Efficient Algorithms for Shortest Path Queries in Planar Digraphs *

Hristo N. Djidjev

Department of Computer Science, Rice University
P.O. Box 1892, Houston, TX 77251, USA
email: hristo@cs.rice.edu

Abstract. This paper describes algorithms for answering shortest path queries in digraphs with small separators and, in particular, in planar digraphs. In this version of the problem, one has to preprocess the input graph so that, given an arbitrary pair of query vertices v and w, the shortest-path distance between v and w can be computed in a short time. The goal is to achieve balance between the preprocessing time and space and the time for answering a distance query. Previously, efficient algorithms for that problem were known only for the class of outerplanar digraphs and for the class of digraphs of constant treewidth. We describe efficient algorithms for this problem for any class of digraphs for which an $O(\sqrt{n})$ separator theorem holds. For such graphs our algorithm uses $O(S)$ space and answers queries in $O(n^2/S)$ time, for any previously chosen $S \in [n, n^2]$. For the class of planar digraphs improved algorithms are described.

1 Introduction

Let G be a digraph with weights on the edges and let p be a path in G with endpoints v and w. The *length* of p is the sum of the weights of all edges of p. The *shortest-path distance* between v and w is the minimum length of a path from v to w. The path of minimum length is called a *shortest path* between v and w.

Finding shortest path information in a graph is a very important and intensively studied problem with applications in communication systems, transportation, scheduling, computation of network flows, etc. [1]. For the single-source shortest path (SSSP) problem with non-negative edge weights, Dijkstra's algorithm takes $O(n^2)$ time [5]. An implementation of Dijkstra's algorithm that uses Fibonacci heaps [10] reduces the time to $O(n \log n + m)$.

Many recent papers address shortest path problems for special classes of graphs. Johnson proposes an $O(nm + n^2 \log n)$-time algorithm for the all-pairs shortest path (APSP) problem for sparse graphs [12]. Frederickson and Janardan construct an $O(n)$ algorithm for the SSSP problem for outerplanar graphs [8]. Using planar separator theorems, Frederickson develops an $O(n\sqrt{\log n})$-time

* This work was partially supported by the NSF grant No. CCR-9409191.

algorithm for the single-source and an $O(n^2)$-time algorithm for the all-pairs shortest path problem for planar digraphs [9]. The time for the single-source problem for planar digraphs was later improved to $O(n)$ in [13]. More generally, given a class of graphs satisfying an $O(n^{1-\delta})$-separator theorem where $0 < \delta < 1$, the SSSP problem can be solved in $O(n)$ time for any n-vertex graph of the class, not including the time to find a recursive separator decomposition [13].

In this paper we consider the so-called on-line version of the shortest path problem, where one has to construct in a preprocessing phase a data structure such that, given an arbitrary pair of query vertices v and w, the shortest-path distance between v and w can be found fast. This version of the shortest path problem arises in applications where shortest-path distances have to be computed only between a small and previously unknown set of vertex pairs. Such applications include facility allocation, transmission in communication systems, circuit and VLSI design. An efficient solution to the on-line version of the shortest path problem is given in [6] for the class of outerplanar digraphs, where the preprocessing algorithms uses $O(n)$ time and space and a distance query is answered in $O(\log n)$ time. This result implies an algorithm for the class of planar digraphs with face-on-vertex cover q with preprocessing time and space $O(n \log n + q^2)$ and query time $O(\log n)$. That algorithm is, however, not efficient for arbitrary planar digraphs, since in the worst case $q = \Theta(n)$ and the preprocessing space is $\Theta(n^2)$, the same as in the APSP algorithm [9] for planar digraphs. An algorithm with $O(\alpha(n))$ query time was described for the class of digraphs of constant treewidth in [3, 4].

We study here the on-line version of the shortest path problem for the class of all planar digraphs and, more generally, for any class of digraphs satisfying an $O(\sqrt{n})$ separator theorem. Our goal is to construct algorithms for which the preprocessing-space $\times$ query-time product is not greater compared to the best algorithms for the SSSP and APSP problems. Specifically, for digraphs with $O(\sqrt{n})$ separators, if S denotes the preprocessing-space and Q denotes the query time, then for the SSSP problem $S = O(n)$ and $Q = O(n)$ and for the APSP problem $S = O(n^2)$ and $Q = O(1)$. In both cases $SQ = O(n^2)$. We will show in this paper that for the same class of graphs we can construct algorithms whose preprocessing spaces and query times yield the same $SQ = O(n^2)$ tradeoff, for S in the range $[\Theta(n), \Theta(n^2)]$. Surprisingly, if the input graph is planar, we can use the topology of planar graphs in order to achieve even better space-query time tradeoff (but for a smaller range for S). We construct an algorithm for planar digraphs with continuous space-query time tradeoff $SQ = O(n\sqrt{S} \log n)$ for $S \in [\Theta(n^{4/3}), \Theta(n^{3/2}))$. This algorithm is most efficient for $S = \Theta(n^{4/3})$, for which value $SQ = O(n^{5/3} \log n)$.

