BIT 32 (1992), 215–236.

EFFICIENT PARALLEL ALGORITHMS FOR SHORTEST PATHS IN PLANAR DIGRAPHS*

GRAMMATI E. PANTZIOU[1], PAUL G. SPIRAKIS[1,2] and CHRISTOS D. ZAROLIAGIS[1]

(1) *Computer Technology Institute, P.O. Box 1122, 26110 Patras, Greece*

(2) *Courant Institute of Math. Sciences, New York University, 251 Mercer Street, New York, NY 10012, USA*

Abstract.

Efficient parallel algorithms are presented, on the CREW PRAM model, for generating a succinct encoding of all pairs shortest path information in a directed planar graph G with real-valued edge costs but no negative cycles. We assume that a planar embedding of G is given, together with a set of q faces that cover all the vertices. Then our algorithm runs in $O(\log^2 n)$ time and employs $O(nq + M(q))$ processors (where $M(t)$ is the number of processors required to multiply two $t \times t$ matrices in $O(\log t)$ time). Let us note here that whenever $q < n$ then our processor bound is better than the best previous one ($M(n)$). $O(\log^2 n)$ time, n-processor algorithms are presented for various subproblems, including that of generating all pairs shortest path information in a directed *outerplanar* graph. Our work is based on the fundamental hammock-decomposition technique of G. Frederickson. We achieve this decomposition in $O(\log n \log^* n)$ parallel time by using $O(n)$ processors. The hammock-decomposition seems to be a fundamental operation that may help in improving efficiency of many parallel (and sequential) graph algorithms.

CR Categories: GT: Algorithm, SD: G1.0, G2.2, F2.2.

Keywords: Planar digraph, outerplanar digraph, shortest path, hammock decomposition, hammock, compact routing tables, parallel tree contraction, CREW PRAM.

1. Introduction

Computing shortest path information is one of the fundamental problems in digraphs. Efficient *sequential* algorithms for this problem have been proposed (see e.g. [5, 8, 12, 10, 13]). In some of the more recent works, the idea of exploiting topological features of the input graph (such as edge sparsity) has been emphasized. The very recent work of [11] deals with n-vertex weighted planar digraphs with real weights but no negative cycles and produces a *succint encoding of all pairs shortest*

* This work was partially supported by the EEC ESPRIT Basic Research Action No. 3075 (ALCOM) and by the Ministry of Industry, Energy and Technology of Greece.

Received January 1991. Revised February 1992.

paths in time ranging from $O(n)$ up to $O(n^2)$, depending on the topological properties of the input graph. This fundamental technique is based on a decomposition of the digraph into special outerplanar subgraphs, called *hammocks*. The decomposition is defined relative to a given face-on-vertex covering of a planar graph, i.e. a set of faces that together cover all vertices. Frederickson (in [11]) manages to approximate (by a constant factor) the *minimum* cardinality covering over *all* embeddings of the graph, in $O(n)$ time. If this cardinality is p, then the shortest path information is computed in $O(np)$ time, by Frederickson's techniques. This decomposition seems to be a fundamental operation that may help in improving efficiency of many sequential and parallel graph algorithms.

The hammock decomposition idea together with the use of succinct encoding (compact routing tables) of shortest paths, manages to avoid the $\Omega(n^2)$ sequential time lower bound. Thus, an efficient parallelization of these techniques might lead to a number of processors much less than $M(n)$ (i.e. the number required for multiplication of two $n \times n$ matrices in $O(\log n)$ time) and, thus tighten the processor-time tradeoff. Up to now, parallel algorithms for all pairs shortest paths in planar digraphs have been considered in e.g. [6] and led to an $O(\log n \log d)$ CREW PRAM algorithm (where d is the *depth* of the digraph), by using $M(n)$ processors. Let us note that d (the maximum of the diameters of the connected components) may be $O(n)$, while the best value for $M(n)$ up to now is $O(n^{2.376})$. Lingas (in [20]) presents a preprocessing of planar digraphs which takes $O(\log^4 n)$ time using a CRCW PRAM with $O(n^{1.5})$ processors. After this preprocessing any single source problem can be solved in $O(\log^2 n)$ time using $O(n)$ processors. The preprocessing algorithm runs in a probabilistic PRAM since it uses the randomized parallel algorithm for a simple cycle $O(\sqrt{n})$ separator in planar digraphs due to Gazit and Miller (see [14]).

Our methods manage to compute a succinct encoding of all pairs shortest path information in a planar weighted digraph G (with real edge weights but no negative cycles) in time $O(\log^2 n)$ and $O(nq + M(q))$ processors in the CREW PRAM. Here q is the cardinality of a face-on-vertex covering $\mathscr{C}$ for a plane embedding $\hat{G}$ of the input graph G. We note that whenever $q < n$, our processor bound is better than the best previous one ($M(n)$) and that our algorithm is efficient in the case where $q \leq n^{0.72}$ (since the sequential algorithm runs in $O(nq)$ time). We note also that both $\mathscr{C}$ and $\hat{G}$ are provided by the input to our techniques. (Plane embeddings of a planar n-vertex graph can be constructed in $O(\log^2 n)$ time by using $O(n)$ processors in the CREW PRAM model (see [18]). The problem of finding a minimum face-on-vertex covering was shown to be NP-complete in [2]. In [23] we discuss how a parallel $O(\log n)$ approximation of a face-on-vertex covering can be achieved, using the ideas of [3], but although it uses $O(n)$ processors, it seems to need $O(\log^6 n)$ time.)

Our work efficiently parallelizes the hammock decomposition technique of [11]. A combination of algorithmic techniques (such as shunt/rake tree operations, connected components, etc.) and of graph properties of the hammock decomposition are essential in our method. The efficient parallelization of the hammock decomposition technique has been proved very useful for solving other problems in planar digraphs. We mention here (i) the work in [7] which presents a preprocessing of a planar digraph G for answering on-line queries requesting the shortest distance between any two vertices in G in logarithmic time using a single processor and (ii) the work in [17] for detecting a negative cycle and constructing a depth first tree in a planar digraph.

We must note here that our algorithm for solving the all paths problem in a planar digraph follows the main steps of the algorithm used in [11]. But in many cases the parallelization of those steps is not easy and usually needs a completely different approach. For this reason, efficient parallel algorithms for several important sub-problems (on a CREW PRAM) are also presented here, which are used as intermediate steps in our main scheme. These include: (a) the decomposition of a planar digraph into hammocks in $O(\log n \log^* n)$ parallel time using $O(n)$ processors, (b) the finding of all pairs shortest paths information in *outerplanar* weighted digraphs (with real edge costs but no negative cycles) in parallel time $O(\log^2 n)$ and $O(n)$ processors, and (c) finding the all pairs shortest paths information between vertices of a hammock of h vertices, (even when the shortest paths have to leave and reenter the hammock) in time $O(\log^2 h)$ using $O(h)$ processors. There are also some steps that can be parallelized more easily. These steps include: finding the compressed version of a hammock and finding all pairs shortest path information between any two hammocks. We shall not go in detail during the description of those steps. (The reader can either understand – without too much difficulty – the details of the parallelization from the description in [11], or see [23].)

A preliminary version of this paper appeared as [22].

2. Definitions and notation.

A graph is called *outerplanar* if it can be embedded in the plane so that all vertices are on one face. If G is a planar graph and $\hat{G}$ an embedding of G in the plane, let a *face-on-vertex covering* $\mathscr{C}$ be a set of faces of $\hat{G}$ that together cover all vertices.

Let $G = (V, E)$ be a planar digraph with real edge costs but no negative cycles. For each edge $\langle v, w\rangle$ incident from a vertex v, let $S(v, w)$ be the set of vertices such that there is a shortest path from v to each vertex in $S(v, w)$ with the first edge on this path being $\langle v, w\rangle$. A *tie* might occur if there is a vertex u with a shortest path from v to u starting with $\langle v, w\rangle$ and also a shortest path from v to u starting with $\langle v, w'\rangle$ where $w' \neq w$. In such a case, we employ an appropriate tie-breaking rule so that $\forall u, v$ $(u \neq v)$, u is *in just one set* $S(v, w)$ for some w. Let each $S(v, w)$ be described as a union of a minimum number of subintervals of $[1, n]$, assuming that $V = \{1, 2, \dots, n\}$. Here we allow a subinterval to wrap around from n to 1. For example, the set $\{i, i+1, \dots, n, 1, \dots, j-1\}$ where $i > j$, is described by the interval $[i, j)$. We call the set $S(v, w)$ (described as above) the *label* of edge $\langle v, w\rangle$.

