A fully dynamic approximation scheme for shortest paths in planar graphs

Philip N. Klein* Sairam Subramanian†
Brown University Brown University

Abstract

In this paper we give a fully dynamic approximation scheme for maintaining all-pairs shortest paths in planar networks. Given an error parameter ϵ such that $0 < \epsilon$, our algorithm maintains approximate all-pairs shortest-paths in an undirected planar graph G with nonnegative edge lengths. The approximate paths are guaranteed to be accurate to within a $1+\epsilon$ factor. The time bounds for both query and update for our algorithm is $O(\epsilon^{-1} n^{2/3} \log^2 n \log D)$, where n is the number of nodes in G and D is the sum of its edge lengths. The time bound for the queries is worst case, while that for the adds is amortized.

Our approximation algorithm is based upon a novel technique for approximately representing all-pairs shortest paths among a *selected* subset of the nodes by a *sparse substitute graph.*

*Research supported by NSF grant CCR-9012357 and NSF PYI award CCR-9157620, together with PYI matching funds from Thinking Machines Corporation and Xerox Corporation. Additional support provided by DARPA contract N00014-91-J-4052 ARPA Order 8225.

†Research supported in part by a National Science Foundation Presidential Young Investigator Award CCR–9047466 with matching funds from IBM, by NSF research grant CCR–9007851, by Army Research Office grant DAAL03–91–G–0035, and by the Office of Naval Research and the Defense Advanced Research Projects Agency under contract N00014-91-J-4052 and ARPA order 8225.

1 Introduction

An algorithm for a given graph problem is said to be *dynamic* if it can maintain the solution to the problem as the graph undergoes changes. These changes could be additions or deletions of edges, or a change in the cost of some edge (if applicable). In such a setting an *update* denotes an incremental change to the input, and a *query* is a request for some information about the current solution. We expect the dynamic algorithm to handle both queries and updates quickly i.e. in time that is substantially less than it would take to solve the problem from scratch every time the input changes. An algorithm is said to be *fully dynamic* if it supports both additions and deletions of edges. while it is said to be *semidynamic* if it supports only one of them. Unfortunately, due to the requirement that the query and update times be very small, designing fully dynamic algorithms seems to be considerably harder than designing sequential algorithms, and very few graph problems have fully dynamic solutions.

In this paper we consider the problem of maintaining shortest-path information in planar graphs. This is a fundamental optimization problem since many applications can be formulated as shortest-path problems. Furthermore, a number of more complex problems can be solved by procedures which use shortest-path algorithms as subroutines.

Given a graph G with n nodes and m edges (with non-negative weights) the shortest path between any two nodes can be computed efficiently by using Dijkstra's algorithm [1, 2] in $O(m + n \log n)$ time. For planar graphs a faster algorithm due to Frederickson [3] runs in $O(n\sqrt{\log n})$ time. Recently in joint work with Rao and Rauch [4] we have given an $O(n)$-time algorithm for computing single-source shortest paths. However, in the dynamic realm this problem is much less well-understood. Though there are many algorithms for the dynamic problem (see for example [5, 6, 7], see also [8]), none of them can simultaneously handle both updates and queries in time that is sublinear in the input size.

Definition 1 Let G be an n-node planar undirected graph with nonnegative integral edge-lengths. Let D be the sum of lengths. The *length* of a path π from u to v (denoted as $l(\pi)$) is simply the sum of the lengths of the edges in π. A minimum-length path from u to v is called a shortest path.

The all-pairs shortest path problem is the problem of finding shortest paths between all pairs of nodes in G.

Given the difficulty in constructing dynamic algorithms for shortest paths, in this paper, we focus on constructing dynamic algorithms for maintaining "approximate shortest paths."

Definition 2 A path π is an ϵ-approximate shortest-path if its length is at most $1+\epsilon$ times the distance (the length of the shortest path) between its endpoints.

In this paper we show that if we are willing to settle for approximate answers then substantial improvements are possible in both the query and update times for maintaining shortest paths in a planar graph. In particular, we give a fully dynamic data structure that maintains ϵ-approximate shortest-paths. Both query and update times for maintaining our data structure are sublinear in n when ϵ^{-1} is no more than a poly-logarithmic function of n.

Theorem 1 *Let G be an undirected n-node planar graph with non negative weights on its edges such that the sum of the edge weights D is $O(\text{exponential}(n))$. Then, for any $0 < \epsilon \leq 1$ such that ϵ^{-1} is $O(\text{polynomial}(n))$, there exists a fully dynamic data structure to maintain ϵ-approximate all-pairs shortest-path information in G. The time per operation is $O(\epsilon^{-1} n^{2/3} \log^2 n \log D)$. The time for queries, edge-deletion, and changing lengths is worst-case, while the time for adding edges is amortized.*

Our approximation algorithm is based upon a novel technique for compactly representing approximate all-pairs shortest paths among a set of k *selected* nodes by a substitute graph with the following properties:

- Each edge uv in the substitute graph corresponds to a path π from u to v in G.
- Each shortest path between selected nodes in G is approximated to within a $1+\epsilon$ factor by a two-edge path in the substitute graph.