This paper is organized as follows. In Section 2, we describe a simple divide-and-conquer algorithm for digraphs with $O(\sqrt{n})$ separators which works for $S = \Theta(n^{3/2})$. That algorithm will be used in the next two sections, where algorithms for $S > n^{3/2}$ and $S < n^{3/2}$ will be described, respectively, for the same class of graphs. More efficient algorithms for the class of planar digraphs are described in Section 5.

2 The basic algorithm

A class of graphs satisfies an $f(n)$ *separator theorem*, if for any n-vertex graph G from the class there exists a set of $O(f(n))$ vertices (called a *separator*) whose removal leaves no component of size exceeding αn for some previously fixed $\alpha < 1$. In the next three sections, by $\mathcal{G}_{\sqrt{n}}$ we will denote a class of graphs for which an $O(\sqrt{n})$ separator theorem holds and G will denote a digraph from $\mathcal{G}_{\sqrt{n}}$ with non-negative edge weights. It is assumed that a recursive separator decomposition can be found in $O(n)$ time (so that the linear time SSSP algorithm from [13] can be used). Given a set C of vertices of G, let $G - C$ denote the subgraph of G induced by $V(G) \setminus C$.

The preprocessing algorithm

Algorithm Preprocess–basic

1. Construct a balanced decomposition tree T_G of G as follows.
   (a) If G contains a single vertex, then return a tree T_G of a single vertex labeled by $V(G)$.
   (b) Construct a set C of $O(\sqrt{n})$ vertices whose removal divides G into components with no more than αn vertices each, where $\alpha < 1$.
   (c) Define a new vertex labeled by C. Find the connected components of $G - C$. For each component K of $G - C$ construct a decomposition tree T_K by running the algorithm recursively on K and make the root of T_K a child of C. Let T_G be the resulting tree rooted at C.
2. For each vertex (set) M of T_G compute and store the shortest-path distances from any vertex of M to all vertices belonging to a descendant of M in T_G (including M).

Note that in Step 2 we compute the distances between any vertex of C and any vertex of G, for any component K of $G - C$ we compute the distances between any vertex of the separator of K and any vertex of K, etc.

Lemma 2.1 *Algorithm* Preprocess–basic *runs in $O(n^{3/2})$ time and uses $O(n^{3/2})$ space.*

Proof: For all steps of the algorithm the time and the space requirements are the same. Hence we shall analyze only the time complexity. Constructing the separator and finding the connected components in Step 1 requires $O(n)$ time for one iteration. Since T_G is balanced, the total time needed for Step 1 is $O(n \log n)$.

Let us now analyze Step 2. Finding the shortest-path distances from a vertex of C to all vertices of G takes $O(n)$ time using the linear SSSP algorithm from [13]. Thus, computing all distances associated with the vertices of C requires $|C|O(n) = O(n^{3/2})$ time.

Similarly, for any component K of $G - C$ (i.e. for any child K of C in T_G) and any vertex v of the separator of K, we would like to compute the distances (in G) between v and all vertices of K in $O(|K|)$ total time. We can not just run a SSSP algorithm in K and ignore the rest of G, since a shortest path may use vertices which are not in K. For this end we modify K by adding edges joining v to any vertex of C with weight equal to the length of the corresponding shortest path. Since K is a component of $G - C$, the shortest-path distance in G between v and any vertex of K is the same as the shortest-path distance between these vertices in the modified graph K'. Moreover, any separator of K corresponds to a separator of K' with at most one additional vertex (vertex v). Thus,the shortest path distances from v to all vertices in K can be computed using the SSSP algorithm from [13] in $O(|K'|) = O(|K|)$ time. For all vertices of the separator C_K of K the time to compute all shortest-path distances is $O(|K||C_K|) = O(|K|^{3/2})$.

Thus, if $T(n)$ is the maximum time needed for Step 2 for any n-vertex digraph from $\mathcal{G}_{\sqrt{n}}$, then $T(n)$ satisfies the recurrence

$$T(n) \leq \max\{\sum_i T(n_i) + O(n^{3/2}) \mid n_i \leq \alpha n, \sum_i n_i = n\},$$

where $\alpha < 1$, whose solution is $T(n) = O(n^{3/2})$. □

The query algorithm

Algorithm Query–basic
$\{$Finds the shortest-path distance between vertices v and $w\}$

1. Find vertices (sets) M_v and M_w of T_G such that $v \in M_v$ and $w \in M_w$.
2. Find the nearest common ancestor (NCA) M in T_G of M_v and M_w.
3. Compute the shortest-path distance between v and w using the formula

$$distance(v, w) := \min\{dist(v, m) + dist(m, w) \mid m \in M\},$$

where distances *dist* have been computed during the preprocessing phase.

Lemma 2.2 *Algorithm* Query–basic *computes the shortest-path distance between v and w in $O(\sqrt{n})$ time.*

Proof: *Correctness:* Let U be the union of the sets of vertices corresponding to M and all its descendants in T_G and let H be the subgraph of G induced by U. By the construction of T_G, M is the separator of H found in Step 1(b) of Algorithm Preprocess–basic and, since M_v and M_w are descendants of M, v and w both belong to H. Thus Step 3 of Algorithm Query–basic correctly computes the shortest-path distance between v and w, if v and w belong to different components of $H - M$.