We shall now discuss the notion of *compact routing tables* (see [24, 11]) which help us to keep shortest path information in a space-efficient way that is very useful in many applications (e.g. message routing in networks, see [9]). It was shown (in [9]) that if G is undirected outerplanar and vertices are named in clockwise order around the exterior face, then each $S(v, w)$ is a single interval $[a, b]$. (This property also holds for outerplanar digraphs and also in the case where the names in clockwise order comprise a constant number of consecutive sequences.) 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. The list is a rotated list and can be searched using a modified binary search. If the graph is not outerplanar, then $S(v, w)$ can consist of more than one subinterval. A compact routing table will then have an entry for each of the subintervals contained in an edge label at v. It can be shown (see [9]) that the total size of a compact routing table T can be made $O(qn)$ by constructing T through a decomposition of a planar digraph into outerplanar subgraphs based on a face-on-vertex covering of cardinality q.

We now use graph embeddings to describe a decomposition of a planar digraph into subgraphs, called *hammocks*, each of which is outerplanar and shares at most four vertices with all other subgraphs in the decompositon. To generate the decomposition, we first convert the embedding $\hat{G}$ of G into an embedded undirected planar graph $\hat{G}_1$ without parallel edges. If $\mathscr{C}_1$ is the face-on-vertex covering of $\hat{G}_1$ that results out of the face-on-vertex covering $\mathscr{C}$ of $\hat{G}$, then all faces in $\hat{G}_1$ (except $\mathscr{C}_1$) are triangulated. Also, the faces in $\mathscr{C}_1$ do not share vertices and the boundary of each face is a simple cycle. Such a graph $\hat{G}_1$ is called *neatly prepared.*

We now group the faces of $\hat{G}_1$ together by using two operations: *absorption* and *sequencing* (see details in [11]). Initially mark all edges bordering the faces belonging to $\mathscr{C}_1$. Let f_1, f_2 be two faces (not in $\mathscr{C}_1$) 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.

Once absorption is no longer possible, identify maximal sequences of faces such that each face in the sequence has a marked edge and each pair of consecutive faces have an edge in common. Expand the faces that were absorbed into faces in the sequence. Each subgraph resulting from expanding such a sequence is called a *(major) hammock.* The two vertices at each end of the hammock are called *vertices of attachment.* Any edge not included in a major hammock induces a *(minor) hammock.* The set of all hammocks is a *hammock decomposition* of $\hat{G}$. In the sequel, $d(v, w)$ will denote the shortest distance from v to w.

3. Shortest paths in outerplanar digraphs.

Let us assume that $G = (V, E)$ is an outerplanar weighted digraph with real edge costs but no negative cycles. Let us also assume that a planar embedding $\hat{G}$ of G is provided, which means that for each vertex v there is a list of its neighbors in clockwise ordering around v. We will show how to compute efficiently (in parallel) the labels $S(v, w)$ for each edge $\langle v, w\rangle$. To simplify the discussion we assume that G is *nice*, i.e. (i) vertices are named in clockwise order around the exterior face, (ii) the undirected version of G is biconnected, (iii) $\forall\langle v, w\rangle \in E$, the edge $\langle w, v\rangle \in E$, and (iv) edge costs follow the generalized triangle inequality (i.e. $\langle v, w\rangle$ is the shortest path from v to w). (These assumptions can be removed as we shall see at the end of the section.)

It is easy to see that edge labelling is sufficient for finding any shortest path information. To store shortest paths or distances, with respect to a given vertex of a graph, to all other vertices it is appropriate to make use of trees. A tree is called *convergent (divergent)* if the edges of the tree point from a node to its parent (children).

LEMMA 1. *A convergent or divergent tree of shortest paths rooted at some vertex x of an n-vertex outerplanar digraph G can be constructed (given that edge labels are known) in* $O(\log n)$ *time using* $O(n/\log n)$ *processors on an EREW PRAM.*

PROOF. From the adjacency list representation of G we can obtain (optimally in $O(\log n)$ time on an EREW PRAM) its array representation as it is described in [21]. From this representation we can easily construct an array A such that each entry of A is associated with an edge $\langle v, w\rangle \in G$ and therefore with its corresponding label $S(v, w)$. Also for all edges incident on v, the corresponding edge labels will be consecutive elements of A and since the vertices are named consecutively around the exterior face, the array is essentially lexicographically sorted.

For the convergent tree case we first copy the array A into a new one, say $ST_c(x)$. Then (in $O(1)$ time with $O(n)$ processors) we execute the following: for each vertex $v \neq x$ such that $x \in S(v, w)$, mark $\langle v, w\rangle$ in $ST_c(x)$ as a tree edge. By doing a parallel prefix computation in $ST_c(x)$ we remove the edges not belonging to the tree and we have the desired convergent tree.

For the other case, do the following: reverse the direction of the edges of G, resulting into a new graph, say G^R. Find edge labelling information (S^R) for all edges in G^R (in the same way as it was done for G). Now, for each vertex $v \neq x$ such that $x \in S^R(v, w)$ (i.e. $\langle v, w\rangle \in G^R$), mark $\langle w, v\rangle$ as a tree edge. Again, a parallel prefix computation completes the construction. ■

In the following, with the term *face* we shall refer to an *internal face*.

ALGORITHM *OUT-LABEL*
BEGIN
Phase 1 (preprocessing):
(1) Construct a counterclockwise list of edges in each face using the adjacency list of each vertex (and the embedding information). (In this construction one of the faces will be the exterior. But this face can easily be recognized since it will contain all the vertices of G.)

(2) Assign a unique name to each face using the pointer doubling technique. (Just assign a processor to each edge and take the smallest processor id.)
(3) For each face, let x be the minimum numbered vertex and x' the maximum numbered one. Then the pair of edges $\langle x, x' \rangle$ and $\langle x', x \rangle$ are called the associated pair of edges of that face. For every face find its associated pair of edges using pointer doubling.
(4) Find the clockwise and counterclockwise distances from x to every vertex in the same face, again using pointer doubling in the list of edges of the face.
(5) For each face f_i, create a node. For each associated edge $\langle x, x' \rangle$ create a pointer to the face f_j containing $\langle x', x \rangle$. A *tree* is thus constructed.
(6) Order the tree edges (by sorting) and convert the tree of faces into a binary one if necessary. (This is needed for the parallel tree-contraction technique in phase 2, see e.g. [1].)
(7) For each face, compute the edge labelling information (inside the face) as follows: for each vertex v, find a vertex z such that the counterclockwise distance from v to z (in the face) is less than or equal to the clockwise distance from v to z, and z is the farthest such vertex in counterclockwise order. This can be done using binary search. The distances can be easily computed (in $O(1)$ time) using the distance information from x to every other vertex in the face, as well as the counterclockwise distance of the face (computed in step 4). Then, $S(v, w_1) = \{u : u \in [z, w_1]\}$ (where w_1 is the counterclockwise neighbour of v). Similarly, for the clockwise neighbour w_2, we have $S(v, w_2) = \{u : u \in [w_2, z)\}$ (see figure 3.1). Note that with these assignments to the labels, we enforce a tie-breaking rule which guarantees that a vertex u will be in only one set $S(v, w)$, $\forall v$.

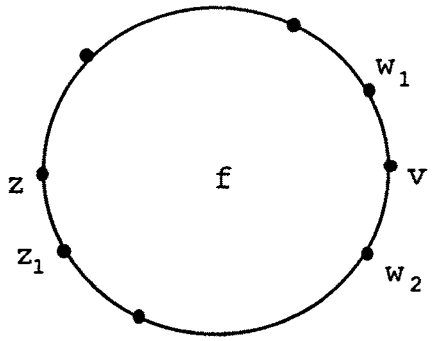

Figure 3.1

Phase 2:
(8) Perform a tree-contraction (in the tree of faces) (for more details about the tree-contraction technique see e.g. [1, 19]) using the *SHUNT* operation as follows (see figure 3.2.): Suppose that each l_x or f_x is either a simple face or an outerplanar graph (provided by unioning simple faces). Also assume that we can find edge labelling information between two outerplanar graphs that share an edge. Suppose l_j is performing a *SHUNT*. Then we can find edge labelling information between l_j, f_j

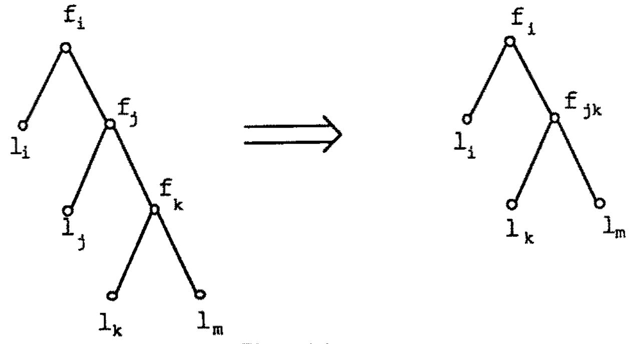

Figure 3.2

and then with f_k, and substitute these three nodes of the tree with one (f_{jk}) containing edge labelling information among vertices in l_j, f_j, f_k. END

COMMENT. Note that although (in general) the vertices inside a face f are not named consecutively in clockwise order (around f), the fact that $S(v, w_1) = \{u : u \in [z, w_1]\}$ and $S(v, w_2) = \{u : u \in [w_2, z)\}$ is correct (since e.g. any shortest path from v to every vertex in $[z, w_1]$ will definitely have $\langle v, w_1 \rangle$ as its first edge). The problem is how to augment these labels with vertices belonging to $[z_1, z)$ where z_1 is the immediate predecessor of z (in clockwise order) in f (see figure 3.1). For this reason we can use the tree structure of the internal faces and do the parallel tree-contraction. Therefore, the problem is concentrated on the computation of edge labelling information between two faces or outerplanar graphs G_1, G_2 that share an edge, provided that edge labelling information in each G_i $(i = 1, 2)$ is given.

