ELSEVIER Theoretical Computer Science 168 (1996) 121–154

Theoretical Computer Science

Hammock-on-ears decomposition: A technique for the efficient parallel solution of shortest paths and other problems[1]

Dimitris J. Kavvadias[a,b], Grammati E. Pantziou[c], Paul G. Spirakis[a,d], Christos D. Zaroliagis[e,*]

[a] *Computer Technology Institute, P.O. Box 1122, 26110 Patras, Greece*
[b] *Department of Mathematics, University of Patras, 26500 Patras, Greece*
[c] *Computer Science Department, University of Central Florida, Orlando, FL 32816, USA*
[d] *Department of Computer Engineering & Informatics, University of Patras, 26500 Patras, Greece*
[e] *Max-Planck-Institut für Informatik, Im Stadtwald, 66123 Saarbrücken, Germany*

Abstract

We show how to decompose efficiently in parallel any graph into a number, $\tilde{\gamma}$, of outerplanar subgraphs (called *hammocks*) satisfying certain separator properties. Our work combines and extends the sequential hammock decomposition technique introduced by Frederickson and the parallel ear decomposition technique, thus we call it the *hammock-on-ears decomposition*. We mention that hammock-on-ears decomposition also draws from techniques in computational geometry and that an embedding of the graph does not need to be provided with the input. We achieve this decomposition in $O(\log n \log\log n)$ time using $O(n+m)$ CREW PRAM processors, for an n-vertex, m-edge graph or digraph. The hammock-on-ears decomposition implies a general framework for solving graph problems efficiently. Its value is demonstrated by a variety of applications on a significant class of graphs, namely that of *sparse (di)graphs*. This class consists of all (di)graphs which have a $\tilde{\gamma}$ between 1 and $\Theta(n)$, and includes planar graphs and graphs with genus $o(n)$. We improve previous bounds for certain instances of shortest paths and related problems, in this class of graphs. These problems include all pairs shortest paths, all pairs reachability, and detection of a negative cycle.

1. Introduction

The efficient parallel solution of many problems often requires the invention and use of original, novel approaches radically different from those used to solve the same

* Corresponding author. E-mail: zaro@mpi-sb.mpg.de.

[1] This work was partially supported by the EU ESPRIT Basic Research Action No. 7141 (ALCOM II) and by the Ministry of Education of Greece. The work of the second author was also partially supported by the NSF postdoctoral fellowship No. CDA-9211155.

problems sequentially. Notorious examples are list ranking [9], connected components [5], etc.

In other cases, the novel parallel solution paradigm stems from a non-trivial parallelization of a specific sequential method, e.g. merge-sort for EREW PRAM optimal sorting [7]. This second paradigm is demonstrated in our paper. Specifically, we provide an efficient parallel algorithm for decomposing any graph into a set of outerplanar subgraphs (called *hammocks*). We call this technique the *hammock-on-ears decomposition*. As the name indicates, our technique is based on the sequential hammock decomposition method of Frederickson [16, 17] and on the well-known ear decomposition technique [31], and non-trivially extends our previous work for planar graphs [36] to any graph. We demonstrate its applicability by using it to improve the parallel (and in one case the sequential) bounds for a variety of problems in a significant class of graphs, namely that of *sparse (di)graphs*. This class consists of all n-vertex (di)graphs which can be decomposed into a number of hammocks, $\tilde{\gamma}$, ranging from 1 up to $\Theta(n)$ (or alternatively have an $O(n)$ number of edges). This class includes planar graphs and graphs with genus $o(n)$.

The hammock-on-ears decomposition (like the sequential hammock decomposition) decomposes any (di)graph G into a number of outerplanar subgraphs (the *hammocks*) that satisfy certain separator conditions. The decomposition has the following properties: (i) each hammock has at most *four* vertices in common with any other hammock (and therefore with the rest of the graph), called the *attachment vertices*; (ii) each edge of the graph belongs to exactly one hammock; and (iii) the number of hammocks produced is of the minimum possible order among all decompositions, and is bounded by a function involving certain topological measures of G (genus, or crosscap number).

