Computing Shortest Paths and Distances in Planar Graphs*

Hristo N. Djidjev [1] *Grammati E. Pantziou* [2] *Christos D. Zaroliagis* [2]

(1) Center of Informatics and Computer Technology, Bulgarian Academy of Sciences
Acad. G. Bonchev str. bl. 25-A, 1113 Sofia, Bulgaria

(2) Computer Technology Institute, P.O. Box 1122, 26110 Patras, Greece
Computer Sc and Eng Dept, University of Patras, Greece

Abstract

We provide here efficient sequential and parallel solutions to the following problem: given a planar digraph G (with real edge weights but no negative cycles) for preprocessing, answer on-line queries requesting the shortest distance (or path) between any two vertices in G. Our algorithms for preprocessing need $O(n \log n + q^2)$ space and $O(n \log n + q^2)$ sequential time. (Here q is the cardinality of a set of faces of a planar embedding of G that cover all vertices.) A parallel implementation on a CREW PRAM needs also $O(n \log n + q^2)$ space and runs in $O(\log^2 n)$ time using $O(n + M(q))$ processors (where $M(q)$ is the number of processors required to multiply two $q \times q$ matrices in $O(\log q)$ time), provided that the q faces are given by the input. This enables us to achieve $O(\log n)$ time using a single processor for a "distance" query, or $O(L + \log n)$ time for a "path" query (where L is the length of the path). Note that this is a considerable improvement over previous results in the case where $q = o(n)$. Our techniques are based on the hammock decomposition of a planar digraph and the use of separators for computing quickly internal distances in the graph. Several other results are achieved. For outerplanar graphs, our algorithms preprocess the graph in $O(n \log n)$ space and run either in $O(n \log n)$ sequential time, or in $O(\log^2 n)$ time using $O(n)$ processors on a CREW PRAM. A "distance" query can be answered in $O(\log n)$ time using a single processor. A "path" query is answered in $O(L + \log n)$ time. An optimal solution is given in the case of trees. We achieve $O(1)$ time per "distance" query and we need $O(n)$ sequential time, or $O(\log n)$ time and $O(n/\log n)$ processors (on an EREW PRAM) for preprocessing. A "path" query is answered in $O(L)$ time.

1 Introduction

An important class of graph problems are related to finding shortest path information in graphs. Let G be a digraph with real edge weights but no negative cycles and p be a path in G with endpoints v and w. The *cost* of p is the sum of the weights of all edges of p and the *distance* between v and w is the minimum cost of a path p' joining v and w (called a *shortest path* between v and w). Many recent papers address the problem of finding shortest paths between all pairs of vertices for special classes of graphs, e.g. Fredman and Tarjan [9] propose an $O(nm + n^2 \log n)$-time algorithm for n-vertex m-edge graphs (efficient for sparse graphs), Frederickson gives an $O(n^2)$-time algorithm for planar digraphs [5], parallel algorithms and other versions of the problem have been investigated in [1, 11, 12].

Frederickson [6] shows that certain topological characteristics can be exploited in the development of efficient shortest-path algorithms in the case of planar digraphs. He defines a parameter q as the

*This work was partially supported by the EEC ESPRIT Basic Research Action No. 3075 (ALCOM), by the Ministry of Education of Greece and by a bilateral scientific agreement between Bulgaria and Greece.

minimum cardinality of a subset of the faces that cover all vertices of the digraph and gives an $O(qn)$-time algorithm for succinct encoding of shortest path information in an embedded n-vertex planar digraph ([6]).

Computing the distances between all pairs of vertices in an n-vertex digraph obviously requires $\Omega(n^2)$ time because of the output size. In some applications, e.g. in computational geometry, decision making and parallel processing it is enough to know only the distance between a small number of pairs of vertices, often given in a dynamic fashion. For such kind of applications our algorithms give performance superior to that of any previous algorithm. Informally, the first problem investigated in this paper is the following.

Problem 1: Let G be an embedded n-vertex planar digraph with real edge weights but no negative cycles. Preprocess G in order to answer in polylogarithmic time queries of the form "given any two vertices v and z find the distance between v and z".

By the best currently known algorithms that can be used for solving problem 1 one has either to compute shortest paths between all pairs and store their lengths in an $n \times n$ matrix (which will require $\Omega(n^2)$ space and time for preprocessing and $O(1)$ query time [5]) or construct compact routing tables in $O(qn)$ time and space (see [6]), in which case the distance is computed in time $O(L \log n)$, where L is the *length* of a shortest path between v and z (i.e. the number of edges which can be $\Omega(n)$). Also in [6], a different encoding of all pairs shortest path information (not only compact routing tables) provides $O(n + q^2)$ preprocessing time and space but the query time remains $O(L \log n)$.

In this paper we provide efficient sequential and parallel algorithms for the above problem. In sequential computation $O(n \log n + q^2)$ time and $O(n \log n + q^2)$ space is needed for preprocessing the planar digraph. A query can be answered in $O(\log n)$ time. Here q is the cardinality of a face-on-vertex covering, i.e. a set of faces that cover all the vertices in a planar embedding of the planar digraph. In parallel computation (in the CREW PRAM model) our methods need $O(\log^2 n)$ time using $O(n + M(q))$ processors and $O(n \log n + q^2)$ space for the preprocessing of the graph, and $O(\log n)$ time per query (using a single processor). Thus the query time of our algorithm is a considerable improvement over the $\Theta(L \log n)$ query time (which in the worst case can be $\Omega(n \log n)$ [6]). Let us also note here that if we use k processors on an CREW PRAM, then k such queries can be answered in $O(\log n)$ time using k processors.