The size of the substitute graph depends both on the number of selected nodes and on their distribution over the faces in G. In particular, given an n-node undirected planar graph with nonnegative edge-lengths that sum to D, we have the following bounds on the size of sparse substitutes:

Theorem 2 (Face-boundary substitute) *If there are $k = O(\sqrt{n}$ selected nodes, all on the boundaries of a constant number of faces of G then there exists a sparse substitute graph having $O(\epsilon^{-1}k \log k \log D)$ edges and $O((\epsilon^{-1} + k) \log k)$ nodes that approximates the all-selected-node-pair shortest paths in G to within a $1 + \epsilon$ factor. Furthermore, the substitute graph can be constructed in $O(\epsilon^{-1} n \log^2 n \log D)$ time.*

2 Preliminaries

In this section we introduce some basic terminology regarding planar graphs. More details and related background can be found in [18, 9, 3, 11]

Definition 3 A graph G is said to be planar if we can embed the nodes and edges of the graph on the plane such that no two edges cross each other.

Definition 4 A *cycle* in G is a set of nodes $u_1, u_2, \cdots, u_k$ such that $u1 = u_k$ and u_i is connected to u_{i-1} and u_{i+1}. A cycle $C = \{u_1, u_2, \cdots, u_k\}$ is a simple cycle if all the nodes u_1 through u_{k-1} are distinct. Given an embedding of G on the plane a simple cycle $c = \{u_1, \cdots, u_k\}$ is called a face if no other nodes of g are topologically embedded inside c in the embedding.

Definition 5 A set X of nodes in G is called a separator if the removal of X divides G into two or more disconnected pieces. The nodes of X are called *boundary* nodes.

Lipton and Tarjan [12] showed that given an n-node planar graph G and given a subset B of the nodes of G, in linear time one can find a separator such that $|X| = O(\sqrt{n})$, and none of the pieces created by the removal of X has more than $\frac{2}{3}$ of the nodes from B. Such a separator X is called a *balanced separator* of B. Miller [13] showed that if G is two-connected and triangulated then we can find a balanced separator X that is a simple cycle. If G is not two-connected we can first two-connect it in the following manner:

- Make G connected by adding edges between all the disconnected pieces.
- Add one dummy node per face of the original graph, with dummy edges connecting the dummy node to the nodes on the boundary of the original face.

After this process none of the faces will have more than 3 boundary nodes; thus the resulting graph is triangulated.

Lemma 1 *Given a planar graph G the process of adding dummy nodes and edges described above results in a triangulated two-connected graph.*

Proof: The second step results in a triangulated graph as noted above. To prove that it is two-connected assume for a contradiction that G is not two-connected after the addition of the dummy nodes and edges. This implies that there is a node u that is a separator. Let the two components that u separates be C_1 and C_2. Let f be the outer face containing u and nodes from C_1 and C_2. By construction f can have only 3 nodes. Therefore C_1 and C_2 (since they are both non-empty) each contribute one node (x_1 and x_2 respectively) to f. This implies C_1 and C_2 must contain only one node each. Otherwise more than one node of C_1, and C_2 will be on the face f. However, this leads us to our base case since one of u, x_1, or x_2 is a dummy node. Thus the resulting graph is two-connected and triangulated. □

Thus we can use Miller's algorithm to find cycle separators for graphs that are not necessarily two-connected and triangulated.

2.1 Cluster Decompositions

Frederickson [9, 3] showed how to construct dynamic algorithms for graph problems by dividing a graph into *clusters*. Such a division is called a *cluster decomposition*. Frederickson [9] used the clustering idea to construct a fully dynamic data structure for maintaining minimum spanning trees in general graphs. In the context of planar graphs [3] he used a separator based cluster decomposition (obtained by repeated division of the graph using separators) to derive improved sequential algorithms for single-source shortest paths.

Galil, Italiano, and Sarnak [10, 11] used the separator algorithm due to Lipton and Tarjan [12] to repeatedly divide the underlying planar graph into clusters. Galil and Italiano [10] used such a decomposition to derive a fully dynamic data structure for maintaining two and three-vertex connectivity information in planar graphs. Galil, Italiano, and Sarnak [11] used cluster decompositions to develop a fully dynamic planarity-testing algorithm. We borrow this technique and their terminology. For reasons that will be apparent soon, in this paper we will use the planar separator algorithm by

Miller [13] to construct our decomposition. Our dynamic shortest-path data structure will require the decomposition of the given planar graph G into clusters. However, we will use a somewhat restrictive notion of what is a decomposition of a planar graph G into clusters.

Definition 6 A *cluster partition* of a graph G is a partition of the edges of G into edge-induced subgraphs. A node of G is a *boundary node* of the partition if it belongs to more than one subgraph. A *num-cluster partition* of an n-node planar graph G is a cluster partition of G into $r = O(num)$ subgraphs $G_1, G_2, \cdots, G_r$ with the following properties:

1. Each subgraph G_i contains $O(n/num)$ edges.
2. The number of boundary nodes in each G_i is $O(\sqrt{n/num})$.
3. In each subgraph G_i, the boundary nodes all lie on the boundaries of a constant number of *faces*. (Note that a face of G_i need not be a face of G.)

Note that because the subgraphs of a cluster partition are edge-induced, a node belongs to such a subgraph only if an edge incident to the node belongs to the subgraph. The next lemma follows from the arguments of Frederickson [3].