We achieve this decomposition in two major phases. In the first phase, whose outcome are outerplanar portions of the input (di)graph G (called outerplanar outgrowths), we transform an initial arbitrary ear decomposition into a new decomposition of G into paths. These paths include with certainty the outerplanar outgrowths. Then, by employing techniques from parallel computational geometry, we identify in each path the outerplanar outgrowths if they exist. In the second phase, we identify the hammocks by using the output of phase one and by performing some local degree tests.

This decomposition allows us to partially reduce the solution of a given problem Π on G, to the solution of Π in an outerplanar graph. The *general scheme* for solving problems using the hammock-on-ears decomposition technique consists of the following major steps:

1. Find a hammock-on-ears decomposition of the input (di)graph G, into a set of $\tilde{\gamma}$ hammocks.
2. Solve the given problem Π in each hammock separately.
3. Generate a compressed version of G of size $O(\tilde{\gamma})$, and solve Π in this compressed (di)graph using an alternative method.
4. Combine the information computed in steps 2 and 3 above, in order to get the solution of Π for the initial (di)graph G.

The above scheme was used in a specific way in many sequential [14, 16, 17] and parallel applications [14, 36], but was not employed as a general framework for solving problems. We apply this scheme to the following fundamental graph problems and improve upon previous results in the case of sparse digraphs: all pairs shortest paths (APSP), finding a negative cycle and all pairs reachability (APR). (For a definition of these problems, see Section 2.)

1.1. *Overview of previous work and motivation*

In the following, let $G = (V(G), E(G))$ denote any (di)graph, and let $n = |V(G)|$ and $m = |E(G)|$. The sequential hammock decomposition technique was developed by Frederickson in [16, 17]. Let $\tilde{\gamma}$ be the minimum number of hammocks into which G decomposes. It was shown in [16] that $\tilde{\gamma} = \Theta(\gamma(G')) = \Theta(\bar{\gamma}(G'))$, where $\gamma(G')$ and $\bar{\gamma}(G')$ are the genus and crosscap number [24] of a graph G', respectively. Here G' is G with a new vertex v added and edges from v to every vertex of G. Moreover, $\gamma(G') \leqslant \gamma(G) + q$, where G is supposed to be cellularly embedded on an orientable surface of genus $\gamma(G)$ such that all vertices are covered by at most q of the faces.[2] Therefore, $\tilde{\gamma}(G)$ can range from 1 up to $\Theta(m)$ depending on the topology of the graph. (Note that $\tilde{\gamma} = q$, if G is planar, and $\tilde{\gamma} = 1$ if G is outerplanar.) Frederickson showed in [16] how to produce a decomposition of G into $\Theta(\tilde{\gamma})$ hammocks in $O(n+m)$ time, without an embedding of G into some topological surface to be provided with the input.

Hammock decomposition seems to be an important topological decomposition of a graph which proved useful in solving efficiently APSP problems [16, 17, 36], along with the idea of storing shortest path information into compact routing tables (succinct encoding) [19, 42]. (In this representation each edge $\langle v, w\rangle$ is associated with at most $\tilde{\gamma}$ disjoint subintervals of $[1, n]$ such that $\langle v, w\rangle$ is the first edge in a shortest path from v to every vertex in these subintervals.) Compact routing tables are very useful in space-efficient methods for message routing in distributed networks [19, 42]. The main benefit from this idea is the beating of the $\Omega(n^2)$ sequential lower bound for APSP (if the output is required to be in the form of n shortest path trees or a distance matrix) when $\tilde{\gamma}$ is small. Frederickson showed how to produce such an encoding in $O(n\tilde{\gamma})$ time for planar digraphs [17] and in $O(n\tilde{\gamma} + \tilde{\gamma}^2 \log \tilde{\gamma})$ time for sparse digraphs [16]. Alternative encodings of APSP information, with partial use of compact routing tables, take $O(n + \tilde{\gamma}^2)$ (resp. $O(n + \tilde{\gamma}^2 \log \tilde{\gamma})$) time for planar (resp. sparse) digraphs [16, 18].

