ancestor of A. By Corollary 3.8, A has at least one stable ancestor, so this is well-defined.

As mentioned above, between two successive invocations A and B with the same region R, a computation on a region R' outside R might cause a key in R to decrease. Since computations on R' only affect nodes of R', it must be that some node in both R' and R (an entry node of R) had its label decreased, and the new label becomes the minimum key in $Q(R)$. In this case the invocation B starts with this entry node. Thus intuitively a series of invocations $E, A_1, A_2, \ldots, A_k$ with the same region R during which no A_i starts with an entry node represents a period during which the computation on R is not affected by computation on regions outside R, so $minKey(Q(R))$ does not decrease during this period.

To characterize such a series of invocations, we define the *entry-predecessor* of an invocation A to be the latest invocation $E \leq A$ such that E starts with an entry node. Note that the series $E, A_1, A_2, \ldots, A_k$ is characterized by the fact that all these invocations have the same entry-predecessor, namely E. We denote the entry-predecessor of A by entryPred(A)

Here is our charging scheme. We charge a truncated invocation C to the pair (R, v) where R and v are, respectively, the region and the start node of entryPred(stableAnc(C)). To spell it out more explicitly, let A be the lowest stable ancestor *stableAnc*(C), and let R be the region of A. Thus the region of C is a subregion of R. Also, let E be the latest predecessor of A that starts with an entry node, i.e. $E =$ entryPred(A), and let v be its start node. By definition of entryPred, the start node v is an entry node of the region of E, which is R because $E \leq A$. Thus the *chargee* (R, v) of C is a region R that is an ancestor of the region of C, together with an entry node of R. We say C is a *charger* of (R, v).

We prove in Lemma 3.14 that for any invocations $A < B$ with entryPred$(A) =$ entryPred(B), if B is stable then the children of A are stable. For intuition on how this lemma helps us prove the charging scheme invariant, suppose that some descendent B' of B were charged to (R, v) where R is the region of B. We infer that B is the lowest stable ancestor of B' and therefore in particular that B is stable. Hence by Lemma 3.14 the children of A are stable, and hence A is not the lowest stable ancestor of any of its proper descendents. This shows that no such proper descendents of A are charged to (R, v).

3.6 Proving the charging scheme invariant

Lemma 3.9 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 Process 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 an boundary node of R. Hence in either case v is an entry node of R. $\Box$

We immediately obtain two corollaries.

Corollary 3.10 Let A and B be invocations with region R such that A is B's immediate predecessor. If $\mathsf{end}(A) > \mathsf{start}(B)$ then B starts with a entry node of R.

Corollary 3.11 For each region R, the first invocation with region R starts with an entry node of R.

Lemma 3.12 If $A < B$ are two invocations with the same start node then $\mathsf{start}(A) > \mathsf{start}(B)$.

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 $\mathrm{start}(A) > \mathrm{start}(B)$. $\Box$

Lemma 3.13 For every invocation A,

$$\mathsf{start}(\mathsf{entryPred}(A)) \leq \mathsf{start}(A).$$

Proof: Let $A_0 = \mathrm{entryPred}(A)$, 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 3.10, $\mathrm{start}(A_0) \leq \mathrm{start}(A_1) \leq \cdots \leq \mathrm{start}(A)$. $\Box$

Lemma 3.14 Suppose $A < B$ are two invocations such that $\mathsf{entryPred}(A) = \mathsf{entryPred}(B)$. 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 $\mathrm{entryPred}(A) = \mathrm{entryPred}(B)$, the start nodes of $C_1, \ldots, C_p$ are not entry nodes of R. Hence it follows by Corollary 3.10 that $\mathrm{end}(C_{i-1}) \leq \mathrm{start}(C_i)$ for $i = 1, \ldots, p$. Moreover, it follows from Lemma 3.7 that $\mathrm{start}(C_i) \leq \mathrm{end}(C_i)$ for $i = 1, \ldots, p$. We infer that

$$\mathrm{end}(A) \leq \mathrm{start}(C_i) \tag{6}$$

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

$$\mathrm{start}(A') \leq \mathrm{start}(C') \tag{7}$$

Let C be the parent invocation of C' (so the region of C' is R). If $C = A$, then (7) follows from Lemma 3.7.

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

Case 1: $C = C_i$ for some $i \geq 1$. In this case, $\mathrm{end}(A) \leq \mathrm{start}(C)$ by (6).

Case 2: $C > B$. In this case, $\mathrm{end}(A) \leq \mathrm{start}(B)$ follows by (6), and $\mathrm{start}(B) \leq \mathrm{start}(C)$ follows by the stability of B, so we obtain $\mathrm{end}(A) \leq \mathrm{start}(C)$. $\Box$

Finally, we prove the charging scheme invariant.

Lemma 3.15 For each pair (R, v) there is an invocation A with region R such that every charger of (R, v) is a descendant of A.

Proof: The lemma is trivial if (R, v) has no chargers. Otherwise, let C be the earliest charger. Let $A = \text{stableAnc}(C)$, and let $E = \text{entryPred}(A)$. Then R is the region of A and E, and v is the start node of E. Moreover, by Lemma 3.13,

$$\text{start}(E) \leq \text{start}(A) \tag{8}$$

Let B be any invocation such that $B > A$. Our goal is to show that no descendant of B is a charger of (R, v). Suppose for a contradiction that C' is a descendant of B and is a charger of (R, v). It follows that $B = \text{stableAnc}(C')$ and in particular that B is stable. Let $E' = \text{entryPred}(B)$, 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 $\text{entryPred}(A)$ would be E'). Moreover, by Lemma 3.12, $\text{start}(E') < \text{start}(E)$. Combining this inequality with (8), we obtain $\text{start}(E') < \text{start}(A)$, contradicting the stability of A. Thus we may assume $E = \text{entryPred}(B)$.
Case 1: $C = A$. In this case, A is truncated, so $\text{end}(A)$ 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, $E \leq A' < B$. Since the start key of A' must be finite, it follows from Corollary 3.10 that A' starts with an entry node of R, contradicting the fact that $E = \text{entryPred}(B)$.
Case 2: $C \neq A$. By Lemma 3.14, the child of A that is an ancestor of C is stable, contradicting the fact that A is the lowest stable ancestor of C. $\Box$

3.7 An expression for the running time of the algorithm

In this subsection, we obtain an expression for the running time of the algorithm in terms of the characteristics of the recursive decomposition and the parameters α_i. In the next section, we state what our algorithm requires of the recursive decomposition. We show that if this requirement is met, there is a way to set the parameters to achieve linear time.

Our estimate of the running time consists of several parts. We formulate a recurrence relation for the number s_j of level-j invocations of Process, and then give a formula in terms of the s_j's for the running time not including the time for GlobalUpdate. We then consider two kinds of calls to GlobalUpdate, and separately bound the total time for calls of each kind.