Lemma 2 *Given a planar graph G, a num-cluster partition can be obtained in $O(n \log n)$ time.*

Proof: To create a cluster decomposition of G we two-connect and triangulate it as described above by adding dummy nodes and edges. By applying Miller's algorithm to this graph, we can obtain a cycle separator that divides the non-dummy nodes of the graph into pieces none of which have more than $\frac{2}{3}n$ nodes. The separator is a cycle that contains both dummy and non-dummy nodes. The set X of non-dummy nodes of the separator need not form a cycle in the original graph. However, it does divide the graph into two pieces G_1 and G_2 such that such that, in the induced subgraph $H_i = G_i \cup X$ the nodes of X all lie on the boundary of a single face. The dummy nodes do not play any role in the data structure, and are used only to divide up the graph.

To obtain a num cluster partition we start with with G and repeatedly divide it using a planar cycle separator [13] (as discussed above) until all the pieces have $O(n/num)$ edges. We re-triangulate subgraphs when faces get too big so that we are guaranteed to have small cycle separators. As mentioned earlier, these dummy nodes and edges are transient and play no role in the actual data structure.

Using techniques due to Frederickson [3] we can also make sure that none of the pieces have too many boundary nodes. This is accomplished as follows: The separator algorithm can be used to separate a node-weighted version of the graph into pieces none of which has more than two-thirds the original weight. In order to split the boundary nodes we give all the non-boundary nodes weight 0 and give weight 1 to the boundary nodes. We then run the separator algorithm to find a weighted separation. This automatically gives us a division of the boundary nodes. Proceeding in this fashion we can generate a num-cluster partition. □

Definition 7 Consider a num-cluster partition of G. In such a partition we define the *parent* of edge uv (denoted by G_{uv}) to be the subgraph G_i that contains it. Similarly, if u is a non-boundary node, we define its parent (denoted G_u) to be the subgraph G_i containing it. If u is a boundary node, we arbitrarily select one of the subgraphs G_i that contains u, and assign it to be u's parent (denoted by G_u).

The remainder of the paper is organized as follows: In Section 3 we describe our dynamic data structure for mainitaining approximate shortest paths. In Section 4 we address the issue of constructing face-boundary substitutes and in Section 5 we discuss some extensions of our algorithm.

3 A fully dynamic data structure for approximate shortest paths

In this section we describe our dynamic data structure for maintaining approximate shortest paths that satisfies the bounds of Theorem 1. Our data structure uses the face-boundary substitutes of Theorem 2. Section 4 gives the details of how the face-boundary substitutes are constructed.

Throughout this paper we assume that all the edge-additions are planarity-preserving. To see whether edge-additions preserve planarity we

procedure preprocess(G, num)
  [Division]
    Find a num-cluster partition of G into regions $G_1, G_2, \cdots, G_r$
    each with n/num nodes and $O(\sqrt{n/num})$ boundary nodes.

  [Local Computation]
    for each region G_i **do**
      construct a substitute graph $\hat{G}_i$ representing
      boundary-to-boundary shortest paths in G_i.

  [Forming the Skeleton]
    $S \leftarrow \hat{G}_1 \cup \hat{G}_2 \cup \cdots \cup \hat{G}_r$.
  end [procedure preprocess]

Figure 1: The generic preprocessing step.

can run the planarity-testing algorithm from [11] in the background to prevent addition of edges that destroy planarity. Doing this only increases the time-complexity of our update operations by a constant factor. In the descriptions of our algorithms we will not explicitly mention these additional steps.

Our data structure supports the following operations:

1. **distance**(u, v): Find the approximate distance between u and v in G.
2. **add**(u, v, w): Add a new edge uv of length w.
3. **change**(uv, w): Change the length of the edge uv to w.
4. **delete**(uv): Delete edge uv.
5. **remove**(u): Remove an isolated node (a node that has no edges) u.

Figures 1, 2, and 3 give the preprocessing, query, and update routines. The procedure for removing an isolated node u involves no change to the data structure. We just remove it from the corresponding parent cluster G_u

To initialize or data structure, we find an num-cluster partition of G (the optimal value for num will be derived later), and precompute *substitute graphs* $\hat{G}_1, \hat{G}_2, \cdots, \hat{G}_r$ that approximately represent the boundary-to-boundary shortest paths in the respective subgraphs $G_1, G_2, \cdots, G_r$. The $\hat{G}_i$ are face-boundary substitutes as described in Theorem 2 that approximate the boundary-to-boundary shortest paths in G_i to within a $1+\epsilon$ factor. These substitute graphs are then unioned to form a skeletal graph S.

The skeletal graph S is a compact representation for the shortest-paths among the boundary nodes and is used in the query-stage to compute the distance between the two query points.

To answer a query concerning the distance between two given nodes u and v, we form an auxiliary graph H by unioning the regions G_u and G_v along with the skeletal graph S. We then run a sequential shortest-path algorithm on the auxiliary graph H to compute the distance between u and v. See Figure 2 for the query-procedure **distance**.

procedure distance(u, v)
  [Forming the auxiliary graph]
    $H \leftarrow G_u \cup G_v \cup S$.

  [The Query]
  1. Run Dijkstra's algorithm in H with u as the source node.
  2. Return the distance between u and v found in the previous step.
  end [procedure distance]

Figure 2: Performing a query using the skeleton S constructed in the preprocessing step.

To prove the correctness of our query-procedure we need to show that the distance between u and v in H is an accurate estimate of the distance between u and v in G, consider a path π of minimum length from u to v in G. Mark all the boundary nodes in π. This marking divides π into a sequence of subpaths such that the first and the last subpaths lie entirely within G_u and G_v respectively, and each intermediate subpath is a boundary-to-boundary path that lies entirely within one of the subgraphs G_1 through G_r. Since the boundary-to-boundary shortest paths in G_i are estimated by the substitute graph $\hat{G}_i$ to within a $1+\epsilon$ factor, it follows that H contains a path from u

to v whose length is no more than $1+\epsilon$ times the length of π.