Assume for the sake of contradiction that v and w belong to some component H' of $H - M$. The subtree $T_{H'}$ of T_G corresponding to H' contains both v and w and is rooted at a proper descendant of M. Then the nearest common ancestor of M_v and M_w is a proper descendant of M which contradicts the choice of M.

Time analysis: Searching T_G and finding the NCA of M_v and M_w in Step 1 takes $O(\log n)$ time since T_G is balanced. Step 3 requires finding the minimum of $|M|$ numbers and thus takes $O(|M|) = O(\sqrt{n})$ time. □

We summarize the above results in the following theorem.

Theorem 1 *Given an n-vertex digraph G from $\mathcal{G}_{\sqrt{n}}$, a data structure of size $O(n^{3/2})$ can be constructed in $O(n^{3/2})$ time so that, given any two vertices v and w, the shortest-path distance from v to w can be computed in $O(\sqrt{n})$ time.*

Theorem 1 shows that we can reduce the space and the preprocessing time for the shortest path problem by a factor of $\Theta(\sqrt{n})$, which will lead to an increase of the query time by a factor of $\Theta(\sqrt{n})$ (from $\Theta(1)$ to $\Theta(\sqrt{n})$). (A similar result is independently obtained in [2].) A natural question arises whether it is possible to construct an algorithm that allows faster queries, possibly at the expense of more preprocessing time or space, or alternatively, algorithms that are more efficient in terms of preprocessing resources, but slower in answering queries. In the next sections we will show that the answer to both questions is positive.

3 Algorithms with improved query times

By repeated application of a separator theorem, one can divide a graph G from $\mathcal{G}_{\sqrt{n}}$ into multiple components of roughly the same size. In addition to the requirement that the separator must be small in size, we will also require that each component into which G is divided is adjacent to a roughly the same (small) number of separator vertices. In order to formally specify this requirement we need some definitions. The vertices of G will be divided into sets called *regions*, and vertices will be classified as either *internal* or *boundary* vertices. Internal vertices belong to only one region and boundary vertices belong to more than one region. Moreover, any pair of adjacent internal vertices belong to the same region.

For any internal vertex v, by $R(v)$ we denote the region containing v, by $C(v)$ we denote the component of the graph induced by $R(v)$ that contains v, and by $B(v)$ we denote the set of boundary vertices of $R(v)$. $B(v)$ will be called the *boundary* of $R(v)$. If v is a boundary vertex we define $R(v) = C(v) = B(v) = \{v\}$.

We will use the following result of Frederickson [9].

Theorem 2 *For any n-vertex graph G from $\mathcal{G}_{\sqrt{n}}$ and any $\varepsilon \in (0,1)$ the vertices of G can be divided into $O(1/\varepsilon)$ regions with $O(\varepsilon n)$ internal vertices and $O(\sqrt{\varepsilon n})$ boundary vertices each. Moreover, each boundary vertex is contained in no more than 3 regions. Such a division (called an ε-*division*) can be found in $O(n \log n)$ time and $O(n)$ space.*

The preprocessing algorithm

The algorithms described in the following sections compute in a preprocessing step shortest path distances between the boundary vertices of some ε-division and the other vertices of the graph. We illustrate that storing an appropriate subset of that information and choosing a suitable value of ε can result in various shortest path algorithms with improved performance. The following algorithm gives the typical structure of such a preprocessing algorithm with details to be given in the concrete implementation.

Algorithm Preprocess–generic

1. Let $\varepsilon \in (0,1)$. Construct an ε-division of the input graph G as in Theorem 2.
2. Compute the connected components of the subgraph of G induced by the set of internal vertices of the division. Keep a pointer from each internal vertex u of G to the component $C(u)$ and to the region $R(u)$ containing u.
3. For any boundary vertex b compute the shortest-path distances from b to each vertex of G and from each vertex of G to b.
4. **Store a subset** (to be specified in the concrete implementation) of the shortest path information computed in the previous step.
5. Do **additional preprocessing** (to be specified in the concrete implementation), if necessary.

Lemma 3.1 *Steps 1–3 of Algorithm* Preprocess–generic *require* $O(n^{3/2}\varepsilon^{-1/2})$ *time and* $O(n)$ *space.*

Proof: Step 1 requires $O(n \log n)$ time and $O(n)$ space according to Theorem 2. Step 2 takes $O(n)$ time and space. The total number of boundary vertices according to Theorem 2 is $O(\sqrt{n/\varepsilon})$. Thus Step 3 takes $O(\sqrt{n/\varepsilon}\, n) = O(n^{3/2}\varepsilon^{-1/2})$ time. The space for any iteration of Step 3 is $O(n)$ and that space can be reused; thus the total space for that step is $O(n)$. □

Next we give the main preprocessing and query algorithms of this section.

Algorithm Preprocess–I

Steps 1–3 are the same as in Algorithm Preprocess–generic.