Our methods are based: (i) on the decomposition of a planar digraph into certain outerplanar graphs called hammocks (see [6, 12]), and (ii) on the use of separators in a very convenient tree structure which allows us to compute several distances in the graph in an extremely fast way.

The heart of our algorithms for solving the shortest distance problem on a planar digraph with real edge weights but no negative cycles, is based on the solution of the same problem in an outerplanar digraph. In this case, we need for preprocessing $O(n \log n)$ space and either $O(n \log n)$ time, or $O(\log^2 n)$ time using $O(n)$ processors on a CREW PRAM. A query can be answered in $O(\log n)$ time using a single processor. (Also k such queries can be answered in $O(\log n)$ time using k processors.)

An *optimal* solution is also given to an important subproblem which is used very often as an intermediate step in our main scheme. This concerns the shortest distance problem in the case of trees. We achieve $O(n)$ sequential time, or $O(\log n)$ time and $O(n/\log n)$ processors (on an EREW PRAM) for preprocessing, $O(1)$ time per query and $O(n)$ space.

The second problem investigated in this paper is the following.

Problem 2: Let G be an embedded n-vertex planar digraph with real edge weights but no negative cycles. Preprocess G in order to answer queries of the form "given any two vertices v and z find a shortest path between v and z" in time proportional to the length of the path.

We will show that a modification of our preprocessing algorithms (for computing distances) enables us to answer on-line "path" queries in planar (or outerplanar) graphs in $O(L + \log n)$ time where L is the length of the path. In the case of trees the query is answered in $O(L)$ time. Both results are improvements over the best previous $O(L \log n)$ algorithm from [6].

2 Preliminaries

Let $G = (V, E)$ be a planar digraph with real edge weights but no negative cycles. For each edge $\langle v, w\rangle$ of G we define the *label* $S(v, w)$ of the edge as follows:

$$S(v, w) = \{u | \langle v, w\rangle \text{ is the first edge in a shortest path from } v \text{ to } u\}$$

Each $S(v, w)$ is described as a union of a minimum number of subintervals of $[1, n]$, assuming $V = \{1, 2, ..., n\}$. (A subinterval is allowed to wrap around n.)

Edge labels are used in the succinct encoding of all pairs shortest paths information in what it is called *compact routing tables* (see [15, 6]). Compact routing tables have been used for keeping shortest path information (edge labels) in a space-efficient way, either in sequential ([6]) or in parallel computation ([12]). Here, we shall consider their use only for outerplanar graphs. If G is outerplanar then each $S(v, w)$ is a single interval $[a, b]$ (see [8]). In this case a compact routing table for v consists of a list of initial values a of each interval, along with pointers to the corresponding edges. It is clear that the total size of a compact routing table for an outerplanar graph is $O(n)$.

A *hammock decomposition* of G is a decomposition of the graph into certain outerplanar subgraphs called *hammocks*. This decomposition is defined relative to a given *face-on-vertex covering* of G. Let us call $\hat{G}$ the embedding of G in the plane. Then a face-on-vertex covering $\mathcal{C}$ is a set of faces that cover all vertices in $\hat{G}$. The problem of finding the minimum face-on-vertex covering of an embedded planar digraph $\hat{G}$ is NP-complete ([2]). Frederickson in [6] gives an approximation to the above problem which is at most 4 times the optimal solution. The approximation is computed in $O(n)$ time. A parallel $O(\log n)$ approximation can be obtained using the ideas of [3] (see [12]), but it needs $O(n)$ processors and $O(\log^6 n)$ time. The question of finding an efficient parallel approximation to the face-on-vertex covering problem that is a constant multiple away of optimal still remains open.

A hammock shares at most four vertices with all other hammocks in the decomposition. These vertices are called *attachment vertices*. To generate the decomposition, we first convert $\hat{G}$ into an embedded undirected planar graph $\hat{G}_1$ called *neatly prepared* since: (a) $\hat{G}_1$ has no parallel edges; (b) if $\mathcal{C}_1$ is the face-on-vertex covering of $\hat{G}_1$ (resulting from that of $\hat{G}$) then all faces in $\hat{G}_1$ (except $\mathcal{C}_1$) are triangulated and the faces of $\mathcal{C}_1$ do not share vertices; (c) the boundary of each face is a simple cycle. We now group the faces in $\hat{G}_1$ using two operations: *absorption* and *sequencing* (see details in [6, 12]). Initially mark all edges bordering the faces belonging to $\mathcal{C}_1$. Let f_1, f_2 be two such faces (not in $\mathcal{C}_1$) that share an edge. If f_1 contains two marked edges then absorb f_1 into f_2 as follows: first contract one marked edge in f_1. The first face then becomes a face bounded by two parallel edges, one of which is shared with the second face. Delete this edge, thus merging f_1 and f_2. Repeat this operation until it can no longer be applied. Then identify maximal sequences of faces such that each face in the sequence has a marked edge and each pair of consecutive faces share an edge in common. Expand the "absorbed" faces. Each subgraph resulting from expanding such a sequence is called a *(major) hammock*. The two vertices at each end of the hammock are called *attachment vertices*. Any edge not included in a major hammock induces a *(minor) hammock*. Note that the hammock decomposition spans all the edges of a planar digraph.

The model of parallel computation used in this paper is the PRAM one. A PRAM employs P synchronous processors. Each processor may have access to a shared-memory (and to its local memory) and may execute simple word and bit operations in constant time (see [16]). By allowing or disallowing simultaneous access by more than one processor to the same memory location, we get variants of the PRAM model. In this paper we use the CREW PRAM (which allows concurrent read operations but no concurrent write ones) and the EREW PRAM (which allows neither concurrent read nor concurrent write operations to the same memory location).