To perform an add operation we add a new edge uv of length w to the parent region G_u of u. Similarly to delete an edge uv we simply remove it from its parent region G_{uv}. To implement the **change** operation we simply change the weight of uv in G_{uv} to w. For each of these three operations we need to recompute the substitute graphs of the affected regions. We do this using a procedure **update** that recomputes the substitute graphs for all the regions that are affected by the change. Figure 3 gives the pseudocode for procedure **update**. Lemma 3 shows that only a constant number of subgraphs are affected during a single add, delete, or change operation.

procedure update(u, v)
  [Changing the boundary information]
    if u or v change their boundary status **then**
      change their labels in G_u and/or G_v;

  [Rebuilding the skeletal graph]
  1. Remove $\hat{G}_u$, $\hat{G}_v$, and $\hat{G}_{uv}$ from S;
  2. Recompute the substitutes $\hat{G}_u$, $\hat{G}_v$, and $\hat{G}_{uv}$;
  3. Add these substitutes to S and rebuild it
end [procedure update]

Figure 3: Updating the data structure

Lemma 3 *Adding, deleting, or changing the length of an edge affects only a constant number of subgraphs in the cluster partition. Therefore, only a constant number of substitute graphs need to be recomputed to modify the skeletal graph S.*

Proof: We discuss the case of a *delete* operation. The arguments for addition and changing edge-lengths are similar. Consider deleting the edge uv. This results in a change in the subgraph G_{uv}. Furthermore, it is possible that deleting this edge could make either u (or v) a non-boundary node (this could happen if all the remaining edges incident at u (v) now lie in a single subgraph). Such a change in the boundary-status can affect only G_u and G_v. Therefore, to modify S we need only recompute $\hat{G}_{uv}$, $\hat{G}_u$, and $\hat{G}_v$. □

As we continue to add edges more and more nodes will become boundary nodes, since every time an edge uv is added either u or v may become a boundary node. This will cause the skeletal graph S to grow in size thus increasing the query time. Also the cluster decomposition will start losing its properties, since all the added edges could end up in the same region. We therefore recompute the cluster partition after the number of *add* operations exceeds the value of a preset parameter *limit*. This will imply that the time for the add operation is amortized.

We are now ready to discuss our bounds for maintaining approximate shortest paths. We first discuss the query time assuming the substitute graphs and the skeletal graphs obey the restrictions of Theorem 2. Subsection 3.1 discusses the details of the periodic re-computation of the cluster decompositions and the time required for the update operations.

To maintain approximate shortest paths we set both *num* and *limit* to be $n^{1/3}$. For each G_i we let the substitute graph $\hat{G}_i$ be the face-boundary-substitute that represents the all-boundary pair shortest paths in G_i to within a $1+\epsilon$ factor.

To maintain approximate shortest paths we follow the description of the generic algorithm except for one difference: Since the size of the face-boundary substitutes depends on the distribution of the boundary nodes we modify our updating procedure slightly. Whenever the number of *boundary faces* (faces that contain boundary nodes) in some cluster G_i exceeds three, we apply Miller's separator algorithm to the graph with dummy nodes added. In order to reduce the number of boundary faces, we first give two-connect and triangulate the graph as before. We then give a weight of 1 to the dummy nodes corresponding to the boundary faces. All the other nodes are given weight 0. We then use Miller's algorithm to find a separator X that divides up the dummy nodes corresponding to boundary faces. Each of the two resulting pieces has at most two-thirds of the four boundary faces, hence at most two old boundary faces. The new separator introduces an additional boundary face, for a total of three per piece. If the separator includes dummy nodes corresponding to old boundary faces, then in the resulting pieces these faces are merged with the new boundary face. Hence each piece ends up with at most three boundary faces. This modification of the update procedure is shown in Figure 4.

To bound the time taken for a query let us consider the size of the skeletal graph S at any time during the computation. Initially S is composed

[**Modified version of skeletal graph rebuilding**]
1. Remove $\hat{G}_u$, $\hat{G}_v$, and $\hat{G}_{uv}$ from S.
2. if $G_u(G_v)$ has more than 3 boundary-faces **then**
Use a cycle separator to split them into smaller subgraphs with boundary nodes on at most 3 faces.
3. Construct substitute graphs for G_{uv}, G_u, and G_v, or for their pieces if they were divided in the previous step.
4. Add these substitutes to S and rebuild it.
end [**modification**]

Figure 4: Modifying the rebuilding step in procedure **update**

of the face-boundary substitute graphs $\hat{G}_1, \hat{G}_2, \cdots, \hat{G}_r$. Since $num = n^{1/3}$, the size of G_i for $1 \leq i \leq r$ is $O(n^{2/3})$. Also by the definition of a num-cluster partition the number of boundary nodes in G_i is $O(n^{1/3})$. Therefore the size of $\hat{G}_i = O(\epsilon^{-1}(n^{1/3} \log n \log D)$ (see Theorem 2). By definition of the num-cluster partition $r = O(num) = O(n^{1/3})$. Therefore $|S| = O(\epsilon^{-1}n^{2/3} \log n \log D)$. Also the number of edges in the regions G_u and G_v is $O(n^{2/3}$. Therefore the size of $H = S \cup G_u \cup G_v = O((\epsilon^{-1}n^{2/3} \log n \log D)$. Running Dijkstra's algorithm on H therefore requires $O((\epsilon^{-1}n^{2/3} \log^2 n \log D)$ time.