Let the recursive decomposition be a $(\bar{r}, f)$-recursive decomposition, and write $\bar{r} = (1, r_1, r_2, \ldots, r_k)$. For notational convenience, let r_0 denote 2.

There are $O(n/r_i)$ level-i regions, each having $O(f(r_i))$ boundary nodes. Thus the total 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 (R, v) 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 3.6. Summing over all levels i, we obtain the following lemma.

Lemma 3.16 The number of level-j truncated invocations is

$$\sum_i O(nf(r_i)\beta_{ij}/r_i) \tag{9}$$

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, we get $s_0 = \sum_i O(nf(r_i)\beta_{i,0}/r_i)$ directly from Lemma 3.16. 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 satisfies the following recurrence.

$$s_0 = \sum_i O(nf(r_i)\beta_{i,0}/r_i) \tag{10}$$

$$s_j \leq s_{j-1}/\alpha_j + \sum_i O(nf(r_i)\beta_{ij}/r_i) \tag{11}$$

The time required for a single level-j invocation ($j > 0$), not including further calls made to Process 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 contains $O(r_j^{O(1)})$ items, the time for these operations is $O(\log r_j)$. Thus the time for such an invocation is $O(\alpha_j \log r_j)$. The time for a single level-0 invocation is $O(1)$, which is $O(\log r_0)$ since we previously set $r_0 = 2$. For notational convenience, set $\alpha_0 = 1$. Thus we obtain the following lemma.

Lemma 3.17 The total time for the algorithm, not including time spent in GlobalUpdate, is

$$O\left(\sum_{j\geq 0} s_j \alpha_j \log r_j\right)$$

Next we analyze the time spent in GlobalUpdate. Consider a level-0 invocation A_0 of Process, 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

$$\sum_{i=1}^{p} O(\log r_i) \tag{12}$$

The key to the analysis is the following lemma bounding number of terms in this series. We postpone its proof until the end of the section.

Lemma 3.18 Suppose that in step A2 a call GlobalUpdate$(R(vw), vw, d(v))$ occurs. Let $R(vw)=R_0, R_1, \ldots, R_{p-1}, R_p$ be the regions to which the resulting recursive invocations of GlobalUpdate are applied, and suppose $p \geq 1$. Then v is an entry node of R_{p-1}.

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 $1 + \textit{level}(v)$. We obtain the following corollary.

Corollary 3.19 Suppose that in step A2 a call GlobalUpdate$(R(vw), vw, d(v))$ occurs. The total time for the resulting series of calls to GlobalUpdate is $O(\sum_{i=1}^{1+level(v)} \log r_j)$

Suppose the chargee of the level-0 invocation A_0 is (R, x). 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, for each level j there are $O(nf(r_j)/r_j)$ pairs (R, x) where R is a level-j region and x 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 by Corollary 3.19 the time required for the resulting series of calls to GlobalUpdate is $\sum_{i=1}^{j+1} O(\log r_i)$. We thus obtain the following lemma.

Lemma 3.20 The total time spent in GlobalUpdates called from type 1 invocations is

$$O\left(\sum_j (nf(r_j)/r_j)\beta_{j,0} \cdot \sum_{i=1}^{j+1} \log r_i\right) \tag{13}$$

Now we bound the time for type 2 invocations. We do this by bounding the following quantity.

$$\sum_j \sum_{v:level(v)=j} \sum_{\text{edges } uv} \text{GlobalUpdate time for type 2 invocations with region } R(uv) \tag{14}$$

Fix an edge uv, and let $j = level(v)$. An invocation with region $R(uv)$ is charged to a pair (R, x) where R is an ancestor of $R(uv)$ and x is a boundary node of R. Let $R_0, R_1, R_2, \ldots$ be the ancestors of $R(uv)$, where R_ℓ is the ancestor at level ℓ. If an invocation with region $R(uv)$ has type 2 then it is charged to a pair (R, x) where $level(R) < j$. Thus GlobalUpdate time for type 2 invocations with region $R(uv)$ is

$$\sum_{\ell<j} \sum_{\text{boundary nodes } x \text{ of } R_\ell} \text{GlobalUpdate time for chargers of } (R_\ell, x) \text{ with region } R(uv) \tag{15}$$

By Corollary 3.6, the number of level-0 chargers of (R_ℓ, x) is at most $\beta_{\ell 0}$. By Corollary 3.19, the Global Update time for a single invocation with region $R(uv)$ is $O(\sum_{i=1}^{j+1} \log r_i)$. Hence the GlobalUpdate time for chargers of (R_ℓ, x) with region $R(uv)$ is $\beta_{\ell 0} \cdot O(\sum_{i=1}^{j+1} \log r_i)$. We substitute into 15 and use the fact that R_ℓ has at most $f(r_\ell)$ boundary nodes, obtaining

$$\sum_{\ell<j} f(r_\ell)\beta_{\ell 0} \cdot O\left(\sum_{i=1}^{j+1} \log r_i\right) \tag{16}$$

for the GlobalUpdate time for type 2 invocations with region $R(uv)$. We substitute into 14 and use the fact that each node v has indegree at most two, obtaining

$$\sum_j \sum_{v:level(v)=j} 2\sum_{\ell<j} f(r_\ell)\beta_{\ell 0} \cdot O\left(\sum_{i=1}^{j+1} \log r_i\right) \tag{17}$$

as a bound on 14.

Recall that $level(v) = j$ means that v is a boundary node of a level-j region. The number of such regions is $O(n/r_j)$, and each has $f(r_j)$ boundary nodes, so we can rewrite 17 as

$$\sum_j O(n/r_j) \cdot f(r_j) \cdot 2\sum_{\ell<j} f(r_\ell)\beta_{\ell 0} \cdot O\left(\sum_{i=1}^{j+1} \log r_i\right)$$

We thus obtain the following lemma.

Lemma 3.21 The total time spent in GlobalUpdates called from type 2 invocations is

$$\sum_{j>1} O(nf(r_j)/r_j)\left(\sum_{\ell=1}^{j-1} f(r_\ell)\beta_{\ell 0}\right)\left(\sum_{i=1}^{j+1} \log r_i\right) \tag{18}$$

We now proceed with the proof of Lemma 3.18.

Proof: Let A_0 be the invocation of Process during which the initial call to GlobalUpdate was made, and let $R(uv)$ be the (atomic) region of A_0. 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 $p = 0$, a contradiction. Otherwise, let A' be the child of A that is an ancestor of A_0. 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 $minKey(Q(R_p))$. $\Box$

3.8 The linear time bound

In this subsection, we specify a condition on the quality of the recursive decomposition, and we assign values to the parameters α_i appearing in the procedure Process. Based on these, we show a linear bound for the running time of the algorithm.