4. Store: the shortest-path distances from (boundary vertex) b to each vertex of G and from each vertex of G to b.

5. Additional preprocessing: apply Algorithm Preprocess–basic on any component of internal vertices found in Step 2.

Lemma 3.2 *Algorithm* Preprocess–I *runs in* $O(n^{3/2}\varepsilon^{-1/2})$ *time and uses* $O(n^{3/2}\varepsilon^{-1/2})$ *space.*

Proof: The total number of boundary vertices according to Theorem 2 is $O(\sqrt{n/\varepsilon})$. Thus Step 4 takes $O(\sqrt{n/\varepsilon}\, n) = O(n^{3/2}\varepsilon^{-1/2})$ time and uses $O(n^{3/2}\varepsilon^{-1/2})$ space. By Lemma 2.1 the time and space for Step 5 is $O((n\varepsilon)^{3/2})O(1/\varepsilon) = O(n^{3/2}\varepsilon^{1/2}) = O(n^{3/2}\varepsilon^{-1/2})$ (since $\varepsilon < 1$). □

The query algorithm

Algorithm Query–I
{Finds the shortest-path distance between vertices v and w}

1. If v or w is a boundary vertex, then find the shortest-path distance between v and w by using the information computed in Step 3 of Algorithm Preprocess–I and halt.
2. Determine the components $C(v)$ and $C(w)$ and the boundary $B(v)$.
3. If $C(v) \neq C(w)$, then compute the shortest-path distance between v and w by the formula

$$distance(v, w) := \min\{dist(v, b) + dist(b, w) \mid b \in B(v)\},$$

   where *dist* are the shortest-path distances computed by the preprocessing algorithm.
4. If $C(v) = C(w)$, then compute the shortest-path distance between v and w by applying Algorithm Query–basic on $C(v)$ with a query pair v, w.

Lemma 3.3 *Algorithm* Query–I *computes the shortest-path distance between v and w in $O(\sqrt{\varepsilon n})$ time.*

Proof: *Correctness:* If $C(v) \neq C(w)$, then any path from v to w must contain a boundary vertex from $R(v)$. Thus Step 3 correctly computes the shortest-path distance between v and w in that case. If $C(v) = C(w)$, then the correctness of Algorithm Query–I follows from the correctness of Algorithm Query–basic (Lemma 2.2).

Time analysis: Steps 1 and 2 take $O(1)$ time. Step 3 takes time proportional to the maximum size of a boundary of a region, which is $O(\sqrt{\varepsilon n})$. Finally, Step 4 requires $O(\sqrt{|C(v)|}) = O(\sqrt{|R(v)|}) = O(\sqrt{\varepsilon n})$ time. □

We have the following theorem that summarizes the results from this section.

Theorem 3 *Given an n-vertex digraph G from $\mathcal{G}_{\sqrt{n}}$ and any number $S \in [n^{3/2}, n^2]$, a data structure of size $O(S)$ can be constructed in $O(S)$ time so that, for any two vertices v and w, the shortest-path distance from v to w can be computed in $O(n^2/S)$ time.*

Proof: Follows from Lemma 3.2 and Lemma 3.3 by setting $\varepsilon = n^3/S^2$. □

Theorem 3 yields a continuous trade-off $SQ = \Theta(n^2)$ between the area S and the query time Q, for S in the range $[n^{3/2}, n^2]$. For $S = n^2$, Theorem 3 gives the same time as the optimal algorithm for the all pairs shortest paths problem for digraphs with $O(\sqrt{n})$ separator [9], which uses $O(n^2)$ space and answers queries in $O(1)$ time.

Next, we will extend the above result by describing an algorithm achieving the same space–query time tradeoff $SQ = \Theta(n^2)$ as in Theorem 3, but for S in the range $[n, n^{3/2}]$.

4 Space–efficient algorithms

The preprocessing algorithm

The algorithm is similar to Algorithm Preprocess–I except that, for any boundary vertex b, we store only the shortest-path distances between b and the other boundary vertices. (In Algorithm Preprocess–I we stored the shortest-path distances between any boundary vertex b and all vertices of G.) A similar preprocessing for a dynamic version of the shortest path problem was used in [7].

Algorithm Preprocess–II

Steps 1–3 are the same as in Algorithm Preprocess–generic.

4. *Store:* the shortest-path distances between (boundary vertex) b and all boundary vertices of G (in both directions).
5. *Additional preprocessing:* None.

Lemma 4.1 *Algorithm* Preprocess–II *runs in* $O(n^{3/2}\varepsilon^{-1/2})$ *time and uses* $O(n/\varepsilon)$ *space.*

Proof: The time bound follows from Lemma 3.1. The space needed to store the shortest-path distances from any boundary vertex b is $O((n/\varepsilon)^{1/2})$. The space bound follows, since the number of all boundary vertices is $O((n/\varepsilon)^{1/2})$. □

The query algorithm

Algorithm Query–II
{Finds the shortest-path distance between vertices v and w}

1. If both v and w are boundary vertices, then find the shortest-path distance between v and w by using the distance table computed in Step 3 of Algorithm Preprocess–II and stop.
2. If $C(v) \neq C(w)$, then
   (a) compute the shortest-path distances from v to all vertices of $B(v)$ by using the linear time SSSP algorithm applied on graph $R(v)$;
   (b) compute the shortest-path distances from all vertices of $B(w)$ to w by using the SSSP algorithm applied on graph $R(w)$;
   (c) compute the shortest-path distance D from v to w by the formula

$$D := \min\{d_{query}(v,b_1)+dist(b_1,b_2)+d_{query}(b_2,w) \mid b_1 \in B(v), b_2 \in B(w)\},$$

   where $dist$ denotes the distance computed by the preprocessing algorithm, and d_{query} denotes the distance computed in Steps 2 (a) and (b) of this algorithm.