Suppose that we are given two outerplanar graphs G_1 and G_2 that share an edge. We want to find shortest path information from every vertex of each G_i to the vertices of the other (provided that shortest path information in each G_i is given). We only describe how to find edge labelling information from each vertex of G_1 to vertices of G_2. Let v be a vertex of G_1 and u be a vertex of G_2. Let $\langle x, x' \rangle$ be the common edge between G_1 and G_2. The shortest path from v to u is through x iff either $d(v, x) + d(x, u) \leq d(v, x') + d(x', u)$ or $d(v, x) - d(v, x') \leq d(x', u) - d(x, u)$ $(*)$.

We shall give some technical lemmas and then describe how edge labelling information is generated. The following lemma is called the *monotonicity property* in [11].

LEMMA 2. [11] *Let x, y be vertices and f a face in an embedded planar graph. Define $h_{xy}(v) = d(v, x) - d(v, y)$, $v \in f$. Let v_1, $v_2 \in f$ such that $h_{xy}()$ achieves a minimum and a maximum respectively, over all $v \in f$. Then $h_{xy}()$ is nondecreasing on the clockwise sequence of vertices of f from v_1 to v_2, and nonincreasing on the clockwise sequence of vertices of f from v_2 to v_1. If $x \in f$, then $h_{xy}()$ realizes a minimum at x, and if $y \in f$, $h_{xy}()$ realizes a maximum at y.*

LEMMA 3. *Let $\hat{G}$ be an embedded outerplanar graph. Let x, y be endpoints of an edge. Then the function $h_{xy}(v) = d(v, x) - d(v, y)$ i) has a minimum (maximum) at $x(y)$, and ii) is non-decreasing as v moves clockwise from x to y.*

PROOF. Comes easily from lemma 2. ■

A related function is defined as: $\bar{h}_{xy}(v) = d(x, v) - d(y, v)$. By reversing the direction of all edges in the graph and then applying lemma 3, $\bar{h}_{xy}(v)$ also possesses the monotonicity property.

Now we return to the generation of edge labelling information from each vertex of G_1 to vertices of G_2. The functions $h_{xx'}(v)$ and $\bar{h}_{x'x}(u)$ $(v \in G_1, u \in G_2)$ can be found: i) by constructing (for x and x') convergent trees for G_1 and divergent trees for G_2 and ii) by using known parallel tree traversing techniques (Euler tour or tree contraction, see e.g. [19]). Each $v \in G_1$ now uses (*) and a binary search on the vertices of G_2 in order to determine how $S(v, w)$ will be augmented. The binary search can be done because of lemma 3. Note that only two edge label sets will be updated for each $v \in G_1$ (since x, or x', belongs to just one set). Let $x \in S(v, w_1)$ and $x' \in S(v, w_2)$. Then $S(v, w_1)$ will be augmented with the set of vertices satisfying (*). It is clear that this set comprises a contiguous portion of vertices in G_2 along its exterior face. The vertices in G_2 that do not satisfy (*) will belong to $S(v, w_2)$.

LEMMA 4. *Shortest path information between two outerplanar graphs G_1 and G_2, of n_1 and n_2 vertices respectively, that share an edge, can be found in $O(\log(n_1 + n_2))$ time by using $O(n_1 + n_2)$ processors on a CREW PRAM, provided that the shortest path information within each G_i, $i = 1, 2$ is given.*

PROOF. Correctness of the approach comes from inequality (*) and lemma 3. Lemma 3 also guarantees that each vertex of $G_1(G_2)$ needs only a binary search in the vertices of $G_2(G_1)$ (according to the values of $h_{xx'}()$, $\bar{h}_{x'x}()$) in order to find the farthest (in clockwise order) vertex u such that (*) is satisfied. The resource bounds come from: i) the construction of the four trees (at x and x'), ii) the parallel traversing of these trees, and iii) each processor associated with a vertex in $G_1(G_2)$ needs $O(\log n_2)$ $(O(\log n_1))$ time for binary search. It is clear that edge labels can be augmented in $O(1)$ time. ■

We now describe what we do in the case where G is not nice. (i) If the undirected version of G is biconnected, then vertices can be named easily in clockwise order around the exterior face using pointer doubling. Otherwise we do the following. Run the parallel biconnectivity algorithm of [16]. For every cutpoint v having degree $d \geq 3$, substitute it with a d-gon, i.e. replace v with new vertices $v_1, \ldots, v_d$, add edges $\{\{v_i, v_{(i+1) \bmod d}\} \mid 1 \leq i \leq d\}$ and replace edges $\{\{w_i, v\} \mid 1 \leq i \leq d\}$ with $\{\{w_i, v_i\} \mid 1 \leq i \leq d\}$. Find the list of vertices belonging to the exterior face. Now, each cutpoint appears twice in the list. Run the parallel list ranking algorithm of [4] to this list. If v appears twice in the list, then take the copy of the higher rank and set

the cost of the link $u \rightarrow v$ (where u is the predecessor of v in the list) equal to zero. Run again the parallel list ranking algorithm to this list resulting in the correct clockwise naming of the vertices of G around the exterior face. Finally, remove all edges added during this step.
(ii) If the graph is not biconnected, then for each v such that there is no edge $\langle v, w\rangle$ to w which follows v numerically, we add the edge $\langle v, w\rangle$ to G with cost ∞.
(iii) $\forall\langle v, w\rangle \in E$; if the edge $\langle w, v\rangle$ does not exist, then we increase it with cost ∞.
(iv) If an edge violates the generalized triangle inequality then we change its cost to be the shortest distance from its tail to its head, and mark it as a "dummy-edge". The original edge labels can be obtained by unioning the edge label on each "dummy-edge" into the label of the first edge in the shortest path from the tail to its head. The label of the "dummy-edge" is set to the empty set. The edges that violate the generalized triangle inequality can be identified (and the appropriate updating of their costs can be achieved) by using a recursive tree contraction technique in the tree of faces similar to that of step (8) (for details see [23]).

THEOREM 1. *Given an n-vertex outerplanar digraph with real edge costs but no negative cycles, all pairs shortest path information can be computed in $O(\log^2 n)$ time, using $O(n)$ processors on a CREW PRAM.*

PROOF. Correctness of the *OUT-LABEL* algorithm comes from lemma 4. It is clear that phase 1 of *OUT-LABEL* needs $O(n)$ processors and $O(\log n)$ time. (These resource bounds come from pointer doubling in steps (2) through (5), sorting in step (6) and the computation of initial values for the edge labels in each face in step (7). Note that step (1) needs $O(1)$ time and $O(n)$ processors.) Phase 2 is essentially a tree contraction which means that we need $O(\log n)$ iterations and since each iteration (by lemma 4) takes $O(\log n)$ time, we have a total of $O(\log^2 n)$ time using $O(n)$ processors. The resource bounds on a CREW PRAM for converting an outerplanar graph G into a nice one is as follows: step (i) needs $O(\log n \log^* n)$ time and $O(n)$ processors, from the results of [4, 16]. Steps (ii) and (iii) need $O(1)$ time and $O(n)$ processors. Step (iv) which can be achieved by a similar tree contraction technique as that of step (8), needs $O(\log n)$ time and $O(n)$ processors (because here we have $O(\log n)$ iterations and each iteration but the first needs $O(1)$ time; the initial step needs $O(\log n)$ time). ■

4. The general case of a planar digraph.

Our technique to determine the labels of all the edges of a planar digraph (whose embedding $\hat{G}$ and a face-on-vertex covering of cardinality q are given) consists of the following major steps:

ALGORITHM *LABEL*
(1) Decompose the embedded digraph $\hat{G}$ into $O(q)$ hammocks.