Recall from the previous subsection that r_i denotes the maximum number of edges in a level-i region, and each such region has $O(f(r_i))$ boundary nodes. We can show that our algorithm runs in linear time if our recursive decomposition satisfies the following condition.

$$\frac{r_i}{f(r_i)} \geq 8^i f(r_{i-1}) \log r_{i+1} \left(\sum_{j=1}^{i+1} \log r_j \right) \tag{19}$$

for all r_i's exceeding a constant.

Let us assume that the condition holds. We set $\alpha_i = 4 \log r_{i+1} / \log r_i$. Recall that β_{ij} is defined to be $\alpha_i \alpha_{i-1} \ldots \alpha_{j+1}$. We obtain $\beta_{i,j} = 4^{i-j} \frac{\log r_{i+1}}{\log r_{j+1}}$. Using this setting, we state the following inequalities for later use. These inequalities are derived in the Appendix. They hold for all r_i's exeeding a constant.

$$\beta_{i,0} \frac{f(r_i)}{r_i} \leq \frac{1}{2^i} \tag{20}$$

$$\alpha_i \log r_{i+1} \frac{f(r_i)}{r_i} \leq \frac{4}{8^i} \tag{21}$$

$$\beta_{i,0} \frac{f(r_i)}{r_i} \left(\sum_{j=1}^{i+1} \log r_j \right) \leq 1/2^i \tag{22}$$

$$f(r_i) \beta_{i,0} \geq 2 f(r_{i-1}) \beta_{i-1,0} \tag{23}$$

$$\beta_{i-1,0} \frac{f(r_i)}{r_i} f(r_{i-1}) \left(\sum_{j=1}^{i+1} \log r_j \right) \leq 1/2^i \tag{24}$$

The total time consumed by the calls to Process is $O(\sum_{j \geq 0} s_j \alpha_j \log r_j)$. We will show that the jth term in this series is bounded by $cn/2^j$ for some constant c. This leads to an $O(n)$ bound on the series.

Lemma 3.22 $s_j\alpha_j \log r_j \le cn/2^j$ for some constant c.

Proof: The proof is by induction on j. Recall from 10 that $s_0 = \sum_i c_1\beta_{i,0} nf(r_i)/r_i$ for some constant c_1. By inequality 20, the ith term in this series is at most $c_1 n/2^i$. Hence $s_0 \le 2c_1 n$. We previously set $\alpha_0 = 1$ and $\log r_0 = 1$. Hence $s_0\alpha_0 \log r_0 \le 2c_1 n$. The induction basis therefore holds if we choose $c \ge 2c_1$.

Assume by the inductive hypothesis that $s_i\alpha_i \log r_i$ is bounded by $cn/2^i$. Our goal is to bound $s_{i+1}\alpha_{i+1}\log r_{i+1}$ by $cn/2^{i+1}$. We consider inequality 11, which is restated below (with a change of index names and the introduction of c_2 in place of a big-O notation).

$$s_{i+1} \le s_i/\alpha_{i+1} + \sum_j \frac{c_2 nf(r_j)\beta_{j,i+1}}{r_j}$$

Since $s_i\alpha_i \log r_i$ is bounded by $cn/2^i$, and $\alpha_i = 4\frac{\log r_{i+1}}{\log r_i}$ we can rewrite the inequality above as follows.

$$s_{i+1} \le \frac{cn}{4(2^i\alpha_{i+1}\log r_{i+1})} + \sum_j \frac{c_2 nf(r_j)\beta_{j,i+1}}{r_j}$$

By substituting $\beta_{j,i+1} = 4^{j-i-1}\log r_{j+1}/\log r_{i+1}$ and using inequality 21 we can simplify the inequality further as follows.

$$\begin{aligned} s_{i+1} &\le \frac{cn}{4(2^i\alpha_{i+1}\log r_{i+1})} + \sum_{j\ge i+1} \frac{4c_2 n}{8^{i+1}(2^{j-i-1}\alpha_j \log r_{i+1})} \\ &\le \frac{cn}{4(2^i\alpha_{i+1}\log r_{i+1})} + \frac{8c_2 n}{(8^{i+1}\alpha_{i+1}\log r_{i+1})} \\ &\le \frac{cn}{2^{i+1}\alpha_{i+1}\log r_{i+1}} \end{aligned}$$

The second inequality above follows from the fact that each α_j in the summation is at least as large as α_{i+1} and the fact that 2 is an upper bound on the geometric series defined by $\sum_{i\ge 0} 1/2^i$. The final inequality follows for $c > 3c_2$. We have shown $s_{i+1}\alpha_{i+1}\log r_{i+1} \le cn/2^{i+1}$, completing the induction step. $\Box$

The above lemma implies that the total Process time is linear.

We proceed by verifying that the total time for GlobalUpdates is linear. First we consider the type 1 GlobalUpdate time which is bounded by the expression in (13), restated below.

$$O\left(\sum_i (nf(r_i)/r_i)\beta_{i,0} \cdot \sum_{j=1}^{i+1} \log r_j\right)$$

By inequality 22, each term of the summation above is bounded by $cn/2^i$ for some c . Thus the total type 1 GlobalUpdate time is $O(n)$.

To finish, we bound the total type 2 GlobalUpdate time. Recall that this time is bounded by equation 18 which is restated below.

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

Using equation 23, we can simplify the summation to

$$\sum_{j>1} O(nf(r_j)/r_j) f(r_{j-1})\beta_{j-1,0} \left(\sum_{i=1}^{j+1} \log r_i \right).$$

(We note that the constants in the big O notation in the two summations above are different.) We can bound this summation by $O(n)$ by using inequality 24 to bound the ith term by $cn/2^i$. Thus the total type 2 GlobalUpdate time is linear.

This completes the analysis of the algorithm for computing shortest-path with nonnegative edge-lengths.

In the following theorem, we show that for a class of graphs with small enough separators, a recursive division exists that is sufficiently good for the shortest-path algorithm to run in linear time. The proof is given in the Appendix.

Theorem 3.23 For any f such that $f(k) = O(k/2^{\log^\epsilon k})$ where ϵ is a constant, for any subgraph-closed f-separable 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.

The above analysis does not include the time to find the recursive decomposition. For a minor-closed $O(\sqrt{n})$-separable graph class where finding a separator of size $O(\sqrt{n})$ takes linear time, we give a linear-time algorithm to construct a sufficiently good recursive division. The algorithm is given in Section 4. The class of planar graphs is such a class, so we obtain the following corollary.

Corollary 3.24 Single-source shortest paths in planar graphs with nonnegative lengths can be computed in linear time.