The following results have been proved in [6] (the sequential version) and in [12] (the parallel version).

Theorem 1 *Given an n-vertex planar digraph G with a face-on-vertex covering of cardinality q, then G can be decomposed into $O(q)$ hammocks either in $O(n)$ sequential time, or in $O(\log^2 n)$ time using $O(n)$ processors on a CREW PRAM.*

Theorem 2 *Shortest path (edge labeling) information in an n-vertex outerplanar graph can be computed either in $O(n)$ sequential time, or in $O(\log^2 n)$ time using $O(n)$ processors on a CREW PRAM.*

It is easy to see why edge labeling is useful for finding any shortest path information. To store shortest paths or distances with respect to a fixed vertex of a graph to all other vertices it is appropriate to make use of trees.

Definition 1 *A tree is called convergent (divergent) if the edges of the tree point from a node to its parent (children).*

The following lemma has been proved in [6] (sequential version) and in [12] (parallel version).

Lemma 1 *Given edge labeling information in an n-vertex outerplanar graph G, then: i) A convergent or divergent tree of shortest paths rooted at some vertex x can be constructed either in $O(n)$ sequential time, or in $O(\log n)$ time using $O(n/\log n)$ processors on an EREW PRAM. ii) Let H be a biconnected subgraph of G_o and let $h = |H|$. Then in H, a convergent or divergent tree of shortest paths rooted at some vertex $x \in H$ can be constructed either in $O(h)$ sequential time, or in $O(\log h)$ time using $O(h/\log h)$ processors on an EREW PRAM.*

Definition 2 *An outerplanar digraph $G = (V, E)$ is called nice if, (i) for every edge $\langle v, w\rangle \in E$, its opposite $\langle w, v\rangle \in E$, (ii) the undirected version of G is biconnected, (iii) edge costs satisfy the generalized triangle inequality (i.e. $\langle v, w\rangle$ is the shortest path from v to w), and (iv) vertices are named consecutively in clockwise order around the exterior face.*

The following has been proved in [6] (sequential version) and in [12] (parallel version).

Lemma 2 *An n-vertex outerplanar graph can be converted into a nice one, either in $O(n)$ sequential time, or in $O(\log n)$ time using $O(n)$ processors on a CREW PRAM.*

Remark: The conversion back to the initial graph can be easily done within the same resource bounds (as it is described in [6, 12]).

3 Finding Shortest Distances

First we solve efficiently the problem for the class of trees and then use the solutions in the description of algorithms for outerplanar and planar graphs.

3.1 Shortest distances in trees

The algorithms for trees are simple and make use of results from [13]. Suppose we are given an ordered (rooted or unrooted) tree $T = (V_T, E_T)$ with real edge weights. (We require the tree to be ordered because of the EREW model, i.e. we have to know the relative order of the subtrees of a node, see e.g. [10], page 306.) Then simple tree functions (like pre- and post-order numbering, level, number of descendants, etc) of each node can be computed using the Euler tour technique on trees of [14] and its optimal parallel implementation in [13]. The idea is to reduce the computation of these functions to weighted list ranking. More specifically we use the following results from [13].

Lemma 3 *Simple tree functions in an n-vertex tree can be computed optimally either in $O(n)$ sequential time, or in $O(\log n)$ time using $O(n/\log n)$ processors on an EREW PRAM.*

Corollary 1 *With the appropriate preprocessing (by the previous lemma) the lowest common ancestor (LCA) of any two nodes in a tree can be computed in $O(1)$ time using a single processor.*

Our algorithm for preprocessing a tree is the following.

ALGORITHM *P-TREE*
(1) compute the necessary tree functions in order to answer LCA queries in $O(1)$ time.
(2) for each node v of the tree compute its weighted distance $d(v,r)$ from the root r.

Lemma 4 *The P-TREE algorithm runs either in $O(n)$ sequential time, or in $O(\log n)$ time using $O(n/\log n)$ processors on an EREW PRAM.*

Proof: By lemma 3. ∎

Given any two nodes $v, z \in V_T$, the query algorithm for finding their distance $d(v,z)$ is given below.

ALGORITHM *Q-TREE*
(* let $LCA(v,z)$ be the lowest common ancestor of v,z and r be the root of the tree *)
(1) find $LCA(v,z)$.
(2) $d(v,z) = d(v,r) + d(z,r) - 2d(LCA(v,z),r)$.

Lemma 5 *The Q-TREE algorithm finds the shortest distance between any two nodes of a tree in $O(1)$ time using a single processor.*

Proof: In the full paper [4]. ∎
Remark: If concurrent read operations are allowed, then we can answer k queries in $O(1)$ time using k processors.

3.2 Shortest distances in outerplanar graphs

Let us assume that we are given an n-vertex embedded outerplanar digraph G_o (with real edge costs but no negative cycles).

Definition 3 *A separator edge is called a good separator for G_o, if the removal of its endpoints from G_o, divides G_o into subgraphs of sizes $\leq \varepsilon n$ for some constant $(1/2) \leq \varepsilon < 1$.*

We will make use of the well known fact that a good separator (for $\varepsilon = (2/3)$) exists for any triangulated outerplanar graph and that it can be found in $O(n)$ time. Our algorithm for preprocessing G_o is the following.