(2) Run the outerplanar algorithm in each hammock.
(3) Compress each hammock into an $O(1)$-sized outerplanar graph and then the embedded digraph $\hat{G}$ into a planar digraph $C(\hat{G})$ of $O(q)$ nodes.
(4) Run the matrix powering algorithm in $C(\hat{G})$ to find all pairs shortest paths between any pair of attachment vertices.
(5) For each pair of hammocks, find shortest path information for each vertex in one hammock to all vertices in the other hammock.
(6) For each hammock, find shortest path information between vertices in the same hammock. (* This is needed since a shortest path between two vertices in a hammock may leave and reenter the hammock.*)

THEOREM 2. *Given an n-vertex embedded planar digraph G with a face-on-vertex covering of cardinality q, we can generate all pairs shortest path information in* $O(\log^2 n)$ *time using* $O(nq + M(q))$ *processors on a CREW PRAM.*

PROOF. In the next sections we will analyze the steps of the above algorithm. Theorems 3 and 1 and lemmas 7, 8, 12 (corresponding to the steps (1), (2), (3), (5) and (6) respectively) will prove that our method needs (in total) a parallel time of $O(\log^2 n)$ and $O(nq + M(q))$ CREW PRAM processors. ■

5. Parallel decomposition of a planar digraph into hammocks.

In this section we discuss the decomposition of a planar digraph G into special subgraphs called hammocks. Let $\hat{G}$ be the embedding of G with a face-on-vertex covering $\mathscr{C}$, $|\mathscr{C}| = q$, $q > 1$. To generate the decomposition, we first convert $\hat{G}$ into a neatly prepared graph $\hat{G}_1$ with a face-on-vertex covering $\mathscr{C}'$, $|\mathscr{C}'| = q$. Then we identify hammocks in $\hat{G}_1$ and decompose $\hat{G}_1$ into $O(q)$ hammocks. Now, once we have made the decomposition we have to return back to the initial graph $\hat{G}$. This can be done as follows. Delete any edges from the hammocks that may be added during the construction of $\hat{G}_1$. Note that if a minor hammock looses its single edge it can be deleted. Finally replace the undirected edges by the corresponding directed ones.

Let $\hat{G}$ be an embedded planar digraph with set of faces F and a face-on-vertex covering $\mathscr{C}$, with $|\mathscr{C}| = q, (q > 1)$. Following the definitions of section 2, we first have to produce the *neatly prepared graph* $\hat{G}_1$ with face-on-vertex covering $\mathscr{C}'$ with the following properties: (i) it has no parallel edges; (ii) the boundary of each face is a simple cycle; (iii) no pair of faces of $\mathscr{C}'$ share a vertex; (iv) all faces of $\hat{G}_1$, except $\mathscr{C}'$, are triangulated.

We assume that each face of $\mathscr{C}$ is given as a list of the vertices of the face in a counterclockwise ordering around it. We do pointer doubling in the list of edges of each face f in order to assign a unique name to f (e.g. the highest id of the processors associated to it). After that, each face of $\mathscr{C}$ can be given a distinct number in $\{1, \ldots, q\}$ by sorting. We then remove directions and duplicate edges in parallel. The detection of a vertex v that appears more than once in a clockwise walk around a face f is done as follows. We again use pointer doubling to find the rank of each vertex in the clockwise list of f. Obviously, the vertices we are looking for are the vertices that have more than one rank. Then, for all but one occurence of v in the clockwise list of f do the following: if v_1, v_2 are the predecessor and successor of v respectively (in the clockwise walk around f), add the edge $\{v_1, v_2\}$ to the graph.

Shared vertices among at least two faces of $\mathscr{C}'$, produced from $\mathscr{C}$ in this way, are assigned to just one face (that with the smallest id among those sharing the vertex) and their appearances in any other face f_i are marked. Find a 2-ruling set U (see [4]) in the set of vertices of each face f_i. (A subset U of a set V is a 2-ruling set if no two vertices of U are adjacent in V and for each vertex v in V there is a path from v to some vertex of U whose edge length is at most 2.) If v is a marked vertex which belongs to U then we add the edge $\{v_1, v_2\}$ to the graph and delete v from f_i, where v_1, v_2 are the preceding and succeeding vertices of v in f_i. If f_i contains more than four vertices repeat the above steps. Clearly, after $O(\log n)$ steps each face will either have no shared vertices (associated with it), or have only four vertices some of which are shared with other faces. In the latter case, we choose any shared vertex v, add $\{v_1, v_2\}$ to the graph and delete v from the list of f_i. Now, if there is still a vertex shared by two faces f_i, f_j, replace v by v_i, v_j in the lists of f_i, f_j respectively, and add the edge $\{v_i, v_j\}$ to the graph.

In the sequel, we name the vertices in clockwise order around each face of $\mathscr{C}'$ by sorting the vertices according to face number. The triangulation of a face ϕ not in $\mathscr{C}'$ can be easily achieved by using the vertex numbers as follows: If v is the vertex of ϕ with the smallest id then we add an edge from each other vertex (of ϕ) to v.

LEMMA 5. *The conversion of the digraph $\hat{G}$ with face-on-vertex covering $\mathscr{C}$ ($|\mathscr{C}| = q$) into the neatly prepared graph $\hat{G}_1$ with set of faces F_1 and a face-on-vertex covering $\mathscr{C}'$ ($|\mathscr{C}'| = q$) can be done in $O(\log n \log^* n)$ time with $O(n)$ processors in a CREW PRAM.*

PROOF. Clearly all steps of the conversion but one can be done in $O(\log n)$ time and $O(n)$ processors, since they only use pointer doubling and/or sorting. The only step which needs $O(\log n \log^* n)$ time is the one which removes the shared vertices among two or more faces of $\mathscr{C}$. This step needs $O(\log n)$ iterations, where each iteration takes $O(\log^* n)$ time since we have to find a 2-ruling set in each face. ■

5.1. *Decomposing $\hat{G}_1$ into hammocks.*

We now show how to efficiently parallelize the decomposition of $\hat{G}_1$ with face-on-vertex covering $\mathscr{C}'$ into hammocks. We suppose that $|\mathscr{C}'| > 2$; the case $|\mathscr{C}'| = 2$ is discussed at the end of the section. The decomposition is done in two phases. In the first phase, corresponding to the absorption procedure, the faces that will be absorbed are identified and also some of the hammocks are constructed. In the second phase, corresponding to the sequencing procedure, we construct the remaining hammocks.

As in [11] we associate with each face ϕ_1 of the set of faces F_1 of $\hat{G}_1$ (except for the faces in $\mathscr{C}'$) a data structure consisting of two fields. The *first field* is a 4-tuple (i, j, k, r). The values (i, j, k) are the names of the faces in $\mathscr{C}'$ that contain the three vertices of ϕ_1 (ordered in such a way that $k = j$ implies $i = j$). The value r is the number of edges of ϕ_1 in common with the faces of $\mathscr{C}'$. We call this first field the *label of the face*. The *second field* is a triple (w_i, w_j, w_k) where w_i, w_j, w_k are the names of the vertices of ϕ_1 belonging to faces f_i, f_j, f_k, respectively.

5.1.1. *Absorption procedure.*

During this procedure, we construct a *forest of faces* of $F_1 - \mathscr{C}'$. The trees of the forest help us to specify: (a) The groups of the faces that will be absorbed one into the other. This grouping facilitates the handling of faces in $F_1 - \mathscr{C}'$ during the second phase of the construction of the hammocks; and (b) the groups of faces in $F_1 - \mathscr{C}'$ that constitute some of the hammocks. Each such group of faces is associated with another 4-tuple consisting of the names of four vertices of the hammock that may be shared by one or more other hammocks. These vertices are called the *attachment* vertices of the hammock. Since hammocks must be outerplanar subgraphs, we have to prevent the violation of the outerplanarity condition. So, when one attachment vertex is shared by two (or more) hammocks it must be split into two (or more) vertices.

The forest is constructed as follows: we create a node for each $\phi_k \in F_1 - \mathscr{C}'$ with first field value either (i, i, i, r) with $r = 0, 1, 2$, or $(i, i, j, 0)$ with $j \neq i$ for some faces f_i, f_j in $\mathscr{C}'$. For each $f_i \in \mathscr{C}'$, if a ϕ_k in $F_1 - \mathscr{C}'$ (with first field (i, i, i, r) or $(i, i, j, 0)$ with $j \neq i$) has an edge in common with a face ϕ_l in $F_1 - \mathscr{C}'$ (with first field (i, i, i, r) or $(i, i, j', 0)$ for any $f_{j'}$ with $j' \neq i$) then we add the edge $\{\phi_k, \phi_l\}$ to the forest. Some trees are thus constructed. They are binary, unrooted and unordered. We then run a parallel connected components algorithm (see [16]) to identify each tree in the forest. By sorting the array of face numbers by component name, we get all the faces belonging to the same tree to be consecutive in the array.