4 Finding a recursive division in linear time

Recall that a recursive division of a graph is a division of the graph into regions, a division of each of those regions into subregions, and so on. In this section, we give a linear-time algorithm to find an $((r_0, r_1, r_2, \ldots), f)$-recursive decomposition where $f(x)$ is a function to be defined. (Roughly speaking, $f(x) = O(x^{3/4}\log^2 x)$.) Recall that this means that the level-j decomposition consists of $O(n/r_j)$ regions, each containing $O(r_j^{O(1)})$ edges, and each having $O(f(r_j))$ boundary nodes. The number of levels for our division is roughly $\log^* n$. The values r_j are determined in Lemma 4.3, where we show that this recursive division is good enough for inequality 19 to be satisfied, and therefore good enough for the shortest-path algorithm to run in linear time.[5] The algorithm works for any minor-closed $\sqrt{n}$-separable family of graphs. In fact, the algorithm can be modified to work for any minor-closed $n^{1-\epsilon}$-separable family of graphs, where ϵ is a positive constant.

The recursive decomposition algorithm is based on an idea of Frederickson [Freb]. In his Lemma 4, Frederickson shows how to find an r-division of an n-node planar graph in $O(n \log r + (n/\sqrt{r}) \log n)$ time. The approach is as follows: decompose the graph into connected subgraphs all of roughly the same size. Contract each of these subgraphs into a single node. Apply the basic algorithm for finding an r-division to the contracted graph. Undo the contractions, obtaining a division of the original graph. Finally, apply the basic algorithm to each of these regions.

In our recursive-division algorithm we use the same basic strategy, but we apply it recursively and with different parameters. We call the contraction operation $\textsc{Contract}(G, z)$. This operation uses a

[5]The top-level division consists of a single region, and the next division consists of regions of size roughly polynomial in $\log\log n$.

procedure FINDCLUSTERS from [Frea] to determine a decomposition of the graph G into at most n/z connected subgraphs each containing at most $3z$ nodes, and then contracts each subgraph to a single node. The time required by CONTRACT on an n-node graph is $O(n)$. For a node v in the contracted graph, let EXPAND(v) denote the set of nodes contracted to form v.

The basic procedure for finding a division, which we call DIVIDE, comes from [Freb]. Given an n-node graph G and a node-subset S, DIVIDE(G, S, r) divides up the graph into regions each having at most r nodes. Call a node v a boundary node if (1) it lies in more than one region, or (2) it belongs to S. The output division has the property that each region has at most $c_1\sqrt{r}$ boundary nodes, where c_1 is a constant. Furthermore, the number of regions is at most $c_2(|S|/\sqrt{r} + n/r)$, where c_2 is a constant. The time required by DIVIDE on an n-node graph is $O(n \log n)$.

Our algorithm has two phases. In the first phase, the algorithm forms a series of contracted versions of the input graph G. Each graph in the series is obtained from the previous graph by a call to CONTRACT.

Let n be the number of nodes in G.
Let $G_0 := G$.
Let $z_0 := 2$.
Let $i := 0$.
While number of nodes in G_i exceeds $n/\log n$, do
    Let $G_{i+1} :=$ CONTRACT(G_i, z_i)
    Let $z_{i+1} := 7^{z_i^{1/5}}$
    Let $i := i + 1$.
Let $I := i - 1$.

In the second phase of the algorithm, the graphs in the series are considered in reverse order and a division is obtained for each of them. The division for G_i is obtained from that for G_{i+1} by (1) further subdividing the regions of the division of G_{i+1}, and (2) obtaining regions of G_i by replacing each node v of G_{i+1} with the nodes of G_i contracted to form v. During this process, the recursive-division tree is constructed, except for the leaves.

Create a vertex v_G to be the root of the recursive-division tree.
Let D_{I+1} be the trivial division of G_{I+1} consisting of a single region.
For $i := I$ down to 0 do
  For each region R of D_{i+1}
    Let S_R be the nodes in region R that also appear in other regions of D_{i+1}.
    Let $D_R :=$ DIVIDE(R, S_R, z_i).
    For each region R' of D_R,
      Expand R' into a region R'' of G_i by replacing each $v \in R'$ with EXPAND(v).
      Create a child $v_{R''}$ of v_R in the recursive-division tree.
  Let D_i be the decomposition of G_i consisting of the regions R'' found above.

In the third phase, the leaves are added to the recursive-division tree.

For each region R of D_0,
  For each edge e of R, create a child v_e of v_R.

For $i = 0, \ldots, I+1$, let n_i denote the number of nodes in G_i, and let k_i denote the number of regions in the division D_i of G_i. The analysis of the above algorithm consists of four parts. First, we analyze the division D_i to show that each of its regions has at most $3z_i^2$ nodes and $O(z_i^{1.5})$ boundary nodes. Second, we show that the number k_i of regions is $O(n_i/z_i^2)$. Third, we show that the recursive-division algorithm takes linear time. Finally, we show that the recursive division obtained is good enough for use in the linear-time shortest-path algorithm.

We start by bounding the number of nodes and number of boundary nodes per region. For notational convenience, let $z_{I+1} = \sqrt{n_{I+1}}$. Then the single region of the division D_{I+1} of G_{I+1} has z_{I+1}^2 nodes. Consider iteration $i \le I$ in the second phase. By the correctness of DIVIDE, the decomposition D_R of a region of D_{i+1} consists of regions R' of size at most z_i. By the correctness of CONTRACT, each node of G_{i+1} expands to at most $3z_i$ nodes of G_i. Hence each region R'' obtained from R' by expanding has size at most $3z_i^2$. Similarly, each region R' has at most $c_1\sqrt{z_i}$ boundary nodes by the correctness of DIVIDE, so the corresponding region R'' has at most $3c_1 z_i^{1.5}$ boundary nodes.

Next we bound the number of regions in each division.

Lemma 4.1 The number k_i of regions in the division D_i is $O(n_i/z_i^2)$.

Proof: We show by reverse induction on i that $k_i \le c_3 n_i/z_i^2$ for all $i \ge i_0$, where i_0 and c_3 are constants to be determined. For the basis, we have $k_{I+1} = 1$.

Consider iteration $i \le I$ in the second phase, and suppose $i \ge i_0$. The regions of D_i are obtained by subdividing the k_{i+1} regions comprising the division of G_{i+1}. By the inductive hypothesis, we have

$$k_{i+1} \le c_3 n_i/z_i^2 \tag{25}$$

Each region R in the division of G_{i+1} has $|S_R| \le 3c_1 z_{i+1}^{1.5}$ boundary nodes. Let n_R be the number of nodes in R. Summing over all regions R in D_{i+1}, we obtain

$$\begin{aligned} \sum_R n_R &= \sum_R (\text{number of non-boundary nodes} + \text{number of boundary nodes}) \\ &\le n_{i+1} + \sum_R 3c_1 z_{i+1}^{1.5} \\ &\le n_{i+1} + 3c_1 k_{i+1} z_{i+1}^{1.5} \end{aligned} \tag{26}$$