3.1 Recomputing the Cluster Decomposition after multiple additions

As discussed earlier successive *add* operations cause new nodes to be labeled boundary nodes, increasing the size of S. If an *add* operation results in a split as described, then we have to recompute the face-boundary substitutes of up to three subgraphs G_u, G_v, and G_{uv}. The edge additions preserve planarity therefore each of the corresponding substitutes adds at most $O(\epsilon^{-1}n^{1/3} \log n \log D)$ edges to S. Since $limit = n^{1/3}$ the total number of edges added before the cluster partition is recomputed is $O(\epsilon^{-1}n^{2/3} \log n \log D)$. Since the auxiliary graph H is constructed from S by unioning S with G_u and G_v, it also obeys the same bound on the number of edges. Thus an execution of Dijkstra's algorithm on H takes $O(\epsilon^{-1}n^{2/3} \log n \log D(\log n + \log \log D + \log(\epsilon^{-1}))$ time. If we assume that

ϵ^{-1} is at most a polynomial in n and that D is no more than exponential in n then we get the bounds of the query time in Theorem 1.

To bound the update time we note that by Theorem 2 the face-boundary substitute for any region G_i can be found in $O(\epsilon^{-1}|G_i|\log^2|G_i|\log D)$ time. Since $limit = O(n^{1/3})$ the size of G_i at any time in between two global computations of the cluster partition is $O(n^{2/3}+n^{1/3}) = O(n^{2/3})$. Therefore, the substitute graph $\hat{G}_i$ can be constructed in $O(\epsilon^{-1}n^{2/3}\log n^2 n\log D)$ time. Thus, the time needed to modify the data structure for any update operation is $O(\epsilon^{-1}n^{2/3}\log^2 n\log D)$.

We also recompute the cluster-partition and all the substitute graphs once every $n^{1/3}$ **add** operations. The time taken to recompute the cluster partition and to build all the substitute graphs is $O(n^{1/3}) \times O(\epsilon^{-1}n^{2/3}\log^2 n\log D) = O(\epsilon^{-1}n\log^2 n\log D)$. Amortizing this over $n^{1/3}$ **add** operations gives us the bounds of Theorem 1.

4 Constructing a face-boundary sparse substitute

In this section we address the issue of constructing a face-boundary substitute to approximately represent the all-pairs shortest paths in G among a set N of $O(\sqrt{n})$ selected nodes (distributed over a constant number of faces).

In Subsection 4.1 we describe a basic sparsification technique that is the key to the construction of our substitutes, and in Subsection 4.2 we give a simple divide-and-conquer procedure that repeatedly uses the sparsification technique to construct a face-boundary substitute.

4.1 A basic sparsification technique

Let ϵ an error parameter and d a distance parameter, and let P be a path in G of length $O(d)$. In this subsection we show how to sparsely represent selected node-pair shortest paths that intersect P and are between d and $2d$ in length. In particular, we show that there exists a substitute graph with $O(\epsilon^{-1}k)$ edges that approximates these shortest paths to within a $1+\epsilon$ factor. We call this substitute a *crossing substitute.*

To construct the crossing substitute we proceed as follows: We first divide P into $O(\epsilon^{-1})$ node-disjoint segments of length at most $\epsilon d/2$ each. The first

procedure basic-sparse
  [Dividing the separating path ρ]
    1. Partition ρ into $O(\epsilon^{-1})$ segments $s_1, s_2, \ldots, s_k$ of length at most $\epsilon d/2$.
    2. **let** x_i be the first node of segment s_i.
    3. **for** $1 \leq i \leq k$ **do**

      [Sparsifying paths that cross segment s_i]
      a. Compute shortest paths from x_i.
      b. For each boundary node u add the edge ux_i (to the substitute),
          with the shortest-path length found in Step **a.**
        end [Sparsification]
end [procedure basic-sparse]

Figure 5: The basic sparsification technique

node x_i in each segment s_i is called the *segment node* of that segment. We now perform single-source shortest-path computations in G from each of the $O(\epsilon^{-1})$ segment nodes. Our crossing substitute consists of a collection of *stars*, one for each segment. The star for segment s_i has x_i as its center and the selected nodes as its leaves. The edge between a selected node u and x_i is labeled with the length of the shortest path from x_i to u. A pseudocode version of the basic sparsification procedure is given in Figure 5.

Lemma 4 *Let CS be the crossing substitute constructed as described above. Then, for any path $\pi \in G \mid d \leq l(\pi) \leq 2d$ between selected nodes that intersects P there is a corresponding two-edge path $\hat{\pi} \in CS$ between the same endpoints, such that $l(\hat{\pi}) \leq (1+\epsilon)l(\pi)$. Furthermore, $|CS| = O(\epsilon^{-1}k)$ and the time required to compute CS is $O(\epsilon^{-1}n)$.*

Proof: To prove our claim let the endpoints of π be the selected nodes u and v. Since π intersects P, as shown in Figure 6, there is an alternate path between the endpoints u and v that detours through x_i along s_i. Since the length of s_i is $\epsilon d/2$ the additional length acquired because of the detour is at most $2\epsilon d/2 = \epsilon d$.