Thus, efficient parallelization of this decomposition (along with other techniques) may lead to a number of processors much less than $M_s(n)$ (i.e. the number required by the best known parallel algorithm that uses the matrix powering method to solve APSP in time $O(\log^2 n)$ on a CREW PRAM), and hence beat the so-called *transitive closure bottleneck* [29]. Such a "beating", for planar digraphs, was first achieved in [36]. (The best value, up to now, for $M_s(n)$ is $O(n^3(\log\log n)^{1/3}/(\log n)^{7/6})$ [26].) If

[2] We say that a face f *covers* a set $S \subseteq V(G)$ of vertices, if every vertex in S is incident to f. Here q is the minimum number of faces that together cover all vertices of G and varies from 1 up to $\Theta(n)$.

the digraph is provided with a balanced $O(n^{\mu})$-separator decomposition,[3] $0 < \mu < 1$, it has recently been shown by Cohen [6] that one can find APSP in $O(\log^3 n)$ time with $O((n^2+n^{2\mu+1})/\log^3 n)$ processors on an EREW PRAM. For the case of planar digraphs, where a balanced $O(n^{1/2})$-separator decomposition can be computed in $O(\log^4 n)$ time using $O(n^{1+\varepsilon})$ CREW PRAM processors for every $0 < \varepsilon < 1$ [20], the results in [6] imply an $O(\log^4 n)$-time, $O(n^2/\log^4 n)$-processor CREW PRAM algorithm for the APSP problem. (To the best of our knowledge, it is not known yet how to compute in general balanced $O(n^{\mu})$-separators in NC.)

In the case where a digraph G has negative edge costs, it is well known [3, 11] that there is a shortest path from a vertex v to a vertex u in G iff no path from v to u contains a negative cycle (i.e. a simple cycle for which the sum of its edge costs is negative). Hence, detecting a negative cycle is an essential problem for finding shortest paths in G. The best previous sequential algorithms for solving the negative cycle problem in general digraphs run in $O(nm)$ time [3, 11, 39]. In the case where the digraph has a balanced $O(n^{\mu})$-separator decomposition, the best previous algorithm is due to Mehlhorn and Schmidt [34] and runs in $O(n^{3\mu} + n^{1+\mu}\log n)$ time. In parallel computation, the main tool used was the matrix powering method (using e.g. the approach described in [30]), which means that one needs $O(\log^2 n)$ time and $M_s(n)$ processors on a CREW PRAM. In the case where the digraph is provided with a balanced $O(n^{\mu})$-separator decomposition, the best previous algorithm was given in [6] and runs in the same resource bounds with the APSP algorithm presented there.

Concerning the APR problem, even in the case of sparse digraphs, the best previous parallel algorithm uses the matrix powering method and hence needs $O(\log^2 n)$ time by employing $M_r(n)$ CREW PRAM processors [28]. (The best value for $M_r(n)$ is $O(n^{2.376})$ [10]. The best sequential algorithm runs in $O(\min\{M_r(n), nm\})$ time [11].) Also, if we are provided with a balanced $O(n^{\mu})$-separator decomposition, Cohen in [6] gives an algorithm for the APR problem which runs in the same resource bounds with the one for the APSP problem presented in that paper.

It is evident by the above discussion that all previous parallel algorithms, especially in the case of non-planar sparse digraphs, perform considerably more work compared to the best known sequential ones. Moreover, in all cases the lower bound of $\Omega(n^2)$ – for the work – seems difficult to beat, even in the case where the digraph has nice topological properties.

We should also note here that all the above problems, besides their own significance, are frequently used as subroutines in many other applications. For example, finding shortest path information in digraphs has applications to network optimization problems [3], as well as to other areas like incremental computation for data flow analysis and interactive systems design [41]. The negative cycle problem has also

[3] A separator of a digraph G is a subset of vertices whose removal disconnects G into at least two components. If each component has size at most a constant fraction of $|V(G)|$, then the separator is called balanced. A balanced $f(n)$-separator decomposition is a recursive decomposition of G using balanced separators, where subgraphs of size n have separators of size $O(f(n))$.

applications to network optimization problems [3]. Moreover, in [39] applications to two dimensional package placement and to checking constraints in VLSI layout, are mentioned. Applications of the reachability problem can be found in [11, 37]. Therefore, efficient solutions of the APSP, APR and negative cycle problems can also lead to efficient solutions for other problems too.