For each region R, by correctness of DIVIDE, the number of subregions into which R is divided is at most $c_2(|S_R|/\sqrt{z_i} + n_R/z_i)$, which in turn is at most $c_2(3c_1 z_{i+1}^{1.5}/\sqrt{z_i} + n_R/z_i)$. Summing over all such regions R and using (26) and (25), we infer that the total number of subregions is at most

$$\begin{aligned} \sum_R c_2(3c_1 z_{i+1}^{1.5}/\sqrt{z_i} + n_R/z_i) &= c_1c_2k_{i+1}z_{i+1}^{1.5}/\sqrt{z_i} + c_2\sum_R n_R/z_i \\ &\le 3c_1c_2k_{i+1}z_{i+1}^{1.5}/\sqrt{z_i} + c_2(n_{i+1} + 3c_1k_{i+1}z_{i+1}^{1.5})/z_i \\ &\le 3c_1c_2\left(\frac{c_3 n_{i+1}}{z_{i+1}^2} z_{i+1}^{1.5}\right)/\sqrt{z_i} + c_2n_{i+1}/z_i + 3c_1c_2\left(\frac{c_3n_{i+1}}{z_{i+1}^2}\right) z_{i+1}^{1.5}/z_i \\ &\le 3c_1c_2c_3n_{i+1}/\sqrt{z_iz_{i+1}} + c_2n_{i+1}/z_i + 3c_1c_2c_3n_{i+1}/z_i\sqrt{z_{i+1}} \\ &\le 3c_1c_2c_3n_i/z_i^{1.5}\sqrt{z_{i+1}} + c_2n_i/z_i^2 + 3c_1c_2c_3n_i/z_i^2\sqrt{z_{i+1}} \end{aligned} \tag{27}$$

where in the last line we use the fact that $n_{i+1} \le n_i/z_i$. We have obtained an upper bound on the total number of subregions into which the regions of D_{i+1} are divided. Each subregion becomes a region of D_i. Thus we have in fact bounded k_i, the number of regions of D_i. To complete the induction step, we show that each of the three terms in (27) is bounded by $c_3n_i/3z_i^2$.

The second term, $c_2 n_i/z_i^2$, is bounded by $c_3 n_i/3z_i^2$ if we choose $c_3 \geq 3c_2$. The third term is smaller than the first term. As for the first term, recall that $z_{i+1} = 7^{z_i^{1/5}}$. For sufficiently large choice of i_0, we can ensure that $i \geq i_0$ implies $\sqrt{z_{i+1}} \geq 9c_1c_2\sqrt{z_i}$. Thus the first term is bounded by $c_3 n_i/3z_i^2$.

We conclude that $k_i \leq c_3 n_i/z_i^2$, completing the induction step. We have shown this inequality holds for all $i \geq i_0$. As for $i < i_0$, clearly $k_i \leq (z_i^2)n_i/z_i^2 \leq (z_{i_0}^2)n_i/z_i^2$. Thus by choosing c_3 to exceed the constant $z_{i_0}^2$, we obtain $k_i \leq c_3 n_i/z_i^2$ for every i. □

Note for the proof of the next lemma that by combining Lemma 4.1 with inequality (26), we infer that the sum $\sum_R n_R$ of sizes of all regions in D_{i+1} is $n_{i+1} + O(n_{i+1}/\sqrt{z_{i+1}})$, which is $O(n_i)$.

Next we analyze the running time of the recursive-division algorithm.

Lemma 4.2 The algorithm requires $O(n)$ time, where n is the number of nodes in the input graph.

Proof: The time required to form the graphs $G_1, G_2, \ldots, G_{I+1}$ is $O(\sum_i n/z_i)$, which is $O(n)$. For $i \leq I$, the time to apply Divide to a region R of G_{i+1} with n_R nodes is $O(n_R \log n_R)$. Each such region has $O(z_{i+1}^2)$ nodes, so the time is $O(n_R \log z_{i+1})$. Summed over all regions R, we get $\sum_R O(n_R \log z_{i+1}) = O(n_{i+1} \log z_{i+1})$. The time to obtain the induced division of G_i is $O(n_i)$. Thus the time to obtain divisions of all the G_i's is $\sum_i O(n_{i+1} \log z_{i+1})$. Since $n_{i+1} \leq n/z_i$ and $\log z_{i+1} = O(z_i^{1/5})$, the sum is $O(n)$. □

Finally we show that the recursive division is good enough for the linear-time shortest-path algorithm.

Lemma 4.3 The recursive division obtained by the above algorithm satisfies inequality 19.

Proof: First note that combining the inequalities $n_{j+1} \leq n_j/z_j$, we obtain

$$n_j \leq n/\prod_{j<i} z_j \tag{28}$$

Note moreover that each node of G_i expands to at most $\prod_{j<i} 3z_j$ nodes of G.

Consider the division D_i of G_i, and the division of G it induces. The division D_i consists of $O(n_i/z_i^2)$ regions, each consisting of $O(z_i^2)$ nodes and $O(z_i^{1.5})$ boundary nodes. Thus it induces a division of G consisting of $O(n_i/z_i^2)$ regions, each consisting of $O(z_i^2 \prod_{j<i} 3z_j)$ nodes and $O(z_i^{1.5} \prod_{j<i} 3z_j)$ boundary nodes.

Let $r_i = z_i^2 \prod_{j<i} z_j$, and define

$$f(r_i) = z_i^{1.5} \prod_{j<i} 3z_j.$$

Then the induced division of G has $O(n/r_i)$ regions each with $O(r_i 3^i)$ nodes and $O(f(r_i))$ boundary nodes. Since $3^i = O(\prod_{j\leq i} z_j)$, the number of nodes per region is $O(r_i^2)$.[6]

We show that the recursive division induced on G satisfies 19. We have

$$\begin{aligned} \frac{r_i}{f(r_i)} &= \frac{z_i^2}{c_1 z_i^{1.5}} \\ &= \sqrt{z_i}/c_1 \end{aligned}$$

Using the definition of z_i, one can verify that $z_{i-1} = \theta(\log^5 z_i)$ and $\prod_{j<i} z_j = O(\log^6 z_i)$. Hence

$$\begin{aligned} f(r_{i-1}) &= c_1 z_{i-1}^{1.5} \prod_{j<i-1} z_j \\ &= O(\log^{7.5} z_i \log^6 \log z_i) \end{aligned}$$

[6]In fact, this is a very rough bound; the number is not much more than r_i.

We have

$$\log r_{i+1} = O(\log z_{i+1}^2) = O(z_i^{1/5})$$

and consequently $\sum_j^{i+1} \log r_j = O(z_i^{1/5})$. It follows that for a sufficiently large constant $\hat{c}$, inequality 19 is satisfied by r_i's exceeding $\hat{c}$. □

5 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 lengths. Our approach may be seen as an adaption of the approach Frederickson used [Freb] for the non-negative length problem.

Frederickson's approach, which we adopt, is essentially the following.