We next select a face in each tree to be the root of that tree. The selection is done as follows. Suppose T_{il} is a tree with faces labelled as (i, i, i, r), $r = 0, 1, 2$ or $(i, i, j, 0)$, $j \neq i$, For each vertex v_{ik} of $f_i \in \mathscr{C}'$ belonging also to a face ϕ of T_{il}, construct a list A_{ik}. A_{ik} consists of the faces ϕ ($\phi \in T_{il}$) to which v_{ik} belongs in clockwise ordering. Construct also a second list B_{il} consisting of the vertices v_{ik} of f_i ($f_i \in \mathscr{C}'$) that also belong to some face of T_{il}. A vertex v_{ik_2} follows v_{ik_1} in B_{il} if: the last face ϕ of the list A_{ik_1} has label (i, i, i, r), $r = 1, 2$ and v_{ik_2} is v_{ik_1}'s next vertex in the clockwise ordering of the vertices of f_i, or the last face ϕ of the list A_{ik_1} has label $(i, i, j, 0)$, $j \neq i$ and the edge $\{v_{ik_1}, v_{ik_2}\}$ is an edge of ϕ.

The root of T_{il} is the face ϕ that satisfies the following two conditions: 1. It has at least one edge that is not shared with another face ϕ' of T_{il} or with the face $f_i \in \mathscr{C}'$ and 2. (a) Suppose that the face ϕ has label $(i, i, j, 0)$, $j \neq i$ and v_{i_1}, v_{i_2} are its two vertices on f_i and also v_{i_2} follows (not necessarily immediately) v_{i_1} in the list B_{il}. Then, ϕ must be the first face in the list A_{i_1} and the last face in A_{i_2}; or (b) Suppose that the face ϕ has label (i, i, i, r), $r = 0, 1, 2$ and $v_{i_1}, v_{i_2}, v_{i_3}$ are its vertices. Suppose also that v_{i_2} follows v_{i_1} and v_{i_3} follows v_{i_2} in the list B_{il}. Then, ϕ must be the first face in the list A_{i_1} and the last face in the list A_{i_3}.

Once each tree is rooted, it can be ordered through parallel sorting. Then, by using the Euler tour technique (see e.g. [19]) we can establish parent-child relationships between all faces (nodes) of the tree, efficiently in parallel.

We shall now discuss the grouping of faces which help us to identify the hammocks. Several cases have to be considered according to the values of the first field of the root of the tree and the corresponding values in the other nodes.

Case 1: The label of the root face ϕ_r is $(i, i, j, 0)$, $j \neq i$. We have the following subcases.

Case 1.a: The tree has one or more faces (except for the root) labelled $(i, i, j', 0)$, $j' \neq j \neq i$ which, in fact, are leaves of the tree (see figure 5.1). In this case we proceed

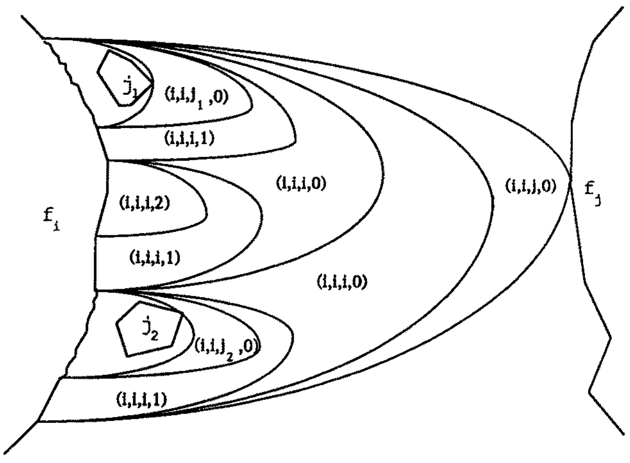

Figure 5.1

as follows. We do a preorder traversal of the tree and each processor associated with a face labelled $(i, i, j', 0)$ writes its preorder number in an array. (The array's element, where the processor writes, is determined by the preorder number of the face.) Using prefix computation, each face can find its next face in the array (if it exists). For every two faces ϕ_1, ϕ_2, that are neighbours in the above array, we find their lowest common ancestor in the tree and give to this lowest common ancestor another label being the tuple (ϕ_1, ϕ_2).

For each face ϕ_k labelled $(i, i, j', 0)$ consider its nearest in the tree face with label (ϕ_x, ϕ_y) (if it exists). Suppose that ϕ_k belongs to the left subtree of the tree rooted at (ϕ_x, ϕ_y). Suppose also that this subtree is rooted at the face $\phi_{x'}$ (see figure 5.2). The

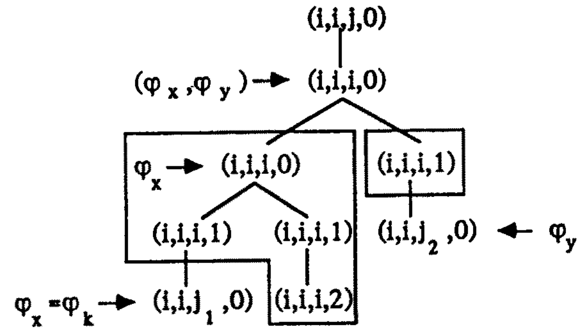

Figure 5.2: The tree corresponding to Figure 5.1. The faces included in the two polygons constitute the two hammocks.

faces belonging to the above subtree (except for the face ϕ_k) constitute a hammock. Associate all these faces with a 4-tuple consisting of the attachment vertices of the hammock. This is easily done as follows: find the size (number of faces) of the subtree; initialize an array of the above size and update it with the faces of the subtree (according to their preorder numbering).

The attachment vertices are determined as follows: Consider the face $\phi_{x'}$ (as it is defined above). Since its label is of the form (i, i, i, r), $r = 0, 1$, its three vertices $v_{x'_1}$, $v_{x'_2}$, $v_{x'_3}$ belong to the face f_i in $\mathscr{C}'$. The two (of the four) attachment vertices of the hammock are the vertices $v_{x'_i}, v_{x'_j}$ $(i, j = 1, 2, 3)$ such that the edge $\{v_{x'_i}, v_{x'_j}\}$ of $\phi_{x'}$ is not shared by another face of the hammock or by the face $f_i \in \mathscr{C}'$. The other two attachment vertices of the hammock are the vertices of ϕ_k that belong to f_i. In the case that ϕ_k has not an ancestor with label (ϕ_x, ϕ_y) (that is, ϕ_k is the only face labelled $(i, i, j', 0)$ except ϕ_r), all the faces of the tree (except ϕ_r and ϕ_k) constitute a hammock. The attachment vertices of the hammock are determined as above assuming that $\phi_{x'}$ is the child face of ϕ_r.

Now, delete from the tree the subtrees that constitute the already identified hammocks and the faces with label of the form (ϕ_i, ϕ_j). As a consequence, the tree may be now disconnected. Each resulting subtree constitutes a new hammock, except for the subtree having as root the face ϕ_r. In this subtree all of its faces (except for the face ϕ_r) constitute a new hammock. The attachment vertices of each new hammock H, identified by the previous operations are determined as follows. Consider a vertex v of f_i in $\mathscr{C}'$ that also belongs to a face ϕ of H. The vertex v is an attachment vertex of H if and only if its previous vertex in the clockwise ordering of the vertices of f_i, or its next vertex or both do not belong to a face ϕ' of H. It is clear, from the construction of the hammocks, that there are at most four attachment vertices in H.

Case 1.b: The tree has no other face (except for the root) with label $(i, i, j', 0)$, $j' \neq j \neq i$. In this case we associate the tree with one new face, whose first field is the 4-tuple $(i, i, j, 1)$ and second field is the same as the second field of the root face ϕ_r. We marked as "deleted" all the faces of the tree.

Case 2: The label of the root face ϕ_r is (i, i, i, r) where $r = 0, 1, 2$. The subcases and their solutions are similar to those of case 1.

5.1.2. *Sequencing procedure and hammock construction.*

This second procedure efficiently parallelizes the sequencing technique and constructs hammocks that were not yet constructed. We create a node for each face with label $(i, i, j, 1)$ or $(j, j, i, 1)$ (for some indices i, j corresponding to faces f_i, f_j in $\mathscr{C}'$). We also create a node for each new face resulting from the absorption operation. If two faces ϕ_1, ϕ_2 have one edge in common, we add the edge $\{\phi_1, \phi_2\}$. The connected components of this new graph is: (i) a sequence of faces in $F_1 - \mathscr{C}'$ between two faces f_i, f_j of $\mathscr{C}'$, or (ii) an isolated face ϕ in $F_1 - \mathscr{C}'$ between two faces of $\mathscr{C}'$ (the face ϕ may be the result of an absorption operation).