1.2. Our results

Given a (di)graph G, we can generate a hammock-on-ears decomposition of G into an asymptotically minimum number of $\Theta(\tilde{\gamma})$ hammocks in $O(\log n \log\log n)$ time using $O(n+m)$ CREW PRAM processors. We note here that an embedding of G on some topological surface does not need to be provided with the input. The time bound can be further reduced to $O(\log n \log^* n)$ if either a CRCW PRAM is used, or G belongs to the class of linearly contractible graphs (see Section 4). Examples of this class are planar graphs and graphs of constant genus. A sequential implementation of our algorithm runs in $O(n+m)$ time and matches the running time of [16].

We next apply the hammock-on-ears decomposition (along with other techniques and the general scheme it implies) to the problems discussed earlier and achieve the following on a CREW PRAM:

(1) We give an algorithm for computing an encoding of APSP information into compact routing tables in any sparse digraph G with real-valued edge costs, but no negative cycles, in $O(\log^2 n)$ time by employing $O(n\tilde{\gamma} + M_s(\tilde{\gamma}))$ processors and using $O(n\tilde{\gamma})$ space. In the case of planar digraphs we can achieve further improvements; namely $O(\log^2 n + \log^4 \tilde{\gamma})$ time and $O(n\tilde{\gamma})$ processors, thus being away of optimality by a polylogarithmic factor only. Note that if we use alternative encodings for APSP information (see Section 4) that need $O(n+\tilde{\gamma}^2)$ space, we can achieve further improvements on the processor bounds. Namely, $O(n+M_s(\tilde{\gamma}))$ processors for the general case and if the graph is planar, the processor bound can be further reduced to $O(n+\tilde{\gamma}^2/\log^4 \tilde{\gamma})$.

(2) We overcome the negative cycle assumption mentioned above, by presenting an algorithm which finds (if it exists) a negative cycle in any sparse digraph G with real-valued edge costs. Our algorithm runs in $O(\log^2 n)$ time by employing $O(n+M_s(\tilde{\gamma}))$ processors and uses $O(n+\tilde{\gamma}^2)$ space. A sequential implementation of our algorithm runs in $O(n+\min\{\tilde{\gamma}^{3\mu}+\tilde{\gamma}^{1+\mu}\log\tilde{\gamma}, \tilde{\gamma}^2\})$ time, where $0<\mu<1$ and G has a balanced separator of size $O(n^\mu)$. The algorithm is based on a novel optimal solution of the same problem when the input digraph is outerplanar (thus tackling step 2 of the general scheme), in $O(\log n \log^* n)$ time by employing $O(n/\log n \log^* n)$ processors. In the case of planar digraphs, we can achieve further improvements; namely $O(\log^2 n + \log^4 \tilde{\gamma})$ time and $O(n+\tilde{\gamma}^2/\log^4 \tilde{\gamma})$ processors. Also, in this case our sequential implementation runs in $O(n+\tilde{\gamma}^{1.5}\log\tilde{\gamma})$ time.

(3) We give an algorithm for providing a succinct encoding of APR information into compact routing tables in any sparse digraph G, in $O(\log^2 n)$ time with $O(n\tilde{\gamma}+M_r(\tilde{\gamma}))$ processors using $O(n\tilde{\gamma})$ space. In the case of planar digraphs our algorithm runs in the same bounds with those given above for the APSP problem. If we use alternative

encodings for APR information (see Section 4) that need $O(n+\tilde{\gamma}^2)$ space, we can achieve further improvements on the processor bounds. Namely, $O(n+M_r(\tilde{\gamma}))$ processors for the general case and if the graph is planar, the processor bound can be further reduced to $O(n+\tilde{\gamma}^2/\log^4\tilde{\gamma})$.