Step 1: Find an r-division of the graph into regions, where r is a parameter to be determined.[7]

Step 2: Compute, for each region R, shortest-path distances within R between each pair of boundary nodes of R. Let H_R be the complete directed graph whose nodes are the boundary nodes of R and where the length of an edge uv in this complete graph is defined to be the u-to-v distance within the region.

Step 3: Compute shortest paths from a given source s in the graph obtained from the input graph by replacing each region with the complete graph H_R.

Step 4: Propagate shortest-path distance information into the interior of the region, obtaining the distance from s to every node in the region.

Frederickson's algorithm uses the above approach in conjunction with Dijkstra-like searches using a data structure called the topology-based heap. Since our algorithm is intended for graphs with negative lengths, we cannot use Dijkstra-type search. Instead, for step 2 we use the nested-dissection algorithm of Lipton, Rose, and Tarjan, and for step 4 we use the shortest-path algorithm of Goldberg. since the complexity of each of these algorithms differs from that of the corresponding search used by Frederickson's algorithm, we chose the reigon size of the division differently as well.

Next we briefly describe the nested-dissection algorithm of Lipton, Rose, and Tarjan, and how we modify it to suit our purposes. That algorithm uses the separator structure of the graph to obtain a certain ordering of the nodes. Then the nodes are *eliminated* from the graph one at a time in the prescribed order. Eliminating a node consists in forming a complete graph among its neighbors that are still in the graph, and then deleting it from the graph. In the application to computing shortest paths, the step in which a node v is eliminated assigns lengths to the edges of the complete graph among v's neighbors. Namely, for any two edges uv and vw, the elimination step assigns length $\text{length}(uv) + \text{length}(vw)$ to the new edge uw. Finally, if there was already an edge uw, the cheaper of the new edge and the old edge is retained. The elimination step has the property that, for any pair of nodes x and y remaining after the step, the x-to-y distance remains unchanged.

To apply this algorithm to a region, we ensure that the boundary nodes of the region occur last in the ordering. Then we stop the elimination process when the only nodes remaining are the boundary nodes. At this point, we have computed shortest-path distances among all boundary nodes of the region, as required in step 2, and have constructed the complete graph needed for step 3.

[7] His algorithm further subdivides these regions into subregions. Ours does not.

The time required to eliminate a node is proportional to the square of the number of neighbors the node has. Lipton, Rose, and Tarjan show that this elimination process takes time $O(n^{3/2})$ for n-node planar graphs.[8]

Once step 3 of our algorithm has computed the distance from the source node to all the boundary nodes of each region, we can carry out step 4, propagating the shortest-path information to the interior of the region, by running the elimination process backwards. That is, we restore the eliminated nodes in the reverse of the order in which they were eliminated. When a node v is restored, the distance from the source to that node can be computed as the minimum, over all incoming edges uv, of the distance to u plus the length of uv.

Now we prove the correctness of the algorithm.

Lemma 5.1 The distances computed by the above algorithm are correct shortest-path distances.

Proof: First let v be a boundary node. To show the distance to v is correctly computed, we show a correspondence between shortest s-to-v paths P in the input graph G and shortest s-to-v paths P' in H.

Given s shortest s-to-v path P in G, mark nodes of P that are boundary nodes of regions of the r-division. This marking divides P into a sequence of subpaths each of which (except possibly the last) starts and ends with boundary nodes and has no internal boundary nodes. Consider such a subpath. Since it has no internal boundary nodes, it lies entirely within some region R. It therefore corresponds to an edge in the auxiliary graph H_R. Let P' be the path obtained from P by replacing each internal mark-to-mark subpath with the corresponding edge in H.

Conversely, given a shortest s-to-v path P' in H, replace each edge of H with the corresponding path in G, obtaining a path P in G.

We have shown that distances from the source s to boundary nodes v are correctly computed. For any other node x, let v be the last boundary node on a shortest s-to-x path P, and consider the subpath P_v from v to x. Since this subpath contains no boundary nodes after v, it lies entirely within some region R. It follows that in step 4 the distance to x is correctly computed. $\square$

Now we analyze the algorithm. Step 1, finding the r-division, can be done using Frederickson's algorithm in $O(n \log n)$ time. Since each region has $O(r)$ edges and nodes, the nested-dissection algorithm used in step 2 requires $O(r^{3/2})$ time per region in the division. Since there are $O(n/r)$ regions in the division, the total time for step 2 is $O((n/r)r^{3/2})$. Each resulting auxiliary graph H_R has $O(\sqrt{r})$ nodes and therefore $O(r)$ edges. The union H of $O(n/r)$ such graphs 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, where D is the sum of absolute values of the negative lengths. Finally, the time for step 4 is the same as the time for step 2. 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 obtain the following theorem.

Theorem 5.2 Let G be a n-node planar directed graph such that the sum of the absolute values of the negative edge-weights is at most D. For any node s in G the shortest-path distances from s to all the other nodes in G can be found in time $O(n^{4/3} \log^{2/3} D)$.

Next we address the problem of computing a feasible flow. The input is a directed network whose edges are labeled by lower and upper capacities, and whose nodes are labeled by demands (both positive and negative). The goal is to compute a feasible flow, a flow assignment such that the sum of the flow into any node is equal to the demand at that node, and such that, for each edge, the flow across that

[8] Indeed, the algorithm works for any $\sqrt{n}$-separable family of graphs, as long as separators can be found quickly enough.

edge is at least the lower capacity and at most the upper capacity. Miller and Naor [MiN] show if the graph is planar the problem can be solved by doing a single-source shortest-path computation in the dual graph. Thus our algorithm can find a feasible flow if one exists in time $O(n^{4/3}\log^{2/3} D)$ time where D is the sum of all the edge-capacities.

Miller and Naor also point out that since this reduction yields an integral flow if the capacities are integral, the same approach solves the problem of finding a perfect matching in a planar bipartite graph. In this case the capacities are all 1, so the time required by our algorithm is $O(n^{4/3}\log^{2/3} n)$.

Finally, Miller and Naor show that by a series of $\log n$ such computations, one can find a maximum source-to-sink flow. Thus we obtain a bound of $O(n^{4/3}\log n\log^{2/3} D)$ for this problem. More generally, if there are multiple sources and sinks that lie on the boundaries of 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.

5.1 Computing shortest paths in parallel

To get an efficient parallel algorithm we follow the same approach but use somewhat different algorithms. In steps 2 and 4, we use a parallel algorithm due to Cohen [Coh] that takes polylog time and does $O(r^{3/2})$ work for an r-node graph. In step 3, instead of using Golberg's algorithm, we use an algorithm due to Gabow and Tarjan [GaT]. Given an x-node e-edge graph H with integral edge-lengths that are at most N in magnitude, the algorithm in [GaT] computes 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 Phase II 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 5.3 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 $O(n^{2/3}\log^{7/3} n\log(nD))$ and work $O(n^{4/3}\log n\log(nD))$.

We obtain similar bounds for feasible flow and maximum flow using the reduction of Miller and Naor.