Each of these components include the faces that constitute a hammock. In order to determine the attachment of these hammocks we proceed as follows. For each component (with more than one face), we consider the nodes (faces in $F_1 - \mathscr{C}'$) having degree one. It is clear that we have two such faces and the attachment vertices of the hammock are vertices of these faces. Suppose that ϕ_1, $\phi_2 \in F_1 - \mathscr{C}'$ are two faces of the above type and their vertices belong to faces $f_i, f_j \in \mathscr{C}'$. We distinguish among the following cases.

1. The two faces ϕ_1, ϕ_2 have label $(i, i, j, 1)$ and their vertices are included in the triples $(v_{j_1}, v_{i_1}, v_{i_x})$, $(v_{j_2}, v_{i_2}, v_{i_y})$ respectively. In this case we take as attachment vertices: (i) the two vertices v_{j_1}, v_{j_2} of f_j, and (ii) the two vertices of f_i that are connected with the vertices v_{j_1}, v_{j_2} and also are not shared with another face of the hammock.

2. The two faces $\phi_1, \phi_2 \in F_1 - \mathscr{C}'$ have label $(i, i, j, 1)$ and $(j, j, i, 1)$ respectively. In this case the attachment vertices are: (i) the vertex of ϕ_1 belonging to f_j; if this vertex is the same as one of the vertices of ϕ_2 that belongs to f_j, split this vertex into two vertices; (ii) the vertex of ϕ_2 belonging to f_i; if this vertex is common with one of the vertices of ϕ_1 that belongs to f_i, split this vertex into two vertices; (iii) one of the two vertices of f_i (f_j) that satisfies the following: (a) it is not shared by another face of the hammock, and (b) the other vertex of f_i (f_j) is not split (by a previous step).

For each component (hammock) that has only one face, we take as attachment all its three vertices. Finally, we return all the faces that were absorbed (marked "deleted") back to each hammock. Any edge that is not included in any hammock is considered individually as a (minor) hammock. Thus,

LEMMA 6. *Let $\hat{G}_1$ be an embedded, neatly prepared planar digraph with n vertices and a face-on-vertex covering of q faces. Then, $\hat{G}_1$ can be decomposed into $O(q)$ hammocks in $O(\log n \log^* n)$ time using $O(n)$ processors on a CREW PRAM.*

PROOF. The proof of the fact that there are $O(q)$ hammocks in the decomposition of $\hat{G}_1$ is similar to the one given in [11]. The time and processor bounds are dominated by the corresponding bounds of running a parallel connected components algorithm (see [16]), since all other steps (i.e. computing the lowest common ancestor, computing simple tree functions, ordering of the trees through sorting) can be done with $O(n)$ processors in $O(\log n)$ time. ■

If $|\mathscr{C}'| = 2$, we decompose $\hat{G}_1$ as follows. Suppose that f_1, f_2 are the two faces of $\mathscr{C}'$. Assign one processor to each edge that connects a vertex v_1 of f_1 with a vertex v_2 of f_2. Using prefix computation find the edge e associated with the processor with the smallest id. Split each one of the two endpoints of e into two vertices. Reconnect the vertices so that the faces f_1, f_2 are merged and the resulting graph is outerplanar. We consider the resulting graph as a hammock.

THEOREM 3. *Let $\hat{G}$ be an embedded planar digraph with n vertices and a face-on-vertex covering of cardinality q. Then, $\hat{G}$ can be decomposed into $O(q)$ hammocks in $O(\log n \log^* n)$ time using $O(n)$ CREW PRAM processors or in $O(\log n)$ time using $O(n)$ CRCW PRAM processors.*

PROOF. Immediate consequence of lemmas 5 and 6. For the CRCW implementation see [23]. ■

6. Further shortest path information.

In order to complete the description of the *LABEL* algorithm, it remains to show: (i) how a compressed version of a planar digraph is computed, and (ii) how shortest path information among vertices of the same hammock or between two hammocks can be computed, given that shortest distances between attachment vertices are known.

A compressed version of a planar embedded digraph $\hat{G}$ is computed by using the compressed version of a hammock in $\hat{G}$ and by adding in the minor hammocks. The generation of the compressed version of a hammock H is done using the idea of [11]. The parallel implementation is achieved by using the pointer doubling technique. Therefore we have the following lemma.

LEMMA 7. *Let H be a hammock, $h = |H|$, and let a_i, $1 \leq i \leq 4$, be its attachment vertices. The compressed version $C(H)$ of H is an outerplanar digraph of $O(1)$ size and can be constructed in $O(\log h)$ time by $O(h)$ processors on a CREW PRAM. Also for every pair a_i, a_j in H $(i \neq j)$, if a shortest path from a_i to a_j exists in H then there is one in $C(H)$ of equal cost.*

For the generation of shortest path information between two hammocks, given the shortest distances between their attachment vertices, we again follow the approach by Frederickson in [11]. The approach starts by computing some very basic information and proceed with information concerning constrained shortest paths. Then information about less constrained shortest paths can be computed, given information about more constrained shortest paths. Finally, information about unconstrained shortest paths is computed. It is not difficult to achieve the parallel implementation which involves combination of well-known parallel tree traversing techniques (see [19]), parallel sorting, parallel prefix computation and finding all pairs shortest paths information between two outerplanar digraphs (see section 3).

LEMMA 8. *Let H_1, H_2 be two hammocks of n_1, n_2 vertices repectively. If the shortest distances between their attachment vertices are known, then edge labelling information among vertices in H_1, H_2 is generated in $O(\log(n_1 + n_2))$ time by using $O(n_1 + n_2)$ processors on a CREW PRAM.*

6.1. *Shortest path information in one hammock.*

In this section we will show how edge labelling information in one hammock can be generated, even when the shortest paths between the vertices of the hammock leave and reenter the hammock. We assume that edge labelling information, as well as the distances between the attachment vertices of the hammock are given. Let a_i, $1 \leq i \leq 4$, be the attachment vertices of a hammock H with $a_1, a_2 \in f$ and $a_3, a_4 \in f'$ (f, f' are the two faces bounding H), and a_1 is adjacent to a_4 and a_2 is adjacent to a_3.

If the shortest paths leave and reenter the hammock through attachment vertices in the same end (i.e. through a_1, a_4 or a_2, a_3), then we simply augment the hammock with edges $\langle a_1, a_4 \rangle$, $\langle a_4, a_1 \rangle$, $\langle a_2, a_3 \rangle$, $\langle a_3, a_2 \rangle$ with costs equal to their shortest distances. Then we run the algorithm *OUT-LABEL.* If the shortest paths leave the hammock at one end and reenter at the other, it is enough to find $\forall v \in H$ a set $U_H(v) = \{u \mid SP(v, u; H)\}$, where $SP(v, u; H)$ denotes that the shortest path from v to u is contained in H. (In the following, $j \neq 5 - i$.) Let $U_{H_{ij}}(v) = \{u \mid d_H(v, u) \leq d_H(v, a_i) + d(a_i, a_j) + d_H(a_j, u)\}$ where $d_H(v, u)$ is the length of the shortest path from v to u that is constrained to stay in H. Then, $U_H(v) = \cap_{\forall i,j} U_{H_{ij}}(v)$ Thus, the problem is concentrated in the computation of the sets $U_{H_{ij}}(v)$, $\forall v \in H$ and $\forall i, j$. For if we have $\forall v \in H$ the set $U_H(v)$, then we update the labels of the edges incident to v as follows. If $u \notin U_{H_{13}}(v)$ or $u \notin U_{H_{24}}(v)$ or both (the other cases or combination of cases have similar solutions) then we take two copies of H, add edges $\langle a_2, a_4 \rangle$, $\langle a_4, a_2 \rangle$, $\langle a_1, a_3 \rangle$, $\langle a_3, a_1 \rangle$ with costs equal to their corresponding shortest distances and run on this new graph (which is obviously outerplanar) the algorithm *OUT-LABEL.* We update the compact routing table of v with u according to this last running of *OUT-LABEL.*

The above approach is used by Frederickson in [11]. In order to find the sets $U_{H_{ij}}(v)$, Frederickson performs a very clever search of the hammock determining the shortest distances between many pairs of selected vertices v and u. Since his method seems to be inherently sequential due to a special data structure used, we keep the basic notions almost the same and we follow a totally different method for computing the sets $U_{H_{ij}}(v)$.