ALGORITHM *P-OUTERPLANAR*
Step 1
Convert G_o into a *nice* outerplanar graph G_n. Then triangulate each face of G_n and with each new edge added associate a weight equal to the corresponding shortest distance between its endpoints (see [6] and [12]). Let us call the resulting graph H_0.
Step 2
Find shortest path (edge labeling) information in H_0.
Step 3
(3.a) Perform a sequence of $O(\log n)$ phases building a binary tree of good separators. Let S_j^i denote the j^{th} good separator of the i^{th} phase ($i = 0, 1, ..., imax$ ($imax = O(\log n)$), $j = 1, 2, ..., 2^i$). The root of the tree is the separator S_1^0 which divides H_0 into two subgraphs H_{11} and H_{12} of sizes εn and $(1-\varepsilon)n$ respectively. Then we find good separators S_1^1 and S_2^1 in H_{11} and H_{12} respectively. The separators S_1^1 and S_2^1 are the children of S_1^0. This process is repeated recursively in each new subgraph produced. In general, the node S_j^i will have S_{2j-1}^{i+1} and S_{2j}^{i+1} as its children. With each node S_j^i we associate a 4-tuple (I_1, I_2, I_3, I_4) where each I_m ($m = 1, 2, 3, 4$) is an interval. Let the endpoints of S_j^i be $u(S_j^i)$ and $w(S_j^i)$. Then the four intervals for S_j^i are shown in figure 1 (with their boundary vertices). Now, for S_{2j-1}^{i+1} we have: $b_1(S_{2j-1}^{i+1}) = b_1(S_j^i)$, $b_2(S_{2j-1}^{i+1}) = u(S_j^i)$, $b_3(S_{2j-1}^{i+1}) = w(S_j^i)$ and $b_4(S_{2j-1}^{i+1}) = b_4(S_j^i)$. For the node S_{2j}^{i+1} we have similarly, $b_1(S_{2j}^{i+1}) = u(S_j^i)$, $b_2(S_{2j}^{i+1}) = b_2(S_j^i)$,

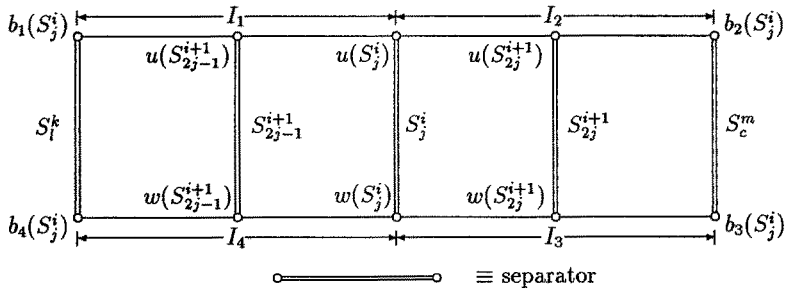

Figure 1: Construction of separator tree.

$b_3(S_{2j}^{i+1}) = b_3(S_j^i)$ and $b_4(S_{2j}^{i+1}) = w(S_j^i)$. Note that initially ($i = 0$) the intervals I_1 and I_4 (and also the intervals I_2 and I_3) coincide. Let us call the above tree as the *separator tree.*

(3.b) For each S_j^i $i = 0, ..., imax$ ($imax = O(\log n)$), $j = 1, 2, ..., 2^i$), create convergent and divergent shortest path trees rooted at $u(S_j^i)$ and $w(S_j^i)$ and involving only those vertices in $[b_1(S_j^i), b_2(S_j^i)]$ and $[b_3(S_j^i), b_4(S_j^i)]$. (Therefore, we have 4 such trees and associate this information to the node S_j^i by creating pointers to the corresponding trees.) Run the algorithm *P-TREE* to each of these trees, in order to be able to answer queries (for the trees) in $O(1)$ time.

Lemma 6 *Algorithm P-OUTERPLANAR runs in $O(n \log n)$ time and needs $O(n \log n)$ space. A parallel CREW PRAM implementation of the algorithm runs in $O(\log^2 n)$ time and uses $O(n)$ processors.*

Proof: G_o can be converted into a nice outerplanar graph G_n either in $O(n)$ sequential time, or in $O(\log n)$ time using $O(n)$ processors on a CREW PRAM by lemma 2. Also the triangulation of faces and assignment of appropriate weights on the new edges can be done as follows: in each internal face of G_n choose a vertex v to be the starting (and ending) point of a clockwise or counterclockwise walk around the face. Then find the unweighted clockwise distance (ranking) of any vertex in the face from v. Do the same but now for the weighted clockwise and counterclockwise distance from v. Suppose that v has $rank(v) = 1$. Then for any vertex u such that $rank(u) \geq 3$ do: (i) add the edge $\langle v, u \rangle$ to the graph with weight the minimum between the clockwise and counterclockwise distance from v to u, (ii) add the edge $\langle u, v \rangle$ with weight the minimum between the clockwise and counterclockwise distance from u to v. Note that these distances can be easily computed. For example, the clockwise distance from u to v is equal to the clockwise distance of the face (the sum of the weights of the edges of the face) minus the clockwise distance from v to u. Thus the triangulation can be done sequentially in $O(n)$ time, while in parallel computation can also be easily computed in $O(\log n)$ time and $O(n)$ processors by pointer doubling. In conclusion, step 1 needs either $O(n)$ sequential time, or $O(\log n)$ time and $O(n)$ processors on a CREW PRAM.

Step 2 needs either $O(n)$ sequential time, or $O(\log^2 n)$ time using $O(n)$ processors on a CREW PRAM, by theorem 2.

In step 3, the discovering of the appropriate separators is facilitated by the naming of the vertices in H_0. For example suppose we want to find S_1^0 in H_0. Since we have $O(n)$ edges we can find it in $O(n)$ sequential time. Note that the naming of the vertices and the triangulation of the faces guarantees that we can find a good separator. To do this in parallel, we associate a processor with each edge of H_0. Then on a CREW PRAM we need $O(\log n)$ time to find S_1^0. We proceed similarly for the other separators. The depth of the separator tree is $O(\log n)$ and in each level we need either $O(n)$ sequential time, or $O(\log n)$ time and $O(n)$ processors in order to find the separators. Thus, step (3.a) is implemented either in $O(n \log n)$ sequential time or in $O(\log^2 n)$ time using $O(n)$ processors on a CREW PRAM. Also the time and processor bounds to implement step (3.b) are the same as those of (3.a). This is because for each level of the tree we build convergent and divergent

shortest path trees which (by lemma 1) need either $O(n)$ sequential time, or $O(\log n)$ time and $O(n)$ processors for the whole graph.