In the substitute we approximate π by two edges, ux_i and x_iv. Since ux_i and x_iv represent shortest paths between their respective endpoints, the length of the two-edge path $\hat{\pi} = ux_i, x_iv$ is no more than the alternate path

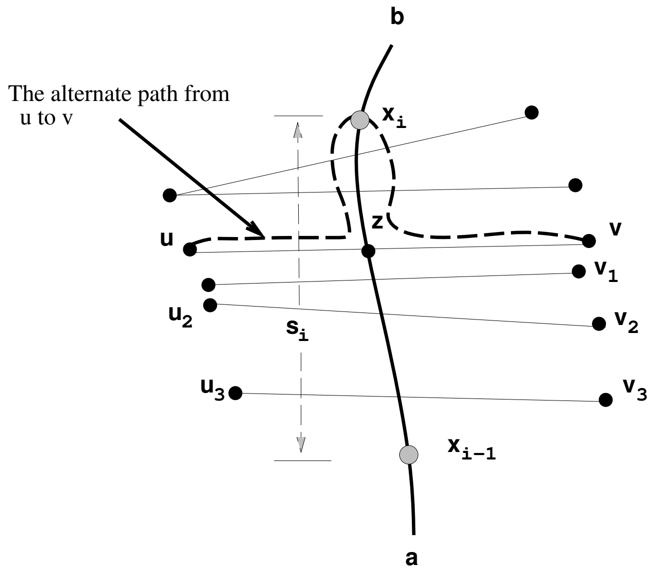

Figure 6: Approximating a path that passes through segment s_i

of Figure 6. Therefore, $\hat{\pi}$ is at most ϵd longer than π. This combined with the fact that π is at least d in length implies that $\hat{\pi}$ is a $1+\epsilon$ factor approximation of π.

To bound the number of edges in our substitute we note that it consist of $O(\epsilon^{-1})$ stars each of which has at most k edges. The bound on the size therefore follows. To bound the time required to compute the crossing substitute we note that the most expensive step is the single-source shortest path computations from the segment nodes. Since there are $O(\epsilon^{-1})$ segment nodes, the entire computation can be carried out in $O(\epsilon^{-1}n)$ time using the shortest-path algorithm of [4]. □

4.2 Face-boundary substitutes

Let N be a set of k *selected* nodes in G that lie on a constant number of faces, and let $\epsilon > 0$ be an error parameter. In this section we show how to construct a face-boundary substitute that approximates the all-pairs shortest paths between selected nodes to within a $1 + \epsilon$ factor.

Here we describe a procedure for sparsification when all the selected nodes lie on the boundary of a single face f. The case of multiple faces is similar. Our sparsification algorithm uses a divide-and-conquer mechanism and makes use of the basic sparsification technique from subsection 4.1. The basic sparsification technique works best with paths that are all roughly of the same length. To accommodate this, we use a *grouping* technique (see, e.g., [14]).

We consider the selected-node-pair shortest paths in $\log D$ different groups and for each group we build a different substitute. The ith substitute approximates selected-node-pair shortest paths in the range $[d, 2d]$, where $d = 2^i$. These $\log D$ substitutes are unioned to get a substitute that sparsely represents paths of all lengths.

To sparsely represent selected-node-pair shortest paths with lengths in the range $[d, 2d]$ we use the divide-and-conquer procedure described below. A pseudocode version of the sparsification procedure is give in Figure 8.

1. **Find separating paths:** We use a procedure called **separate** that gives a separating set Z consisting of one or two paths of length at most $4d$. These paths divide N into subsets N_1 and N_2 of size at most $3|N|/4$ each such that every path of length at most $4d$ between them intersects some path in Z (see Figure 7).

2. **Sparsify intersecting paths:** To sparsely represent shortest paths that cross one of the separating paths we use the basic sparsification technique from subsection 4.1.

3. **Divide G for the recursion:** For each node x in G we determine whether there is a path of length at most $2d$ between x and some node in N_1 (respectively N_2) that does not intersect any of the separating paths in Z. If there is such a path then we place x in V_1 (respectively V_2). We construct G_1 and G_2 by taking node-induced subgraphs of V_1 and V_2 respectively.

   Note that no node can be in both V_1 and V_2 at the same time. Otherwise, we would get a path of length at most $4d$ between N_1 and N_2 that does not intersect any separating path.

   To find the nodes of G that go into V_1 we combine all the nodes of N_1 into a single supernode S and find the nodes of G that are within a distance $2d$ from S in $G - Z$. The set V_2 is determined similarly.