We note that: (i) The hammock-on-ears decomposition fully extends the topological characteristics of the input (di)graph and can be advantageously used in the design of algorithms which are parameterized in terms of these characteristics. The better the topological characteristics are (i.e. the smaller the $\tilde{\gamma}$), the more efficient the algorithms for the above applications become. (ii) The bounds of our parallel algorithms for solving the APSP, APR and negative cycle problems match the previously best bounds when $\tilde{\gamma}=\Theta(n)$. However, in all cases where $\tilde{\gamma}=o(n)$, our bounds are significantly smaller. Furthermore, we beat the $M_s(n)$ or $M_r(n)$ lower bound on the number of processors, for the same problems, on any sparse digraph with $\tilde{\gamma}=o(n)$. (iii) Our algorithms for all the above problems explicitly construct the graph decomposition. If additionally a balanced separator decomposition is provided with the input, our general scheme guarantees that the algorithms presented here will still provide better results for all of these problems. Note that the bounds in this case are similar to the ones given for planar digraphs and are discussed in Section 5. (iv) The sequential version of our negative cycle algorithm is clearly an improvement over the algorithms of [3, 34, 39] (in the case where $\tilde{\gamma}=o(n)$) and moreover, overcomes the assumption of nonexistence of a negative cycle for the APSP problem in the sequential results of Frederickson [16, 17].

The paper is organized as follows. In Section 2 we give some preliminaries and define the hammocks. Our parallel algorithm for generating the hammock-on-ears decomposition of a graph is presented in Section 3, while its applications to the problems discussed earlier are presented in Section 4. Finally, in Section 5 we conclude and give some further results.

2. Preliminaries

We assume familiarity with basic graph terminology. (For terms not defined in the paper, the reader is referred to any standard textbook on the subject, e.g. [27].)

Let $G=(V,E)$ be a graph. If G is directed, we will refer to E as the set of *arcs* of G. Otherwise, we will refer to it as the set of *edges* of G. A graph is called *outerplanar* if it can be embedded in the plane so that all vertices are on one face. A tree is called *convergent* (resp. *divergent*) if the edges of the tree point from a node to its parent (resp. children).

Let $G=(V,E)$ be a digraph with real-valued arc costs. We will call the undirected graph $G_u=(V,E_u)$, where $E_u=\{(v,w)\mid\langle v,w\rangle\in E \text{ or } \langle w,v\rangle\in E\}$, the *undirected version* of G. If there is a (directed) simple path in G from a vertex v to a vertex w, we say that *w is reachable* by v. The *length* of a simple path P is the sum of the costs of all arcs in P and the *distance* between two vertices v and w is the minimum length of a path between v and w. Such a path of minimum length is called *shortest path*.

A simple cycle C in G is a simple path starting and ending at the same vertex v. If the length of C is smaller than zero, then C is called *negative*. The *all pairs shortest paths problem* asks for finding shortest path information between every pair of vertices s and t in G (provided that there are no negative cycles in G). The *negative cycle problem* asks for detecting if there exists a simple cycle in G of negative length. If it exists, then output such a cycle. The *all pairs reachability problem* asks for finding reachability information for every pair of vertices s and t in G, i.e. if there is a directed path from s to t. (Note that this problem is also known as the transitive closure problem [11].)

The notion of *compact routing tables* appeared in [19] and is based on the ideas of [42]. Let the vertices of G be assigned names from 1 up to n (in a manner to be discussed in Section 4). For each arc $\langle v,w\rangle$, 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 arc on this path being $\langle v,w\rangle$. (In the event of ties, apply a tie-breaking rule so that for each v,u, $v \neq u$, u is in just one set $S(v,w)$.) Let each $S(v,w)$ be described as a union of a minimum number of subintervals of $[1,n]$. We allow a subinterval to wrap around i.e. the set $\{i, i+1,\dots,n,1,2,\dots,j\}$ (where $i > j+1$) is denoted as $[i,j]$. The set $S(v,w)$ will be called the *compact label* of $\langle v,w\rangle$. If G is outerplanar, then $S(v,w)$ is a single interval [19]. Otherwise, $S(v,w)$ can consist of more than one subinterval. A compact routing table will then have an entry for each of the subintervals contained in a compact label at v.