6 A fully dynamic data structure for shortest paths in planar graphs with positive and negative edge-lengths

In this section we describe a fully dynamic data structure for maintaining all-pairs shortest paths in a planar directed graph that can have both negative and nonnegative-length edges. As with the sequential algorithm our dynamic algorithm also uses the division-based of Frederickson. We use the cluster-partitioning approach previously used by Frederickson, Galil and Italiano, and others [Frea, GaI, GIS, Sub].

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. When edges are aded or removed we need only recompute the shortest paths in a few pieces.

Throughout this section we assume that the edge-additions preserve planarity. This can be easily enforced within the same time and space bounds by running the dynamic planarity-testing algorithm by Galil, Italiano, and Sarnak [GIS] in the background and only allowing edge-additions that preserve planarity. The edge-additions need not preserve the specific planar embedding since planarity is only required in order to obtain the division, and every so often the algorithm recomputes the division from scratch.

Our dynamic algorithm supports the following operations:

1. **distance**(u, v): Find the distance between u and v in G.

2. **add**(u, v, ℓ): Results in the addition of the edge uv with length ℓ.

3. **delete**(u, v): Results in the deletion of the edge uv.

Our results are stated in the following theorem and its corollaries.

Theorem 6.1 There is a fully dynamic data structure to maintain the all-pairs shortest-path information of a planar graph. Let G be an n-node planar directed graph G such that the sum of the absolute-values of the negative edge-lengths is at most D. The preprocessing to create the data structure from G is $O(n^{10/7})$. The space required by the data structure is $O(n)$. The time per operation is $O(n^{9/7} \log D)$. The time bound 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 a feasible flow in planar graphs. A query determines the amount of flow on a particular edge. The time per operation is $O(n^{9/7} \log C)$, where C is the sum of edge-capacities. This algorithm can in turn be applied to maintain a perfect matching.

We start by finding an r-division of G (the value of r will be determined later). As in the sequential algorithm we use nested dissection to find all-boundary-pair shortest-paths in each of the regions. We also construct the auxiliary graph H as before by unioning the complete auxiliary graphs H_R. Hereafter we will refer to H_R as the *substitute graph* for the region R. Our data structure consists of the auxiliary graph H and the original graph G along with its division.

To find the distance between two given query points u and v, we form a temporary auxiliary graph S by unioning the regions R_1 and R_2 containing u and v along with H. We then run a sequential shortest path algorithm on S to compute the distance between u and v.

Lemma 6.2 The distance between u and v is computed correctly.

Proof: The proof is analogous to the proof of Lemma 5.1. Given s shortest u-to-v path P in G, mark nodes of P that are boundary nodes of regions of the r-division. This marking divides P into a sequence of subpaths each of which (except possibly the first and last) starts and ends with boundary nodes and has no internal boundary nodes. Consider such a subpath. Since it has no internal boundary nodes, it lies entirely within some region R. It therefore corresponds to an edge in the auxiliary graph H_R. Let P' be the path obtained from P by replacing each internal mark-to-mark subpath with the corresponding edge in H. The first subpath starts at u and ends at a boundary node; since it has no internal boundary nodes, it lies entirely within a region, the region containing u. (If u is itself a boundary node, this subpath is trivial.) Similarly the last subpath starts at a boundary node and ends at v, and lies entirely within the region containing u. It follows that P' is a path in the temporary auxiliary graph S. Thus the computed distance in S is at most the actual u-to-v distance in G.

Conversely, given a shortest s-to-v path P' in S, replace each edge of an auxilary graph H_R with the corresponding path in G, obtaining a path P in G. Thus the distance in S is no less than the distance in G. □

To perform an update (whenever an edge is added or deleted, or its cost is changed) we need to recompute the substitute graph of each region that is affected by the change in the input. Lemma 6.3 shows that only a constant number of regions are affected during a single update operation.

Lemma 6.3 Adding, deleting, or changing the length of an edge affects only a constant number of regions in the division. Therefore, only a constant number of substitute graphs need to be recomputed to update H.

Proof: We discuss the case of an edge addition. The arguments for edge deletion and changing edge-lengths are similar. Consider adding the edge uv. If u and v lie in the same region, this addition results in a change to that region. If they lie in different regions, one of the regions is chosen to contain the new edge, and the node not previously in that region becomes a boundary node of that region. Thus that region is changed. Moreover, if that node was previously in only one region, it becomes a boundary node of that region as well, so that region changes. (If the node was previously in more than one region, it was already a boundary node in those regions, so those regions do not change.) □

Repeated requests for adding edges can result in an excess of boundary nodes. Therefore, we re-compute the r-division after the number of *add* operations exceeds the value of a preset parameter $limit$.

We are now ready to discuss our bounds for maintaining all-pairs shortest paths. We set r to be $n^{6/7}$ and $limit$ to be $n^{3/7}$. Just after the r-division is computed, each region has at most $cn^{3/7}$ boundary nodes for some constant c. Since we recompute the division every $n^{3/7}$ adds, each region has at most $(c+1)n^{3/7}$ boundary nodes at any time.

There are $O(n/r)$ regions, each having $O(\sqrt{r})$ boundary nodes. Since $r = n^{6/7}$, the number of boundary nodes is $O(n^{3/7})$. Hence each complete auxiliary graph H_R has $O(n^{3/7})$ nodes and $O(n^{6/7})$ edges. Since there are $O(n^{1/7})$ regions, the auxiliary graph H obtained from the graphs H_R has $O(n^{1/7} \cdot n^{3/7}) = O(n^{4/7})$ nodes and $O(n^{1/7} \cdot n^{6/7}) = O(n)$ edges. The same bounds hold for the size of the auxiliary graph S as well. Therefore, running Goldberg's algorithm [Gol] on S will require $O(n^{9/7} \log D)$ time to find the shortest u-to-v path. This shows that the query time is $O(n^{9/7} \log D)$.

To bound the update time we note that the all-boundary-pair shortest-paths can be computed in $O(n^{9/7})$ time for any region R using nested dissection. Since only a constant number of subgraphs are affected during an update operation, the time for an update is also $O(n^{9/7})$. We also recompute the division[9] and all the substitute graphs once every $n^{3/7}$ *adds*. The time taken to recompute the division is $O(n \log n)$ and the time required to build all the substitute graphs is $O(n^{10/7})$. Amortizing this over $n^{3/7}$ *adds* we find that the amortized time of rebuilding the data structure is $O(n)$ per *add* operation. We therefore get a fully dynamic algorithm for maintaining exact all-pairs shortest paths that requires $O(n^{9/7} \log D)$ time per operation. The bounds of Theorem 6.1 therefore follow.