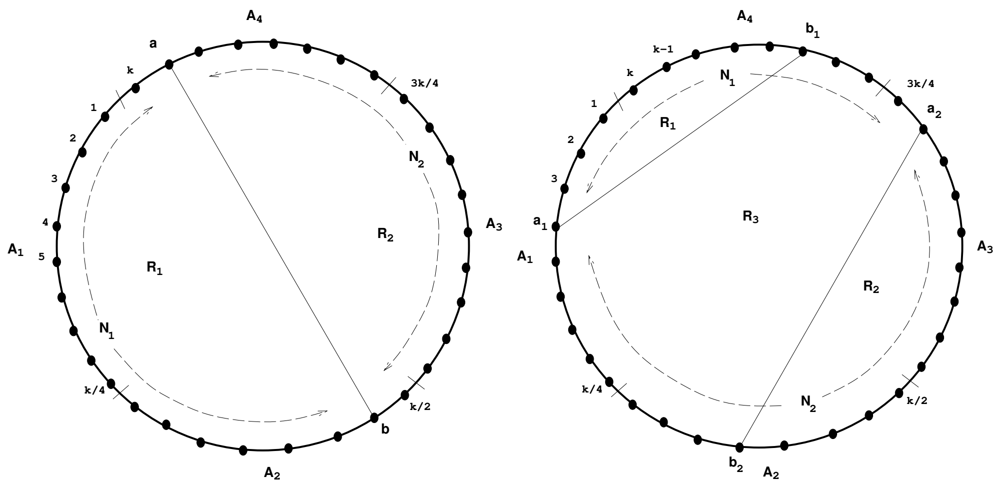

Figure 7: Using one or two paths to cut the boundary

4. **Recursive computation:** Recursively compute sparse substitutes for $[d, 2d]$-shortest paths in G_1 and G_2 with both endpoints in N_1 and N_2 respectively. Union the two recursive substitutes along with the crossing substitute found in step 2 to get the face-boundary substitute for all the selected nodes.

Before we analyze our procedure for sparsification we describe procedure **separate**. Consider a division of N into four subsets A_1, A_2, A_3, and A_4 as shown in Figure 7. If there is a path of length at most $4d$ from A_1 to A_3 or A_2 to A_4 then we can use it to divide N into subsets N_1 and N_2 as shown in Figure 7. Note that the nodes of N_1 are topologically separated from those of N_2 by the separator. Thus every path between N_1 and N_2 crosses the separating path.

When there is no single path of the desired length that divides N into roughly equal subsets we can use two paths as shown in Figure 7. Each of the paths has length at most $4d$, and their endpoints a_1, b_1 and a_2, b_2 satisfy the following properties: Node a_1 is in A_1 while node a_2 is in A_3; b_1 and b_2 occur after a_1 and a_2 respectively in the cyclic order around f; and the separation between a_i and b_i, $i = 1, 2$ is the maximum possible (in terms of their placement on the boundary of f) under the previous constraints. As shown in Figure 7, the exterior of face f is topologically separated into three

```
procedure sparsify
[Initializing the global substitute sub ]
    sub ← φ;
    [Grouping the paths]
        for each value of i in [0, log D] do
        [Group i sparsifies the paths in the range [2^i, 2^(i+1)]]
            let d ← 2^i;

    [ Sparsifying the paths in group i]
        1. Use separate to find a set Z of one or two O(d)-length paths such
           that Z divides N into two (roughly) equal sets N1 and N2,
           and cuts all group i paths between N1 and N2;
        2. Recursively construct substitutes R1 and R2 for the group i paths
           that have both their endpoints in N1 and N2 respectively;
        3. Use basic-sparse to construct a substitute R3 for the paths
           that have one endpoint in each of N1 and N2;
        4. Union R1, R2, and R3 to get sub_i, the substitute for group i;

    [ Adding to the global substitute]
        5. sub ← sub ∪ sub_i;
    end [ for ]
end [ procedure sparsify]
```

Figure 8: Procedure **sparsify**

regions R_1, R_2, and R_3. Any path between N_1 and N_2 that has one of its endpoints in R_1 or R_2 is forced to cross one of the separating paths. Also, by the maximality of the separation between a_i and b_i there is no path from N_1 to N_2 of length at most $4d$ that lies entirely in R_3.

These paths can be easily found by performing $O(\log k)$ single-source shortest-path computations from the nodes on A_1, A_2, A_3, and A_4. Therefore, using the shortest-path algorithm of [4] we can find the separating paths in $O(n \log k)$ time.

To bound the size of the substitute we note that the substitute for shortest paths in any range $[d, 2d]$ is constructed in $O(\log k)$ stages. Furthermore, at each level in the recursion the sum of the sizes of all crossing substitutes is

$O(\epsilon^{-1}k)$. Hence the total size of the substitute for group d is $O(\epsilon^{-1}k \log k)$. This combined with the fact there are $O(\log D)$ groups implies that the size of the substitute is $O(\epsilon^{-1}k \log k \log D)$.

To bound the running time of our sparsification procedure we note that the time required for constructing the crossing substitute in step 2 is $O(\epsilon^{-1}n)$ and the time required to find the separating paths is $O(n \log k)$. This in conjunction with the fact that the recursion depth is $O(\log k)$ and the fact that there are $\log D$ groups implies the total time for the procedure is $O(\epsilon^{-1}n \log^2 k \log D)$.

For the case when the selected nodes are on a constant number of faces $f_1, f_2, \cdots, f_r$ to find a substitute for paths in the range $[d, 2d]$ we proceed as follows:

To find a substitute for paths that go between faces f_i and f_j (for $1 \leq i, j \leq r$) we first find a path of length at most $2d$ between f_i and f_j. If no such path exists then there are no paths in the range $[d, 2d]$ between f_i and f_j. Otherwise let π be one such path. We first find a substitute S_π for all the paths in the range $[d, 2d]$ that cross π by using the basic sparsification technique from Section 4.1. Let $x \in f_i$ and $y \in f_j$ be the two end points of π. For the purposes of the paths in the range $[d, 2d]$ that don't cross π we can conceptually think of the selected nodes of f_i and f_j to be located on a composite face f_{ij} constructed as follows:

Duplicate all the nodes on the path π creating two paths π' and π'' one below the other with endpoints x', y' and x'', y'' respectively. The face f_{ij} is the single face formed in this fashion. See Figure 9. The substitute for all the paths that don't intersect π can now be computed in the same manner as before by assuming f_{ij} as the conceptual single face containing the selected nodes. Since there are only a constant number of such faces we get the bounds of Theorem 2.