An *ear decomposition* $D = \{P_0,P_1,\dots,P_{r-1}\}$ of an undirected biconnected graph $H = (V(H),E(H))$ is a partition of $E(H)$ into an ordered collection of edge-disjoint simple paths $P_0,\dots,P_{r-1}$ such that: (i) P_0 is an edge; (ii) $P_0 \cup P_1$ is a simple cycle; (iii) each endpoint of P_i, $i > 1$, is contained in some P_j, $j < i$; and (iv) none of the internal vertices of P_i are contained in any P_j, $j < i$. The paths in D are called *ears*. An ear is *open* if it is acyclic and is *closed* otherwise. A *trivial ear* is an ear containing only one edge. D is an *open ear decomposition* if all of its ears are open.

We define hammocks following [16]. (For basic notions concerning topological graph theory, the reader is referred to [24].) Let G be a digraph whose undirected version is biconnected. (Otherwise, we consider each biconnected component separately.) Let v be a vertex not in G. Let $G' = (V(G'),E(G'))$, where $V(G') = V(G) \cup \{v\}$, $v \notin V(G)$, and $E(G') = E(G) \cup \{\langle v,w\rangle\colon w \in V(G)\}$. Let $\hat{G}'$ be a cellular embedding of G' on a surface of Euler characteristic neither 1 nor 2. (These cases actually imply that G is outerplanar and therefore lead to an easy decomposition of G into one hammock. For details, see [16].) The hammocks of G will be defined with respect to $\hat{G}'$. First, find the undirected version of $\hat{G}'$. Next, triangulate each face that is bounded by more than three edges in such a way that no additional edges incident on v are introduced. Finally, delete v and its adjacent edges yielding embedding $I(G)$. In $I(G)$ one large face is always created (the *basic face*) containing all the vertices. The remaining faces are all triangles. The resulting $I(G)$ is called a *basic face embedded graph*. Faces are grouped together to yield certain outerplanar graphs called *hammocks* by using two operations: *absorption* and *sequencing*. Absorption can be done by initially marking each edge that borders the basic face. Let f_1, f_2 be two nonbasic faces sharing an edge.

Suppose f_1 contains two marked edges. Then *absorb* f_1 into f_2. (This is equivalent to first contracting one edge that f_1 shares with the basic face. The first face becomes a face bounded by two parallel edges, one of which also belongs to f_2. Then delete this edge, merging f_1 and f_2.) Repeat the absorption until it can no longer be applied.

After the end of absorptions, group remaining faces by sequencing. 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. Each sequence then comprises an outerplanar graph. Expanding the faces that were absorbed into faces in the sequence yields a graph that is still outerplanar. Each such graph is called a (*major*) *hammock*. The first and last vertices on each face of the hammock are called *attachment vertices*. Note that there are four attachment vertices per hammock that constitute the only way of "communication" between the hammock and the rest of the graph. Any edge not included in a major hammock is taken by itself to induce a (*minor*) *hammock*. A *hammock decomposition* of $I(G)$ is the set of all major and minor hammocks. It follows by the sequencing operation (see also [16]) that each major hammock is defined on a set of vertices which form two sequences on the basic face. Each such vertex sequence starts and ends with an attachment vertex and determines a path of marked edges.

Let $\tilde{\gamma}(G)$ be the minimum number of hammocks into which G can be decomposed. It is not hard to see that the total size of a compact routing table T is $O(n\tilde{\gamma}(G))$ by constructing T through a decomposition of G into $\tilde{\gamma}(G)$ hammocks (see also [19]). Having APSP information encoded into compact routing tables, it follows from [17, 36] that a convergent or divergent shortest path tree can be computed in $O(n \log \tilde{\gamma}(G))$ sequential time [17], or in $O(\log n \log \tilde{\gamma}(G))$ time using $O(n/\log n)$ EREW PRAM processors [36]. (The latter computation can be alternatively accomplished in $O(\log n)$ time with $O(n)$ processors.)