Finally for the space needed by the algorithm one can easily observe that all steps need $O(n)$ space except for step (3.b) which needs $O(n \log n)$ space. ∎

Given any two vertices v, z the query algorithm for finding their shortest distance is as follows.

ALGORITHM *Q-OUTERPLANAR*
(1) search the separator tree (starting from the root) to find a node S_j^i such that
$(v \in I_1 \vee v \in I_4) \wedge (z \in I_2 \vee z \in I_3)$ (or vice versa).
(* let e_1, e_2 be the endpoints of the separator found *)
(2) $d(v,z) = \min\{d(v,e_1) + d(e_1,z), d(v,e_2) + d(e_2,z)\}$

Lemma 7 *Algorithm Q-OUTERPLANAR finds the shortest distance between any two vertices in an outerplanar digraph in $O(\log n)$ time using a single processor.*

Proof: We first establish the correctness. Let us assume that v, z are the two vertices and we want to compute $d(v,z)$. Note that it is enough to find a good separator such that v belongs either to I_1 or I_4 and z belongs either to I_2 or to I_3 (or vice versa). In such a case the shortest path from v to z would go through one of the endpoints of the separator. Let e_1, e_2 be these endpoints. Then $d(v,z) = \min\{d(v,e_1)+d(e_1,z), d(v,e_2)+d(e_2,z)\}$. Therefore the approach followed by the algorithm *Q-OUTERPLANAR* correctly determines the shortest distance between v and z.

Now, for the resource bounds we have that steps 1 and 2 need $O(\log n)$ time (the time needed to search a binary tree of depth $O(\log n)$). Step 3 needs $O(1)$ time, because the distances $d(v,e_1)$, $d(e_1,z)$, $d(v,e_2)$, $d(e_2,z)$ can be computed in $O(1)$ time from the convergent and divergent shortest path trees constructed during the preprocessing phase. ∎

Theorem 3 *Given an n-vertex outerplanar digraph G_o, we can preprocess G_o in $O(n \log n)$ space and either in $O(n \log n)$ sequential time, or in $O(\log^2 n)$ time using $O(n)$ processors on a CREW PRAM. This preprocessing enables us to answer queries requesting the shortest distance between any two vertices in $O(\log n)$ time.*

Proof: Immediate consequence of lemmas 6 and 7. ∎

3.3 Shortest distances in planar graphs

The preprocessing algorithm here is based on the hammock decomposition technique and on the algorithms of the previous section. Given an embedded planar digraph G with a face-on-vertex covering of cardinality q, the preprocessing algorithm is the following.

ALGORITHM *P-PLANAR*
Step 1
Find a hammock decomposition of G into $O(q)$ hammocks. For each vertex of G store a pointer to one of the hammocks it belongs to.
Step 2
Run the algorithm *P-OUTERPLANAR* in each hammock.
Step 3
Find all shortest distances and paths between the attachment vertices of the hammocks. This can be done by first compressing each hammock into an $O(1)$ size graph and then compressing G into a planar graph of size $O(q)$.
Step 4
Construct convergent and divergent trees of shortest paths rooted at each attachment vertex of a hammock.

Lemma 8 *Algorithm P-PLANAR runs in $O(n \log n + q^2)$ sequential time and needs $O(n \log n + q^2)$ space. A parallel CREW PRAM implementation runs in $O(\log^2 n)$ time and uses $O(n + M(q))$ processors.*

Proof: Step 1 requires (by theorem 1) either $O(n)$ time, or $O(\log^2 n)$ time using $O(n)$ processors. Step 2 takes (by theorem 3) either $O(n \log n)$ time, or $O(\log^2 n)$ time using $O(n)$ processors. The implementation of step 3 is described in [6] and [12], and needs either $O(q^2)$ time (by [5]), or $O(\log^2 q)$ time using $M(q)$ processors (where $M(q)$ is the number of processors required to multiply two $q \times q$ matrices). Finally step 4 takes (for all hammocks) either $O(n)$ time, or $O(\log n)$ time using $O(n)$ processors. For the space needed by the algorithm we have that step 1 needs $O(n)$ space, step 2 needs $O(n \log n)$ space (by lemma 6) and step 3 needs $O(q^2)$ space. ∎

The query algorithm for finding the shortest distance between any two vertices v, z is the following.

ALGORITHM *Q-PLANAR*
(* let H, H' be the hammocks with attachment vertices $a_i, 1 \leq i \leq 4$ and $a'_i, 1 \leq i \leq 4$, respectively, such that v is in H and z is in H' *)
if $H \equiv H'$ (* i.e. both v, z belong to H *) **then**
(1) run the *Q-OUTERPLANAR* algorithm in H and let $d_H(v, z)$ be the output of that algorithm
(2) $d_{ij}(v, z) = \min_{i,j}\{d(v, a_i) + d(a_i, a_j) + d(a_j, z)\}$
(3) $d(v, z) = \min\{d_H(v, z), d_{ij}(v, z)\}$
else (* $H \neq H'$ *)
$d(v, z) = \min_{i,j}\{d(v, a_i) + d(a_i, a'_j) + d(a'_j, z)\}$

Lemma 9 *Algorithm Q-PLANAR finds the shortest distance between any two vertices in a planar digraph (with real edge weights but no negative cycles) in $O(\log n)$ time using a single processor.*

Proof: We first establish the correctness of the algorithm. If v, z do not belong to the same hammock, then the correctness is obvious. If v, z belong to the same hammock H, then the shortest path between v and z does not necessarily stay in H. Let $d_H(v, z)$ be the shortest distance between v and z corresponding to the shortest path (between v, z) that stays in H. Let also $d_{ij}(v, z)$ be the shortest distance between v, z corresponding to a path that leaves H at one attachment vertex a_i and reenters H at another attachment vertex a_j (and this is the shortest one for all pairs of i, j). Then clearly the shortest distance $d(v, z)$ is the minimum between $d_H(v, z)$ and $d_{ij}(v, z)$.