The following two lemmas are very helpful in the computation of each $U_{H_{ij}}(v)$.

LEMMA 9. [11]. *Let H be a hammock and a_i, a_j attachment vertices at opposite ends. Let $v, u \in H, u \in f, f$ one of the two faces bounding H. If $u \in U_{H_{ij}}(v)$ then every vertex in f between u and the attachment vertex on f that is at the same end as a_i, belongs to $U_{H_{ij}}(v)$.*

LEMMA 10. [11]. *Let H be a hammock and a_i, a_j attachment vertices at opposite ends. Let $v, u \in H$, $v \in f$ (one of the two faces bounding H). If $u \notin U_{H_{ij}}(v)$, then $u \notin U_{H_{ij}}(v')$ for any $v' \in f$ between v and the attachment vertex on f that is at the same end as a_i.*

Assume that we have a hammock H as shown in figure 6.1 and that H is nice. (Recall the definition from section 3.) Assume also that $v \in [a_2, a_1]$, $u \in [a_4, a_3]$ (the

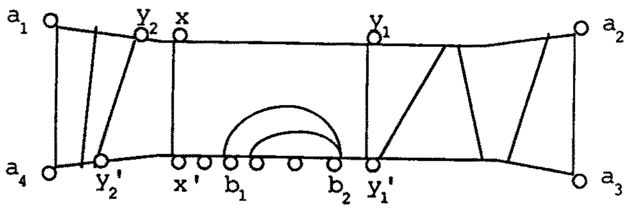

Figure 6.1

other cases are similar). We shall describe now, how $U_{H_{13}}(v)$ is computed. (Again, the other cases are similar.)

Construct a convergent tree of shortest paths rooted at a_1 and a divergent tree of shortest paths rooted at a_3. Perform a parallel tree traversing of the trees to compute the distances $d_H(w, a_1)$ and $d_H(a_3, w)$, $\forall w \in H$. From lemmas 9 and 10 it is sufficient to find $\forall v \in [a_2, a_1]$ the "farthest" (closest to a_3) u_v such that $d_H(v, u_v) \leq d_H(v, a_1) + d(a_1, a_3) + d_H(a_3, u_v)$.

Now consider H. We concentrate on faces f bounded by edges $\{x, x'\}$ and $\{y_1, y_1'\}$ such that $x, y_1 \in [a_2, a_1]$ and $x', y_1' \in [a_4, a_3]$ (with $\{x, x'\}$ we denote the pair $\langle x, x' \rangle$, $\langle x', x \rangle$ and the same holds for $\{y_1, y_1'\}$). Note that $x = y_1$ or $x' = y_1'$ in some cases. (See figure 6.1.)

REMARK. We assume that in such a face f we have done the necessary preprocessing in order to compute in $O(1)$ time the distances $d_H(x, a)$ and $d_H(x', a)$, for any vertex a belonging to f. (According to what we have discussed in section 3, this can be done for all such faces in $O(\log h)$ time using $O(h)$ processors, where $h = |H|$.) Also, it will be needed to handle vertices in intervals $[b_1, b_2]$ (see figure 6.1), i.e. vertices which do not belong to the face bounded by $\{x, x'\}$ and $\{y_1, y_1'\}$. For this, we do the following preprocessing for any such interval $[b_1, b_2]$: We construct divergent and convergent shortest path trees rooted at b_1, b_2 including *only* those vertices $c \in [b_1, b_2]$. Then we find the shortest distances $d_H(b_i, c)$ and $d_H(c, b_i)$ $(i = 1, 2)$ for any $c \in [b_1, b_2]$. This will take for the whole hammock $O(\log h)$ time using $O(h)$ processors.

Such faces bounded by $\{x, x'\}$ and $\{y_1, y'_1\}$ (as defined above) form a sequence (list). Thus, it is easy to construct a doubly-linked list where each node is a face of the above type. There are also two links per node: the *right* link that points to the adjacent face towards vertices a_2, a_3 and the *left* link pointing to the adjacent face towards vertices a_1, a_4. Consider an edge $\{x, x'\}$ (see figure 6.1). Assign to each such edge a processor. (We also assume that the processor is associated with x). Clearly, the total number of processors is $O(h)$, where $h = |H|$. Each processor examines if the following inequality holds:

$$d_H(x, x') \leq d_H(x, a_1) + d(a_1, a_3) + d_H(a_3, x') \quad (I\ 1).$$

If $(I\ 1)$ holds, then $u_x \in [x', a_3]$ and we associate the processor to the face bounded by $\{x, x'\}$ and $\{y_1, y'_1\}$. Otherwise, $u_x \in [a_4, x')$ and the processor is associated with the face bounded by $\{y_2, y'_2\}$ and $\{x, x'\}$. The above test is done in $O(1)$ time. If $u_x \in [x', a_3]$, we examine if it belongs to $[x', y'_1]$. Therefore, we test if $(I\ 2)$ holds

$$d_H(x, y'_1) \leq d_H(x, a_1) + d(a_1, a_3) + d_H(a_3, y'_1) \quad (I\ 2)$$

If it does not hold, then the desired u_x is determined by applying the following procedure, which we call *LBS* (local binary search). We probe to the "median" vertex $c \in [x', y'_1]$ (including those vertices belonging to an interval $[b_1, b_2]$). If $c \in [x', b_1]$ or $c \in [b_2, y'_1]$, the distance $d_H(x, c)$ is computed easily (in $O(1)$ time according to the previous remark). Otherwise, we have that $d_H(x, c) = \min\{d_H(x, b_1) + d_H(b_1, c),\ d_H(x, b_2) + d_H(b_2, c)\}$. Obviously, $d_H(x, c)$ can be computed in $O(1)$ time, since every distance is known a priori (recall the previous remark). Therefore again in $O(1)$ time we can test if $d_H(x, c) \leq d_H(x, a_1) + d(a_1, a_3) + d_H(a_3, c)$ holds. If it holds then we again probe to the "median" vertex of the interval $[c, y'_1]$; otherwise, we probe to the "median" of the interval $[x', c)$. Then we proceed similarly. It is clear that procedure *LBS* needs time logarithmic with respect to the size of the interval $[x', y'_1]$.

A similar test can be done in the case where $u_x \in [a_4, x')$ and in the case $u_x \in [y'_2, x')$ where a call to procedure *LBS* again provides the appropriate u_x. This is the easy case. The difficult case arises where $u_x \notin [y'_2, y'_1]$. In order to overcome the above difficulty we use the left and right links of a face (see figure 6.2). Suppose that $u \in (y'_1, a_3]$. (The other case is similar.) Then the processor associated with x (and also with face f_1) is looking at the face pointed by its right link (f_2). Then x can find its shortest distance to z' as follows: $d_H(x, z') = \min\{d_H(x, y_1) + d_H(y_1, z'),\ d_H(x, y'_1) + d_H(y'_1, z')\}$. Obviously this distance can be found in $O(1)$ time. Also using a similar expression we can find $d_H(x, z)$. If $u_x \in (y'_1, z']$ (or even in the easy case where $u_x \in [x', y'_1]$) we set the right link of the corresponding face to null. If $u_x \notin (y'_1, z']$ then we perform pointer doubling using the right links of the doubly-linked list. This means that the right link of f_1 will point to f_3 in figure 6.2.

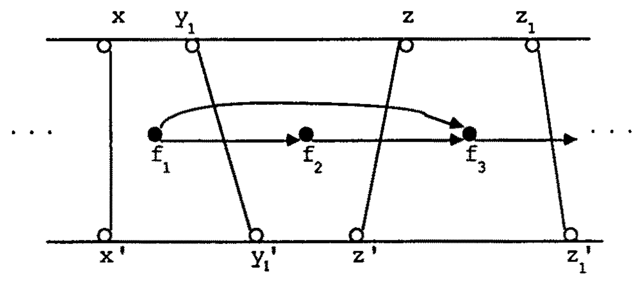

Figure 6.2

Then again in $O(1)$ time we can find the distances $d_H(x, z'_1)$ and $d_H(x, z_1)$, since for example $d_H(x, z'_1) = \min\{d_H(x, z') + d_H(z', z'_1),\ d_H(x, z) + d_H(z, z'_1)\}$. This process continues until some step i where a face f_k is found such that $u_x \in (x'_{i-1}, x'_i]$ (see figure 6.3).