3. If $C(v) = C(w)$, then compute the shortest-path distance between v and w by adding in $R(v)$ edges between v and all vertices of $B(v)$ with weights equal to the lengths of the corresponding shortest paths and applying the SSSP algorithm with source v on the resulting graph.

Lemma 4.2 *Algorithm* Query–II *computes the shortest-path distance between v and w in $O(\varepsilon n)$ time.*

Proof: The correctness of the algorithm follows from the definition of regions and boundary vertices. Step 1 requires $O(1)$ time. The time for Steps 2(a)and (b) is $O(|R(v)| + |R(w)|) = O(\varepsilon n)$. Step 2(c) needs $O(|B(v)|\,|B(w)|) = O(\sqrt{\varepsilon n}\sqrt{\varepsilon n}) = O(\varepsilon n)$ time. Step 3 requires $O(|R(v)| + |B(v)|) = O(\varepsilon n)$ time. □

We combine Lemma 4.1 and Lemma 4.2 the following theorem.

Theorem 4 *Given an n-vertex digraph G from $\mathcal{G}_{\sqrt{n}}$ and any $S \in [n, n^{3/2}]$, a data structure of size $O(S)$ can be constructed in $O(nS^{1/2})$ time so that, for any two vertices v and w, the shortest-path distance from v to w can be computed in $O(n^2/S)$ time.*

Proof: Follows from Lemma 4.1 and Lemma 4.2 by setting $\varepsilon = n/S$. □

Note that the space and the query time from Theorem 4 satisfy the same relationship $SQ = O(n^2)$ as the best known algorithm for the SSSP problem [13] (where $S = O(n)$ and $Q = O(n)$).

In the next section we will show that improvements to Theorem 3 and Theorem 4 are possible in the special case of planar input graphs.

5 Faster algorithms for the class of planar digraphs

The results from the previous sections apply also to planar graphs, since the class of planar graphs has an $O(\sqrt{n})$-separator theorem and recursive separator decomposition can be found in $O(n)$ time [11]. In order to speed-up our algorithms, we will additionally use some topological properties of planar graphs. In particular, we will make use of the Jordan Curve Theorem, which states that any simple closed curve in the plane divides it into exactly two connected regions “inside” and “outside”.

The preprocessing algorithm

In the preprocessing algorithm we will compute, as before, the shortest-path distances between selected pairs of vertices. In this algorithm, however, we will not store information (will ignore) paths that cross the boundaries of some regions multiple times. For this end, we will modify the shortest path trees rooted at the boundary vertices in the following manner.

Let v be any vertex of G. In our algorithms bellow we will need in some cases shortest paths that avoid vertices of $B(v)$. For given $b \in B(v)$, we call a *reduced shortest path tree* from b with respect to $B(v)$ the shortest path tree in

$G - \{B(v) \setminus \{b\}\}$ rooted at b. Note that since there are at most three regions adjacent to any boundary vertex b, we need to compute and store no more than three reduces shortest path trees from b.

Algorithm Preprocess–planar

Steps 1–3 are the same as in Algorithm Preprocess–generic.

4. Store: the shortest-path distances between (boundary vertex) b and all boundary vertices of G (in both directions).

5. Additional preprocessing:

(a) Compute the reduced shortest path trees from any boundary vertex x with respect to the boundaries of all regions adjacent to x and store the corresponding shortest-path distances between x and all boundary vertices of G (in both directions).

(b) Apply Algorithm Preprocess–basic on any component found in Step 2.

Lemma 5.1 *Algorithm* Preprocess–planar *runs in* $O(n^{3/2}\varepsilon^{-1/2})$ *time and uses* $O(n/\varepsilon + n^{3/2}\varepsilon^{1/2})$ *space.*

Proof: The time and space for Steps 1–4 are the same as in Algorithm Preprocess–II, i.e., $O(n^{3/2}\varepsilon^{-1/2})$ time and $O(n/\varepsilon)$ space. The time and space for Step 5 (b) are $O((n\varepsilon)^{3/2})$ per region, according to Lemma 2.1. Hence, the total time and the total space for Step 5 are $O((n\varepsilon)^{3/2})O(1/\varepsilon) = O(n^{3/2}\varepsilon^{1/2})$. □

The query algorithm

The idea of the query algorithm is the following. We have already computed during the preprocessing phase the shortest-path distances between any two boundary vertices of G. Using a similar type of information, in Algorithm Query–II we found the distance between query vertices v and w by computing the length of any shortest path from v to w containing intermediate vertices $x \in B(v)$ and $y \in B(w)$, for all possible choices of x and y. Thus the query time was proportional to $|B(v)||B(w)|$. If we use the planarity of G, we will be able to reduce the number of considered pairs (x, y) to $|B(w)| \log |B(v)|$. For example, assume that c' and c'' are cycles of vertices of $B(v)$ and $B(w)$, and the paths p_1 and p_2 illustrated on Figure 3 are shortest paths from x_1 and x_2 to w containing vertices y_1 and y_2, respectively. Then, if x is a vertex on the clockwise portion of c' from x_1 to x_2, a shortest path will exist from x to w that contains a vertex y from the clockwise portion of c'' from y_1 to y_2. Thus, we can limit our search to only a portion of c''.