Since the direct approach for decomposing G into a minimum number of hammocks would involve finding an embedding of minimum genus (shown to be NP-complete in [40]), Frederickson [16] used an alternative approach, the so called *partial hammock decomposition.* (A partial hammock is a subgraph of a hammock in some basic face embedding $I(G)$.) This decomposition decomposes G into $O(\tilde{\gamma}(G))$ partial hammocks using two operations: pseudo-absorption and pseudo-sequencing. As their names indicate these operations are analogous to absorption and sequencing (defined above) and in general produce only partial hammocks. The interesting feature of these operations is that they are applied to G without an embedding to work with.

3. The algorithm for the hammock-on-ears decomposition

Let G_u be the undirected version of a digraph G. We assume that G_u is biconnected. (If not, we work on each biconnected component separately.)

Let v_1, v_2 be a separation pair of G_u that separates V_1 from $V_2 = V - V_1 - \{v_1, v_2\}$, where the subgraph induced on V_1 is connected. Let J_1 be the subgraph of G_u induced on $V_1 \cup \{v_1, v_2\}$ and let J be J_1 with the edge (v_1, v_2) added (if it is not already in J_1). Let G_1 be the graph resulting from contracting all edges with both endpoints in V_1. (The multiple edges, possibly created during the contractions, are also deleted.) If v_1 and v_2 are chosen such that J is outerplanar and V_1 is maximal subject to J being outerplanar, then J_1 is called an *outerplanar outgrowth* of G_u and (G_1, J) an *outerplanar trim* of G_u. The following lemma has been proved in [16].

Lemma 3.1 ([16]). *Let (G_1, J) be an outerplanar trim of a biconnected graph G_u. Then, $\tilde{\gamma}(G_1) = \tilde{\gamma}(G_u)$ and a basic face embedded graph for G_1 of minimum hammock number can be extended to a basic face embedded graph for G_u of minimum hammock number.*

The above lemma actually says that we may remove all outerplanar outgrowths of the graph G_u, find a minimum decomposition of the remaining graph in outerplanar subgraphs (hammocks) and then reinsert the removed outerplanar outgrowths of G_u. The resulting decomposition still consists of outerplanar subgraphs and moreover, the number of hammocks of the decomposition is minimum. This procedure is called *pseudo-absorption* [16]. The first phase of our parallel decomposition algorithm is to efficiently parallelize the pseudo-absorption procedure. Because of the sequential nature of this procedure, we had to employ different techniques specifically suited for parallel computation such as the ear decomposition search.

After all outerplanar outgrowths have been identified, they are removed from the graph leaving an edge connecting the separation points of the outgrowth, labeled with sufficient information to rebuild the outgrowth after the decomposition. A second procedure, called *pseudo-sequencing* [16] is then applied, in order to identify sufficiently long sequences of faces from each hammock. The second phase of our algorithm is to parallelize the pseudo-sequencing procedure.

3.1. Pseudo-absorption

The first step in our *pseudo-absorption* algorithm, is to create an open ear decomposition of the graph. The key observation is that the first (lower numbered) ear that involves an outerplanar outgrowth enters the outgrowth from one of the separation vertices and exits from the other. Moreover, all ears of the outerplanar outgrowth whose endpoints are vertices of this first ear, have endpoints that are *consecutive* vertices of this ear, otherwise the outerplanarity assumption would be violated. The same holds for any ear with greater number that is included in the outgrowth: It must “touch” a lower numbered ear in consecutive vertices. For the same reason there are no ears (included in the outgrowth) with endpoints in two different ears. The shape of the outerplanar outgrowth tends to be as in Fig. 1, where w_1, u_1 and w_2, u_2 are pairs of consecutive vertices of the ear P_i and w_4, u_4 are consecutive vertices of the ear P_j.