7 Appendix

At the beginning of Subsection 3.8, in analyzing the algorithm for shortest-paths with nonnegative lengths, we stated what we required of the recursive decomposition in inequality 19. In this appendix, we show how inequalities 20 through 24, used in that subsection, can be derived from inequality 19. We also prove Theorem 3.23, which states that even graphs with fairly poor separators have recursive decompositions satisfying inequality 19.

We start by restating the key inequality, inequality 19.

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

7.1 Derivation of auxiliary inequalities

By simple algebra (reciprocating and cross multiplying to obtain $1/2^i$ on the right hand side) and by substituting $\beta_{i,0} = 4^i \log r_{i+1} / \log r_1$, we rewrite 19 as follows.

$$\beta_{i,0} \frac{f(r_i)}{r_i} f(r_{i-1}) \left(\sum_{j=1}^{i+1} \log r_j \right) \leq \frac{1}{2^i \log r_1}, \tag{29}$$

[9] If necessary, we first recompute a planar embedding

By assuming that $\log r_1 \geq 1$ (i.e., the second lowest level of the decomposition has a region that contains at least 2 nodes) and that $f(r_{i-1}) \geq 1$ (i.e., on each recursive level there is at least one border node), we obtain inequality 22. By noting that $\beta_{i-1,0} < \beta_{i,0}$ we can obtain inequality 24 from 22. By additionally noting that $\left(\sum_{j=1}^{i+1} \log r_j\right) \geq 1$ we obtain inequality 20 from inequality 29.

To derive inequality 21, we use simple algebra in 19 and substitute in $\alpha_i = 4\log r_{i+1}/\log r_i$ to obtain

$$\alpha_i \frac{f(r_i)}{r_i} f(r_{i-1}) \left(\sum_{j=1}^{i+1} \log r_j\right) \leq \frac{4}{8^i \log r_i}, \tag{30}$$

By noting that $\left(\sum_{j=1}^{i+1} \log r_j\right) \geq \log r_{i+1}$, that $\log r_i \geq 1$ (for r_i exceeding some constant) and that $f(r_{i-1}) \geq 1$, we obtain inequality 21.

Finally, inequality 23 holds (even without condition 19) because $f(r_i) \geq f(r_{i-1})$ (since f is nondecreasing) and because $\beta_{i,0} \geq 4\beta_{i-1,0}$.

7.2 The proof of Theorem 3.23

We restate and prove Theorem 3.23

Theorem 7.1 For any f such that $f(k) = O(k/2^{\log^\epsilon k})$ where ϵ is a constant, for any subgraph-closed f-separable 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.

Proof: We will use a $(\bar{r}, f)$ recursive division for a sequence $\bar{r}$ that we define in terms of a function g to be specified later. Let $\bar{r} = (r_0, r_1, \ldots, r_s)$ be defined by $r_s = n$, $r_{i-1} = g(r_i)$. For a nonnegative integer i, we use g^i to denote the i-fold composition of g with itself, i.e. $g^0(k) = k$, $g^i(k) = g(g^{i-1}(k))$.. Define $g^{-1}(k)$ to be $\max\{i : g(i) = k\}$. Finally, define $g^*(k)$ to be $\max\{i : g^i(k) = 1\}$. Since $r_i = g(r_{i+1})$ and $f(x) \leq x$, we can ensure that equation 19 holds if for all k the following inequality holds.

$$\frac{k}{f(k)} \geq 8^{g^*(k)} g(k) \log g^{-1}(k) \left(\sum_{j=0}^{g^*(k)} \log g^{-1+j}(k)\right)$$

Since $g(k) \leq k$, we can infer $\log g^{-1+j}(k) \leq \log g^{-1}(k)$ for any positive j. Hence the inequality above is implied by the following inequality.

$$\frac{k}{f(k)} \geq 8^{g^*(k)} g(k) g^*(k) \log^2(g^{-1}(k)) \tag{31}$$

Let $g(k) = 2^{c\log^\epsilon k}$. Then $g^*(k) = O(\log\log k)$, $8^{g^*(k)} = O(\log^{O(1)} k)$, and $\log g^{-1}(k) = O(\log^{1/\epsilon} k)$. Let $\bar{r} = (r_0, r_1, \ldots, r_s)$ be defined by $r_s = n$, $r_{i-1} = g(r_i)$. By Lemma 2.2, there exists an $(\bar{r}, f)$ recursive division. In order to prove inequality 31, we need to prove

$$\begin{aligned} 2^{\log^\epsilon k} &\geq 2^{c\log^\epsilon k}(\log k)^{O(1)+2/\epsilon} \\ &= 2^{c\log^\epsilon k + O(\log\log k) + 2\log\log k/\epsilon}. \end{aligned}$$

By an appropriate choice of c, this inequality can be made to hold for sufficiently large k. □

Acknowledgements

Many thanks to the referees for their remarkably thorough review and very helpful suggestions.

References

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

[AST] N. Alon, P. Seymour & R. Thomas, "A Separator Theorem for Nonplanar Graphs," *Journal of the American Mathematical Society* 3 (1990), 801–808.

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

[Frea] G.N. Frederickson, "Data structures for on-line updating of minimum spanning trees, with applications," *SIAM J. Computing* 14 (1985), 781–798.

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

[Frec] G.N. Frederickson, "An optimal algorithm for selection in a min-heap," *Information and Computation* 104 (1993), 197–214.

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

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

[GaT] H. N. Gabow & 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).

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

[GIS] Z. Galil, G.F. Italiano & N. Sarnak, "Fully dynamic planarity testing," *Proc. 24th Annual ACM Symposium on Theory of Computing* (1992).

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

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

[Gol] A.V. Goldberg, "Scaling algorithms for the shortest path problem," *SIAM Journal on Computing* 24 (1995), 494–504.

[Goo] M. Goodrich, "Planar separators and parallel polygon triangulation," *Journal on Computer and System Sciences* (1995).

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

[HaJ] R. Hassin & D. B. Johnson, "An $O(n \log^2 n)$ algorithm for maximum flow in undirected planar networks," *SIAM Journal on Computing* 14 (1985), 612–624.

[Joh] 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.

[JoV] D.B. Johnson & 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).

[KPS] P. Klein, S. Plotkin, C. Stein & É. Tardos, "Faster Approximation Algorithms for the Unit Capacity Concurrent-Flow Problem With Applications to Routing and Finding Sparse Cuts," *SIAM Journal on Computing* 23 (1994), 466–487.

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

[LRT] R. Lipton, D. Rose & R.E. Tarjan, "Generalized nested dissection," *SIAM J. Numerical Analysis* 16 (1979), 346–358.

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

[MiN] G. L. Miller & J. Naor, "Flows in planar graphs with multiple sources and sinks," *SIAM Journal on Computing* (1995).

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

[PST] S. A. Plotkin, D. B. Shmoys & É. Tardos, "Fast Approximation Algorithms for Fractional Packing and Covering Problems," *Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science* (1991).

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

[Tho] M. Thorup, "On RAM priority queues," *Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms* (1996).