We assume w.l.o.g. that G is a maximal planar digraph. (If G is not maximal, we can add new edges with infinite lengths.) In a maximal planar graph, each face is a triangle and the union of any number of triangles results in a region whose boundary is a union of edge-disjoint cycles. Thus, the boundary of any region of the ε-partition of G is a union of simple edge-disjoint cycles.

By $dist(x, y)$ we denote the distance between vertices x and y computed in Step 5 of Algorithm Preprocess–planar. Assume that v and w are 2 vertices

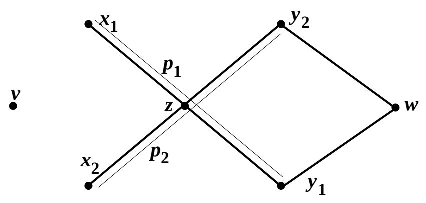

Fig. 1. Intersecting paths p_1 and p_2.

such that $C(v) \neq C(w)$. Moreover, let $x_1 \neq x_2$ be 2 vertices from $B(v)$ and y_1 and y_2 be 2 vertices from $B(w)$ such that, for $i = 1, 2$, the sum

$$dist(x_i, b_i) + dist(b_i, w) \text{ for } b_i \in B(w)$$

is minimized for $b_i = y_i$. Then $dist(x_i, y_i) + dist(y_i, w)$ will be the shortest-path distance from x_i to w. Let p_1 and p_2 be shortest paths from x_1 to y_1 and from x_2 to y_2, respectively. We will show that such paths p_1 and p_2 can be chosen that do not "cross", i.e., have the structure illustrated on Figure 2.

Lemma 5.2 *If $dist(x_2, y_2) + dist(y_2, w) < dist(x_2, y_1) + dist(y_1, w)$, then p_1 and p_2 are vertex disjoint.*

Proof: Assume that p_1 and p_2 have a common vertex, say z (Figure 1). Since both p_1 and p_2 are shortest paths, then $dist(z, y_2) + dist(y_2, w) = dist(z, y_1) + dist(y_1, w)$. Hence $dist(x_2, y_2) + dist(y_2, w) = dist(x_2, z) + dist(z, y_1) + dist(y_1, w) \geq dist(x_2, y_1) + dist(y_1, w)$ – a contradiction. Thus p_1 and p_2 are vertex disjoint. □

If the condition of Lemma 5.2 is not satisfied, i.e., if $dist(x_2, y_2) + dist(y_2, w) \geq dist(x_2, y_1) + dist(y_1, w)$, then we can replace p_2 by another shortest path by choosing $y_2 := y_1$. We can, therefore, make the assumption that in this case p_1 and p_2 consist of two vertex-disjoint paths p_1' and p_2', respectively, followed by a common subpath with an endvertex on $B(w)$ (Figure 2 (a)). If the conditions of Lemma 5.2 are satisfied, then p_1 and p_2 are vertex disjoint (Figure 2 (b)).

Suppose that c' and c'' are the two cycles from the boundaries of v and w, respectively, that separate v and w (Figure 3). Then, any path from v to w will contain at least one vertex from c' and at least one vertex from c''. Let x_1 and

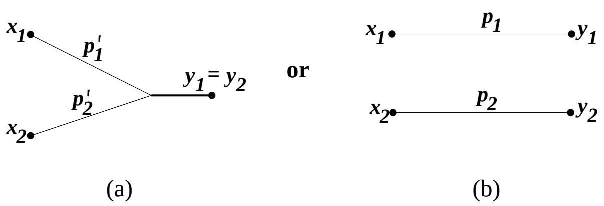

Fig. 2. Possible structure of the paths p_1 and p_2.

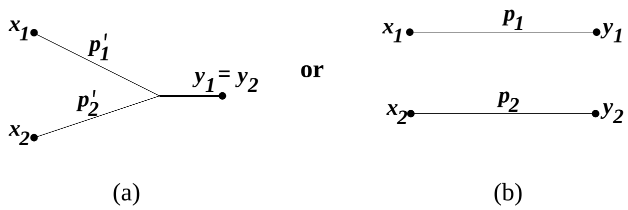

Fig. 3. Separating cycles of boundary vertices.

x_2 be chosen from c' and y_1 and y_2 be chosen from c'', where x_1, x_2, y_1, and y_2 are as defined for Lemma 5.2.

Denote, for any cycle c and two vertices x and y on c, by $Cchain(x, y, c)$ the subpath of c between vertices x and y in clockwise direction. Similarly, denote by $CCchain(x, y, c)$ the subpath of c between vertices x and y in counterclockwise direction. We will write simply $Cchain(x, y)$ and $CCchain(x, y)$ if the cycle c is clear from the context.

Lemma 5.3 *If x is a vertex from $Cchain(x_1, x_2)$ such that there is a shortest path between x and w containing no vertices from $B(v)$, then the sum $dist(x, b) + dist(b, w)$ for $b \in c''$ will be minimized for some y from $CCchain(y_1, y_2)$ (Figure 3).*

Proof: Let p_1 and p_2 be shortest paths from x_1 and x_2 to y_1 and y_2 that have no other vertices from $B(v)$ except for their start vertices.