It is not difficult to show that the else part takes $O(1)$ time, since it involves finding the minimum among 16 weights. The time of the if part is dominated by the running time of algorithm *Q-OUTERPLANAR* which is $O(\log n)$ by lemma 7. (All other steps obviously need $O(1)$ time.) ∎

Theorem 4 *Given an embedded n-vertex planar digraph with a face-on-vertex covering of cardinality q (with real edge weights but no negative cycles), we can preprocess it in $O(n \log n + q^2)$ space and either in $O(n \log n + q^2)$ sequential time, or in $O(\log^2 n)$ time using $O(n + M(q))$ processors on a CREW PRAM. This preprocessing enables us to answer queries requesting the shortest distance between any two vertices in $O(\log n)$ time.*

Proof: Immediate consequence of lemmas 8 and 9. ∎

4 Finding Shortest Paths

In this section we show how to answer on-line queries requesting the shortest path between any two vertices in trees, outerplanar and planar graphs by doing modifications in the preprocessing algorithms presented in the previous section. In the sequel, let $SP(v, z)$ denote the shortest path between v and z and let $SP_H(v, z)$ denote the shortest path between v and z in a connected subgraph H of the input graph.

4.1 Trees

Here the preprocessing algorithm remains the same. Given any two nodes v, z of a tree the query algorithm for finding their shortest path ($SP(v,z)$) is the following.

ALGORITHM Q'-TREE
(1) find $LCA(v,z)$
(2) find $SP(v, LCA(v,z))$ and $SP(LCA(v,z), z)$ and join them.

Lemma 10 *Algorithm Q'-TREE finds the shortest path between any two nodes of an n-vertex tree in $O(L)$ sequential time, where L is the length of the path. A parallel implementation finds the vertices of the shortest path in $O(1)$ time and uses $O(n)$ processors on an EREW PRAM.*

Proof: The sequential implementation is straightforward. For the parallel implementation we associate a processor with each edge (u,w) (the edges are directed towards the root). Denote $M = \{w | w \in SP(v,z)\} = \{w | (w = LCA(v,z)) \vee ((LCA(v,w) = w) \oplus (LCA(z,w) = w))\}$. (Here "$\oplus$" denotes the "exclusive-or" operation.) Therefore M can be determined in $O(1)$ time. ∎

4.2 Outerplanar graphs

First note that if in the input graph G_o ($|G_o| = n$) the generalized triangle inequality is satisfied we have nothing to do for the preprocessing of the graph. Also the shortest path can be easily found using the *Q-OUTERPLANAR* algorithm and the algorithm *Q'-TREE* (working in the convergent and divergent shortest path trees rooted at the endpoints of the separator found).

To handle the case where the generalized triangle inequality is not satisfied we modify the *P-OUTERPLANAR* algorithm. The new algorithm is as follows.

ALGORITHM *P'-OUTERPLANAR*
Step 1
Find shortest path (edge labeling) information in G_o.
Step 2
(2.a) Execute step (3.a) of algorithm *P-OUTERPLANAR*, but in the case where we can not find a good separator (since such a separator does not exist), we take the separator minimizing the difference of the sizes of the resulting subgraphs. Note that the corresponding separator tree is still of depth $O(\log n)$.
(2.b) Execute step (3.b) of algorithm *P-OUTERPLANAR*.
Step 3
We proceed in phases. During phase i, $i = 0, 1, ..., imax$ ($imax = O(\log n)$), we do the following: for each S_j^i, $j = 1, 2, ..., 2^i$, we find and keep the shortest path (and distance) between its endpoints $u(S_j^i)$ and $w(S_j^i)$. This is done by using the information of the convergent and divergent shortest path trees rooted at these endpoints and the shortest paths between $b_1(S_j^i)$ and $b_4(S_j^i)$ (endpoints of S_l^k) and also between $b_2(S_j^i)$ and $b_3(S_j^i)$ (endpoints of S_c^m). (See figure 1.) Note that these shortest paths have been computed in a previous phase.

Lemma 11 *Algorithm P'-OUTERPLANAR runs in $O(n \log n)$ sequential time and needs $O(n \log n)$ space. A parallel implementation on a CREW PRAM needs $O(\log^2 n)$ time and uses $O(n)$ processors.*

Proof: Step 1 needs $O(n)$ sequential time (by [6]) or $O(\log^2 n)$ parallel time using $O(n)$ processors (by [12]). In step 2, if there is not a good separator, then let x, y be a pair of vertices that separate G_o into two subgraphs of size $\leq \varepsilon n$, $(1/2) \leq \varepsilon < 1$. Add the dummy-edges $\langle x, y \rangle$ and $\langle y, x \rangle$ with weight ∞. Obviously, this pair of edges constitutes the good separator we are looking for. But nothing changes if instead of doing this, we choose between the "neighbouring" separators (that really exist left and right of the dummy-edges) that one which minimizes the difference of the sizes of the subgraphs of G_o. Therefore, step 2 needs the same resource bounds as those of step 3 in