5 Extensions

A similar technique has proved useful in solving the dynamic reachability problem in planar digraphs [15]. These techniques can also be used to develop a dynamic data structure for maintaining shortest paths in planar directed graphs. However, for that data structure we need some additional techniques[15, 16]. Details can be found in [17].

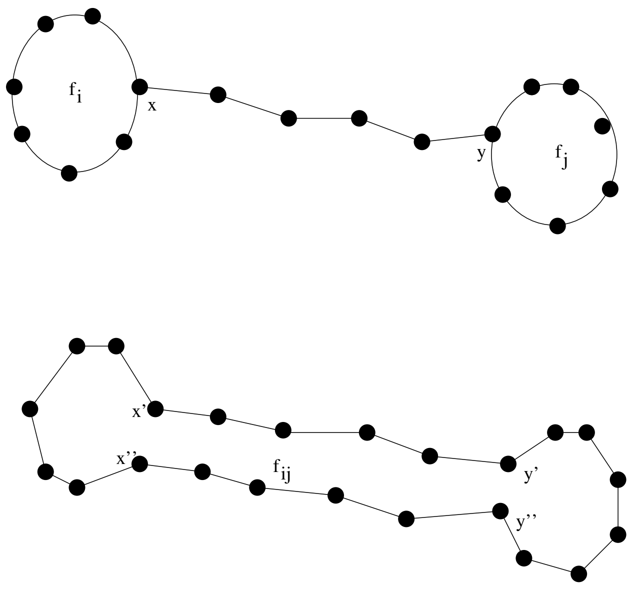

Figure 9: Creating the composite face f_{ij}

6 Acknowledgements

We are grateful to the referees for their comments and for pointing out an error in an earlier version of the document.

References

[1] E.W Dijkistra, " A Note on two Problems in Connexion with Graphs," *Numerische Mathematik* 1 (1959), 269-271.

[2] M. L. Fredman and R. E. Tarjan, "Fibonacci Heaps and their Uses in Improved Network Optimization Algorithms," *Journal of the Association for Computing Machinary*34 (1987) 596-615.

[3] G. N. Frederickson, " Fast Algorithms for Shortest Paths in Planar Graphs," *SIAM Journal on Computing* 16 (1987), 1004-1022.

[4] P. N. Klein, S. Rao, M. Rauch, and S. Subramanian, "Faster Shortest-path Algorithms for Planar Graphs," *Proceedings of the 26th annual ACM STOC* (1994) 27-37.

[5] S. Even and H. Gazit, "Updating Distances in Dynamic Graphs," *Methods of Operations Research* 49 (1985), 371-387

[6] G. Ausiello, G. F. Italiano, A. Marchetti-Spaccamela, and U. Nanni, "Incremental Algorithms for Minimal Length Paths," *Proceedings of annual ACM SIAM Symposium on Discrete Algorithms* (1990), 12-21.

[7] E. Fuerstine and A. Marchetti-Spaccamela, "Dynamic Algorithms for Shortest-path Problems in Planar Graphs," *Proceedings of the 17th International Workshop on Graph Theoretic Concepts in Computer Science* (1991), 187-197.

[8] H.N. Dijdev, G. E. Pantziou, and C. D. Zaroliagis, "Computing Shortest-paths and Distances in Planar Graphs," *Proceedings of the 18th International Colloqium on Automata, Languages, and Programming* (1991), 327-338.

[9] G. N. Frederickson, "Data Structures for On-line Updating of Minimum Spanning Trees with Applications," *SIAM Journal on Computing* 14 (1985), 781-798.

[10] Z. Galil and G. F. Italiano, "Maintaining Biconnected Components of Dynamic Planar Graphs," *Proceedings of the 18th International Colloqium on Automata, Languages, and Programming* (1991), 339-350.

[11] Z. Galil, G. F. Italiano, and N. Sarnak, "Fully Dynamic Planarity Testing," *Proceedings of the 24th annual ACM STOC* (1992) 495-506.

[12] R. J. Lipton and R. E. Tarjan, "A Separator Theorem for Planar Graphs," *SIAM Journal of Applied Mathematics* 36 (1979), 177-189.

[13] G. L. Miller, "Finding Small Simple Cycle Separators for Two-connected Planar Graphs," *Journal of Computer and System Sciences* 32 (1986), 265-279.

[14] P. N. Klein and S. Sairam, "A Parallel Randomized Approximation Scheme for Shortest Paths," *Proceedings of the 24th annual ACM STOC* (1992), 750-758.

[15] S. Subramanian, "A Fully Dynamic Data Structure for Reachability in Planar Digraphs," *Proceedings of the European Symposium on Algorithms* (1993).

[16] P.N. Klein and S. Subramanian, "A Linear-processor Polylog-time Algorithm for Shortest-paths in Planar Graphs," *Proceedings of the annual IEEE FOCS* (1993), 259-270.

[17] S. Subramanian, "Parallel and Dynamic Graph Algorithms: A Combined Perspective," *Ph.d Thesis, Department of Computer Science*, (1994).

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