First, consider the case when p_1 and p_2 are vertex disjoint. Define a cycle C consisting of the following paths: a shortest path from v to x_1, p_1, a shortest path from y_1 to w, a shortest path from w to y_2, p_2, and a shortest path from x_2 to v (Figure 4). Call the *inside* of C the region that contains a vertex from the clockwise chain on c' from x_1 to x_2.

Let p be a shortest path from x to w. We have the following two cases (Figure 3).

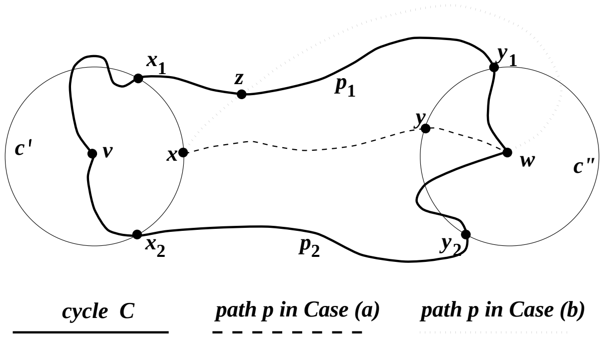

Fig. 4. Illustration to the proof of Lemma 5.3.

(a) p has no other vertex in common with $p_1 \cup p_2$ except w. Then p will contain a vertex from $CCchain(y_1, y_2)$. Hence p will contain a vertex from $CCchain(y_1, y_2)$.

(b) p contains a vertex $z \neq w$ from $p_1 \cup p_2$. Assume that $z \in p_1$. Then the path consisting from the subpath of p from x to z plus the subpath of p_1 from z to w is a shortest path from x to w which includes y_1. The claim follows.

Now consider the case when p_1 and p_2 have a common vertex. Then, by the assumption about the structure of p_1 and p_2, $y_1 = y_2$ (see Lemma 5.2 and Figure 2) and any path from x to w that contains no other vertices from $B(v)$ except x will intersect p_1 or p_2. Hence, a shortest path between x and w that contains y_1 can be constructed as in Case (b) above. □

The following query algorithm is based on Lemma 5.3.

Algorithm QUERY–PLANAR
{Finds the shortest-path distance between vertices v and w}

A. **If** $C(v) \neq C(w)$ **then do**

1. Modify the graph $C(v)$ by adding an edge between any pair of boundary vertices with weight equal to the shortest-path distance in G between the vertices as computed by the preprocessing algorithm. (Note that the number of additional edges is $O(|C(v)|)$.) Let $C'(v)$ denote the resulting (possibly non-planar) graph. Compute the shortest-path distances from v to all vertices from $B(v)$ by running Dijkstra's SSSP algorithm in $C'(v)$. Similarly, compute the shortest-path distances from all vertices from $B(w)$ to w.
2. Choose any two different vertices x_1 and x_2 from $B(v)$. Using the reduced shortest-path trees and Lemma 5.2, find a pair of vertices y_1 and y_2 such that there are shortest paths q_1 from x_1 to y_1 and q_2 from x_2 to y_2 with the properties (i) q_1 and q_2 are vertex disjoint or $y_1 = y_2$; and (ii) q_1 and q_2 contain no other vertices from $B(v)$ except x_1 and x_2.
3. For any vertex $x \in Cchain(x_1, x_2)$ find the minimum length of a path from x to w that does not intersect c' and contains a vertex from $CCchain(y_1, y_2)$ as follows.
   (a) Find a vertex x_3 on $Cchain(x_1, x_2)$ that divides $Cchain(x_1, x_2)$ into equal (within difference in size at most one) parts. Find a vertex $y_3 \in CCchain(y_1, y_2)$ such that the distance
   $$D(x_3) = dist(x_3, b) + d_{query}(b, w) \mid b \in CCchain(y_1, y_2)$$
   is minimized for $b = y_3$, where $dist$ denotes the distance computed by the preprocessing algorithm and d_{query} denotes the distance computed in Step 1 of this algorithm. Store $D(x_3)$ as the shortest-path distance from x_3 to w (avoiding vertices from c').
   (b) Find, recursively, for any vertex $x \in Cchain(x_1, x_3)$, the minimum length of a path from x to w that does not intersect c' and that contains a vertex from $CCchain(y_1, y_3)$.

(c) Find, recursively, for any vertex $x \in Cchain(x_3, x_2)$, the minimum length of a path from x to w that does not intersect c' and that contains a vertex from $CCchain(y_3, y_2)$.

4. Similarly, find the lengths of the shortest paths from all vertices from $CCchain(x_1, x_2)$ to w that do not intersect c' and contain a vertex from $Cchain(y_1, y_2)$.
5. Compute the shortest-path distance from v to w as

$$distance(v, w) := \min\{d_{query}(v, b) + d_{query}(b, w) \mid b \in B(v)\},$$

where distances $d_{query}(v, b)$ are computed in Step 1 and distances $d_{query}(b, w)$ are computed in Steps 3 and 4 of this algorithm.

endif

B. **If** $C(v) = C(w)$ **then** compute the shortest-path distance between v and w by applying Algorithm Query–basic on $C(v)$ with a query pair v, w.