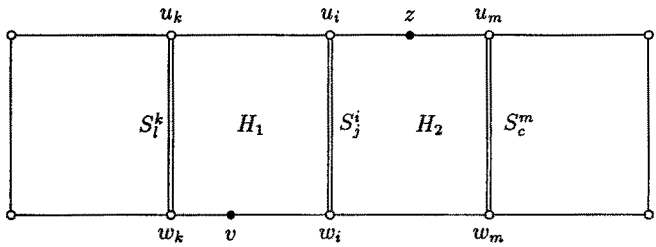

Figure 2: Computation of $SP(u_i, w_i)$.

algorithm *P-OUTERPLANAR*. Let us discuss in more detail how to implement step 3 of the above algorithm efficiently. Consider figure 2 with some abuse of notation. $SP(u_i, w_i)$ either stays in $H_1 \cup H_2$ (denoted in the sequel as $H12$), or (since the generalized triangle inequality is not satisfied) leaves at one endpoint of S_l^k and reenters at the other, or leaves at one endpoint of S_c^m and reenters at the other. Thus,
$SP(u_i, w_i) = \min\{SP_{H12}(u_i, w_i)$,
$\min\{SP_{H12}(u_i, u_m) + SP(u_m, w_m) + SP_{H12}(w_m, w_i), SP_{H12}(u_i, w_m) + SP(w_m, u_m) + SP_{H12}(u_m, w_i)\}$,
$\min\{SP_{H12}(u_i, u_k) + SP(u_k, w_k) + SP_{H12}(w_k, w_i), SP_{H12}(u_i, w_k) + SP(w_k, u_k) + SP_{H12}(u_k, w_i)\}\}$

Here (and in the sequel) "min" denotes minimum in distance and "+" denotes path addition (i.e. concatenation). Using a similar expression we can find $SP(w_i, u_i)$. If $SP(u_i, w_i)$ stays in $H12$ then we keep the entire path (using the convergent and divergent shortest path trees). If, for example, $SP(u_i, w_i)$ leaves $H12$ at u_m and reenters $H12$ at w_m then we keep the paths $SP_{H12}(u_i, u_m)$ and $SP_{H12}(w_m, w_i)$ (from the shortest path trees) and create a pointer to the path $SP(u_m, w_m)$ (determined in a previous phase). Thus in each phase of step 3 we need $O(1)$ time to decide which is the shortest path (recall the expression giving $SP(u_i, w_i)$) and $O(|H_1| + |H_2|)$ sequential time and space to find and keep the path(s) that stay in $H12$ (by lemma 10) which gives $O(n)$ time and space for the whole graph. In parallel (by lemma 10) we need $O(1)$ time and $O(|H_1| + |H_2|)$ processors to find and keep the paths. ∎

The query algorithm for finding the shortest path between any two vertices v and z is the following.

ALGORITHM *Q'-OUTERPLANAR*
(1) search the separator tree (starting from the root) to find a node S_j^i such that
$(v \in I_1 \vee v \in I_4) \wedge (z \in I_2 \vee z \in I_3)$ (or vice versa).
(* let u_i, w_i be the endpoints of the separator found *)
(2) $SP(v, z) = \min\{SP(v, u_i) + SP(u_i, z), SP(v, w_i) + SP(w_i, z)\}$,

where
$SP(v, u_i) = \min\{\min\{SP_{H1}(v, u_i), \min\{SP_{H1}(v, w_k) + SP(w_k, u_k) + SP_{H1}(u_k, u_i),$
$SP_{H_1}(v, u_k) + SP(u_k, w_k) + SP_{H1}(w_k, u_i)\}\}, SP_{H1}(v, w_i) + SP(w_i, u_i)\}$

$SP(u_i, z) = \min\{\min\{SP_{H2}(u_i, z), SP(u_i, w_i) + SP_{H2}(w_i, z)\}, \min\{SP_{H2}(u_i, u_m)$
$+ SP(u_m, w_m) + SP_{H2}(w_m, z), SP_{H2}(u_i, w_m) + SP(w_m, u_m) + SP_{H2}(u_m, z)\}\}$

(For $SP(v, w_i)$ and $SP(w_i, z)$ we have similar expressions to those of $SP(v, u_i)$ and $SP(u_i, z)$ respectively.)

Lemma 12 *Algorithm Q'-OUTERPLANAR finds the shortest path between any two vertices in an outerplanar graph in $O(L + \log n)$ time using a single processor.*

Proof: The correctness follows from the above discussion. Answering the query involves the computation of $SP(v,u_i), SP(u_i,z), SP(u,w_i), SP(w_i,z)$. But these shortest paths can be easily computed from the preprocessing of the graph. The distances of those paths are computed in $O(1)$ time for each level of the separator tree. Thus in $O(\log n)$ time we know which path is the shortest one and in $O(L)$ time we can construct it. ∎

4.3 Planar graphs

Here the algorithms are almost the same as those for computing shortest distances. If in step 2 of the *P-PLANAR* algorithm we substitute the call to *P-OUTERPLANAR* with a call to *P'-OUTERPLANAR* then we have algorithm *P'-PLANAR*. Also in algorithm *Q-PLANAR* we substitute the call to *Q-OUTERPLANAR* with a call to *Q'-OUTERPLANAR* and in each computation of a distance we compute also its corresponding shortest path. This results to algorithm *Q'-PLANAR*.