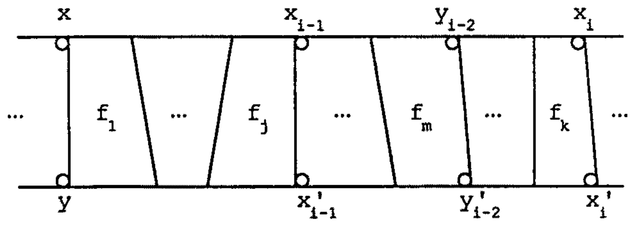

Figure 6.3

Let us assume that $\{x_{i-1}, x'_{i-1}\}$ bounds face f_j pointed by f_1 in step $i - 1$ and $\{y_{i-2}, y'_{i-2}\}$ bounds face f_m pointed by f_j in step $i - 2$. Then in order to find to which face the vertex u_x belongs, we shall perform a binary search in $(x'_{i-1}, x'_i]$ following the steps of f_j in reverse order. This means that we probe first on face f_m (pointed by f_j in step $i - 2$) and find distances $d_H(x, y_{i-2})$ and $d_H(x, y'_{i-2})$ as follows: $d_H(x, y'_{i-2}) = \min\{d_H(x, x_{i-1}) + d_H(x_{i-1}, y'_{i-2}), d_H(x, x'_{i-1}) + d_H(x'_{i-1}, y'_{i-2})\}$. (A similar expression holds for $d_H(x, y_{i-2})$.) Note that the four distances in the brackets have already been computed in steps $i - 2$ and $i - 1$. Then we test if the following inequality holds:

$$d_H(x, y'_{i-2}) \le d_H(x, a_1) + d(a_1, a_3) + d_H(a_3, y'_{i-2}) \quad (I\,3).$$

If $(I\,3)$ holds, then $u_x \in (y'_{i-2}, x'_i)$ and we perform a similar binary search using the steps of f_m now. Otherwise, $u_x \in (x'_{i-1}, y'_{i-2}]$ and the binary search follows the steps of f_j. The above is continued until we find a face that contains u_x between its boundary vertices in $[a_4, a_3]$.

It is clear that such a face is found after at most $O(\log h)$ steps. Then by calling procedure *LBS* in this face now, we can determine u_x exactly.

Finally, it remains to describe how a vertex v, $v \in [y, x]$ (see figure 6.4) can find its

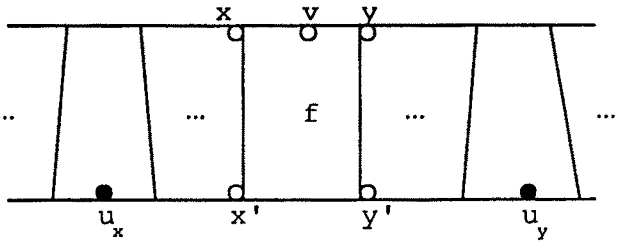

Figure 6.4

u_v given u_x and u_y. We first examine (in the known way) if $u_v \in [x', y']$. (Note that the distances $d_H(v, x')$ and $d_H(v, y')$ can be computed in $O(1)$ time after the preprocessing we have done in the hammock.) If not, suppose that $u_v \in (y', u_y)$. (The other case is similar.) Then we are looking again for a face containing u_v. This can be done using the same technique (as before) of "following in reverse order the steps of face f" in pointer doubling. It is also clear that in each step the distance $d_H(v, u)$, $u \in (y', u_y)$ and u is a boundary vertex, can be computed in $O(1)$ time. Therefore, after $O(\log h)$ steps we shall find a face in which u_v belongs. Then one call to procedure *LBS* will give us the exact u_v.

LEMMA 11. *The sets $U_{H_{ij}}(v)$, $j \neq 5 - i$, for each $v \in H$ can be found in $O(\log h)$ time using $O(h)$ processors on a CREW PRAM.*

PROOF. Comes from lemmas 9 and 10 and the above discussion. ■

LEMMA 12. *Let H be a hammock in an embedded planar digraph $\hat{G}$, and let $h = |H|$. Correct shortest path (edge labelling) information for the edges of each vertex of H (even when the shortest paths leave and reenter the hammock) can be generated in $O(\log^2 h)$ time, using $O(h)$ processors on a CREW PRAM, provided that the shortest distances among the attachment vertices of H are given.*

PROOF. By lemma 11 and the need for rerunning the *OUT-LABEL* algorithm. ■

ACKNOWLEDGEMENTS. We are grateful to G. Frederickson for his valuable comments and criticism, and also for pointing out the error of the recursion scheme in a previous version of this work. We also wish to thank the referees for careful reading of the paper and useful suggestions which improved the presentation.

REFERENCES

1. K. Abrahamson, N. Dadoun, D. Kirkpatrick and T. Przytycka, *A simple parallel tree contraction algorithm*, J. of Algorithms, 10 (1989), pp. 287–302.
2. D. Beinstock and C. 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 and P. Shor, *Efficient NC-algorithms for set cover with applications to learning and geometry*, Proc. 30th IEEE Symp. on FOCS, 1989, pp. 54–59.
4. R. Cole and U. Vishkin, *Appropriate parallel scheduling*. Part I: *The basic technique with applications to optimal parallel list ranking in logarithmic time*, SIAM J. Comp., Vol. 17, No. 1, February 1989, pp. 128–142.
5. E. W. Dijkstra, *A note on two problems in connection with graphs*, Numerische Mathematik, 1 (1959), pp. 275–323.
6. E. Dekel, D. Nassimi and S. Sahni, *Parallel matrix and graph algorithms*, SIAM J. Comp., Vol. 10, No. 4, Nov. 1981, pp. 657–675.
7. H. Djidjev, G. Pantziou and C. Zaroliagis, *Computing shortest paths and distances in planar graphs*, in Proc. 18th ICALP, 1991, LNCS, Vol. 510, pp. 327–339.
8. R. Floyd, *Algorithm 97: shortest path*, Comm. ACM, 5 (1962), pp. 345.
9. G. N. Frederickson and R. Janardan, *Designing networks with compact routing tables*, Algorithmica, 3 (1988), pp. 171–190.
10. G. N. Frederickson, *A new approach to all pairs shortest paths in planar graphs*, Proc. 19th ACM STOC, New York City, May 1987, pp. 19–28.
11. G. N. Frederickson, *Planar graph decomposition and all pairs shortest paths*, JACM, Vol. 38, No. 1, January 1991, pp. 162–204; also TR-89-015, ICSI, Berkely, March 1989.
12. M. Fredman, *New bounds on the complexity of the shortest path problem*, SIAM J. Comp., 5 (1976), pp. 83–89.
13. M. Fredman and R. Tarjan, *Fibonacci heaps and their uses in improved network optimization algorithms*, JACM, 34 (1987), pp. 596–615.
14. H. Gazit and G. Miller, *A parallel algorithm for finding a separator in planar graphs* in Proc. of the 28th IEEE Symp. FOCS, 1987, pp. 238–248.
15. A. Goldberg, S. Plotkin and G. Shannon, *Parallel symmetry-breaking in sparse graphs*, Proc. of the 19th ACM STOC, 1987, pp. 315–324.
16. T. Hagerup, *Optimal parallel algorithms for planar graphs*, Inform. and Computation, Vol. 84, 1990, pp. 71–96.
17. T. Hagerup, G. Pantziou and C. Zaroliagis, *Efficient sequential and parallel algorithms for planar digraph problems*, CTI Tech. Rep. TR-91.09.22, Sept. 1991. Submitted.
18. P. Klein and J. Reif, *An efficient parallel algorithm for planarity*, Proc. 27th Annual IEEE Symp. on FOCS, 1986, pp. 465–477.
19. R. Karp and V. Ramachandran, *A survey of parallel algorithms for shared-memory machines*, Rep. No. UCB/CSD 88/804, University of California, Berkely, 1989.
20. A. Lingas, *Efficient parallel algorithms for path problems in planar directed graphs*, Proc. SIGAL '90, Tokyo, LNCS, Springer Verlag.
21. G. Pantziou, P. Spirakis and C. Zaroliagis, *Optimal parallel algorithms for sparse graphs*, in Proc. of 16th Int'l Workshop on Graph-Theoretic Concepts in Computer Science (WG90), Berlin, 19–22 June, 1990, LNCS Vol. 484, pp. 1–17, Springer-Verlag.
22. G. Pantziou, P. Spirakis and C. Zaroliagis, *Efficient parallel algorithms for shortest paths in planar graphs*, Proc. of the 2nd Scand. Workshop on Algorithm Theory (SWAT90), Bergen, Norway, 11–14 July, 1990, LNCS, Vol. 447, pp. 288–300, Springer-Verlag.
23. G. Pantziou, P. Spirakis and C. Zaroliagis, *Efficient parallel algorithms for shortest paths in planar digraphs*, TR-91.07.21, Computer Technology Institute, Patras, July 1991.
24. J. van Leeuwen and R. Tan, *Computer networks with compact routing tables*, in The Book of L, G. Rozenberg and A. Salomaa (eds.), Springer-Verlag, NY (1986), pp. 259–273.