Lemma 5.4 *Algorithm* Query–planar *computes the shortest-path distance between v and w in $O(\sqrt{\varepsilon n} \log(\varepsilon n))$ time.*

Proof (sketch): The correctness of the algorithm follows from the definition of reduced shortest path trees and Lemma 5.3. Dijkstra's algorithm runs in $O(k \log k)$ time on a graph with k vertices. Thus Step 1 takes $O(\varepsilon n \log(\varepsilon n))$ time, since $\max\{|C(v)|, |C(w)|\} = O(\varepsilon n)$ and the number of new edges added in Step 1 is $O(|B(v)|^2 + |B(w)|^2) = O((\sqrt{\varepsilon n})^2) = O(\varepsilon n)$. Step 2 takes $O(|B(w)|) = O(\sqrt{\varepsilon n})$ time. For the time analysis of Step 3, denote by k and m, respectively, the number of edges of $Cchain(x_1, x_2)$ and $CCchain(y_1, y_2)$. Finding the shortest-path distance between x_3 and $CCchain(y_1, y_2)$ in Step 3 (a) takes $O(m)$ time. Thus the time $T(k, m)$ for Step 3 satisfies the inequality

$$T(k, m) \leq \max\{T(\lfloor k/2 \rfloor, m_1) + T(\lceil k/2 \rceil, m_2) \mid m_1 + m_2 = m\},$$

where m_1 and m_2 correspond to the sizes of the parts into which $CCchain(y_1, y_2)$ is divided by y_3 (see Steps 2 (b) and 2 (c)). The solution to this recurrence is $T(k, m) = O(m \log k)$. Similarly, Step 4 takes $O(m' \log k')$ time, where k' and m' are the lengths of $CCchain(x_1, x_2)$ and $Cchain(y_1, y_2)$. Step 5 takes $O(|B(w)|) = O(\sqrt{\varepsilon n})$ time. The lemma follows since $k + k' = O(\sqrt{\varepsilon n})$ and $m + m' = O(\sqrt{\varepsilon n})$. $\square$

We can summarize the results from the last section in the following theorem, which significantly improves Theorem 4 in the case of planar digraphs.

Theorem 5 *Given an n-vertex planar digraph G and any $S \in [n^{4/3}, n^{3/2}]$, a data structure of size $O(S)$ can be constructed in $O(n\sqrt{S})$ time so that, for any two vertices v and w, the shortest-path distance from v to w can be computed in $O(n \log n / \sqrt{S})$ time.*

Proof: Follows from Lemma 5.1 and Lemma 5.4 by setting $\varepsilon = n/S$. $\square$

For comparison, the time bound to compute the shortest-path distance from v to w in Theorem 4 is $O(n^2/S)$, which is proportional to the square of the corresponding time bound in Theorem 5 (ignoring the logarithmic factor).

For graphs of bounded genus the Jordan Curve Theorem does not hold and the method of proof of Theorem 5 can not be directly applied. It is an interesting open problem to generalize Theorem 5 for such class of graphs.

References

1. Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows : theory, algorithms, and applications.* Prentice Hall, 1993.
2. S. Arikati, D.Z. Chen, L.P. Chew, G. Das, M. Smid, and C.D. Zaroliagis. Planar spanners and approximate shortest path queries among obstacles in the plane. In *Proceedings of ESA'96*, to appear.
3. H. Bodlaender. Dynamic algorithms for graphs with treewidth 2. In *WG'93, Lecture Notes in Computer Science, vol. 790*, pages 112–124. Springer-Verlag, Berlin, Heidelberg, New York, Tokio, 1994.
4. Shiva Chaudhuri and Christos D. Zaroliagis. Shortest path queries in digraphs of small treewidth. In *ICALP'95, Lecture Notes in Computer Science, vol. 944*, pages 244–255. Springer-Verlag, Berlin, Heidelberg, New York, Tokio, 1995.
5. E.W. Dijkstra. A note on two problems in connection with graphs. *Numer. Math*, 1:269–271, 1959.
6. H. Djidjev, G. Pantziou, and C. Zaroliagis. Computing shortest paths and distances in planar graphs. *Proc. of 18th International Colloquium on Automata Languages and Programming*, pages 327–339, 1991.
7. E. Feuerstein and A.M. Spaccamela. Dynamic algorithms for shortest paths in planar graphs. In *WG'91, Lecture Notes in Computer Science, vol 570*, pages 187–197. Springer-Verlag, Berlin, Heidelberg, New York, Tokio, 1991.
8. G. N. Frederickson and R. Janardan. Designing networks with compact routing tables. *Algorithmica*, 3:171–190, 1988.
9. G.N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM Journal on Computing*, 16:1004–1022, 1987.
10. Michael L. Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34:596–615, 1987.
11. Michael T. Goodrich. Planar separators and parallel polygon triangulation. *Proceedings of 24th Symp. on Theory of Computing*, pages 507–516, 1992.
12. Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM*, 24:1–13, 1977.
13. P. Klein, S. Rao, M. Rauch, and S. Subramanian. Faster shortest-path algorithms for planar graphs. In *26th ACM Symp. Theory of Computing*, pages 27–37, 1994.