Theorem 5 *Given an embedded n-vertex planar digraph with a face-on-vertex covering of cardinality q (with real edge weights but no negative cycles), we can preprocess it in $O(n \log n + q^2)$ space and either in $O(n \log n + q^2)$ sequential time, or in $O(\log^2 n)$ time using $O(n + M(q))$ processors on a CREW PRAM. This preprocessing enables us to answer queries requesting the shortest paths between any two vertices in $O(L + \log n)$ time using a single processor, where L is the length of the path.*

Proof: In the full paper [4]. ∎

5 Closing Remarks

If in each step of our *P(P')-OUTERPLANAR* algorithm we find a separator of size $O(\sqrt{n})$ (rather than an $O(1)$-size separator), then we can reduce the sequential preprocessing time and space but increase a little bit the query time. In this case the separator tree has depth $O(\log \log n)$. This leads to the following theorem.

Theorem 6 *Given an n-vertex outerplanar (planar) digraph G, we can preprocess G in $O(n \log \log n)$ ($O(n \log \log n + q^2)$) time and space. This preprocessing enables us to answer queries requesting, (i) the shortest distance between any two vertices in $O(\log n \log \log n)$ time, and (ii) the shortest path between any two vertices in $O(L + \log n \log \log n)$ time.*

Proof: In the full paper [4]. ∎

Unfortunately this idea does not improve the preprocessing resource bounds in parallel computation.

We can achieve a trade-off between the query time and the sequential preprocessing time and space by using separators that separate the outerplanar graph into components of size not exceeding $k(n)$ (instead of $\sqrt{n}$), where $k(n) \in [2, \sqrt{n}]$. For instance if $k(n) = \lceil \log n \rceil$, then the depth of the separator tree is $O(\log n / \log \log n)$ and we have $O(n \log n / \log \log n)$ preprocessing time and space. This preprocessing provides $O(\log n)$ time per "distance" query and $O(L + \log n)$ time per "path" query.

Our results hold for the more general class of nonplanar sparse graphs as considered in [7, 8], but in order to omit a large introductory part we described our techniques for the class of planar graphs.

It will be interesting to provide lower bounds relating the preprocessing and query times for the problems considered in this paper.

Acknowledgments: We are grateful to Paul Spirakis for his help in many technical discussions.

References

[1] G. Ausiello, G.F. Italiano, A.M. Spaccamela, U. Nanni, "Incremental algorithms for minimal length paths", *Proc. of ACM-SIAM SODA*, 1990, pp.12-21.

[2] D. Beinstock, C.L. Monma, "On the complexity of covering faces by vertices in a planar graph", *SIAM J. Comp.* , Vol.17, No.1, Feb. 1988, pp.53-76.

[3] B. Berger, J. Rompel, P.W. Shor, "Efficient NC-algorithms for set cover with applications to learning and geometry", *Proc. 30th IEEE Symp. on FOCS*, 1989, pp.54-59.

[4] H. Djidjev, G. Pantziou, C. Zaroliagis, "Computing Shortest Paths and Distances in Planar Graphs", CTI TR-90.10.26, Computer Technology Institute, Patras, October 1990.

[5] G.N. Frederickson, "Fast Algorithms for Shortest Paths in Planar Graphs with Applications", *SIAM J. Comp.* , Vol.16, 1987, pp.1004-1022.

[6] G.N. Frederickson, "Planar Graph Decomposition and All Pairs Shortest Paths", TR–89-015, ICSI, Berkeley, March 1989. A preliminary version was appeared as "A new approach to all pairs shortest paths in planar graphs", *Proc. 19th ACM STOC*, New York City, May 1987, pp.19-28.

[7] G.N. Frederickson "Using Cellular Graph Embeddings in Solving All Pairs Shortest Paths Problems", CSD–TR-897, Purdue University, August 1989. A preliminary version was appeared in *Proc. 30th IEEE Symp. on FOCS*, 1989, pp.448-453.

[8] G.N. Frederickson, R. Janardan, "Designing Networks with Compact Routing Tables", *Algorithmica*, 3 (1988), pp.171-190.

[9] M.L. Fredman, R.E. Tarjan, "Fibonacci heaps and their use in improved network optimization algorithms", *J. ACM*, 34(1987), pp.596-615.

[10] D. Knuth, "The Art of Computer Programming", Vol.1, Fundamental Algorithms, 2nd ed. Addison-Wesley, 1973.

[11] A. Lingas, "Efficient parallel algorithms for path problems in planar directed graphs", *Proc. of SIGAL90*, LNCS, Vol.450, pp. 447-457.

[12] G. Pantziou, P. Spirakis, C. Zaroliagis, "Efficient Parallel Algorithms for Shortest Paths in Planar Graphs", CTI TR–90.01.02, Computer Technology Institute, Patras, September 1990 (revised version). A preliminary version has appeared as *Proc. of the 2nd Scand. Workshop on Algorithm Theory (SWAT90)*, Bergen, Norway, 11-14 July, 1990, *LNCS*, Vol. 447, pp.288-300, Springer-Verlag.

[13] B. Schieber, U. Vishkin, "On finding lowest common ancestors: simplification and parallelization", *Proc. 3rd AWOC88*, Corfu, Greece, July 1988, LNCS Vol. 319 (ed. J.H. Reif), pp.111-123, Springer-Verlag.

[14] R.E. Tarjan, U. Vishkin, "An efficient parallel biconnectivity algorithm", *SIAM J. Comp.* , 14 (1985), pp. 862-874.

[15] J. van Leeuwen, R.B. Tan, "Computer Networks with compact routing tables", in *The Book of L*, G. Rozenberg and A. Salomaa (eds.), Springer-Verlag, New York (1986), pp.259-273.

[16] J.C. Wyllie, "The Complexity of Parallel Computation", PhD Thesis, TR 79-387, Dept of Computer Science, Cornell University, Ithaca, NY, 1979.