Based on this observation, we start from an open ear decomposition and try to identify ears that have the above property: Their endpoints are consecutive vertices of

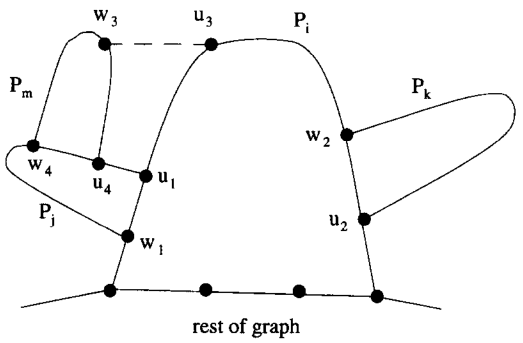

Fig. 1. The shape of a tentative outgrowth on ears P_i, P_j, P_m and P_k, where $m > j > i$ and $k > i$. Note that this is not the final outgrowth and further splitting is necessary, because of the ear with endpoints w_3 and u_3.

another ear. Any such maximal set of ears is a possible outerplanar outgrowth. It is useful to view this set of ears as a new ear, and transform the ear decomposition to a new one where there are no ears having both their endpoints to be consecutive vertices of another ear. Note also (cf. Fig. 1) the possible existence of ears (both trivial and non-trivial) that connect non-consecutive vertices, which may destroy the outerplanarity (e.g. the ear with endpoints w_3, u_3 in Fig. 1), and which are treated separately in the next step. This step consists of retransforming the new ear decomposition by dividing each ear into a number of paths such that each new path is a maximal candidate of being an outerplanar outgrowth. The transformation of the ear decomposition is done in Stage 1 of algorithm *Find_Outgrowths* below.

Algorithm *Find_Outgrowths. Stage* 1:

BEGIN

1.1. Find an open ear decomposition, $D = \{P_0, P_1, \ldots, P_{r-1}\}$, of G_u.

1.2. Construct an auxiliary graph A as follows: Create a vertex for each ear P_i. For each P_i, if both its endpoints belong to the same ear P_j, $j < i$, and are consecutive vertices of P_j, and moreover there is no other non-trivial ear P_k having these two vertices as endpoints, then let (P_i, P_j) be an edge of A.

1.3. Find the connected components of A. Each connected component is a tree (there is no cycle because each endpoint of an ear P_i belongs to a smaller numbered ear). Consider as the root of such a tree the vertex corresponding to the ear that either has its endpoints on different ears, or on the same ear P_k but the endpoints are not consecutive vertices of P_k.

1.4. In each connected component, join the ears of the component into a single ear by converting for each ear the edge between its endpoints into a new trivial ear, and also by rearranging the pointers of the vertices that point to the next vertex in the obvious way. The number of the new ear is equal to the number of the ear which is the root of the tree (connected component). Call the new ear decomposition D_1.

1.5. Now call *internal* vertices of a (new) ear P_i^1, all its vertices except for its endpoints and *internal ears with respect to* P_i^1 all ears (both trivial and non-trivial) whose both endpoints are vertices of P_i^1. Analogously, ears that have only one endpoint to be a vertex of P_i^1 are called *external*. We call *dividing vertices of Stage* 1 the endpoints of all non-trivial ears. Divide each ear P_i^1 into a set of paths each having as endpoints two consecutive (on the ear P_i^1) dividing vertices of Stage 1 (see Fig. 2). Renumber the vertices of the new paths so as to be consecutively numbered. Call the new decomposition into paths, D_2.
END.

Stage 1 of algorithm *Find_Outgrowths* is based on the following lemma.

Lemma 3.2. *An outerplanar outgrowth, not including the edge P_0 of the original ear decomposition, consists of a consecutive subpath of a single path P_i^2 of decomposition D_2 and possibly of some edges that are internal to P_i^2.*

Proof. First observe that the restriction imposed by the lemma (i.e. the edge P_0 should not be part of the outgrowth) follows from the fact that Stage 1 of algorithm *Find_Outgrowths* correctly identifies outgrowths, if the first ear enters the outgrowth by one of the separation vertices and exits from the other. At the end of the current section we show how to remove this restriction.

Let J_1 be an outerplanar outgrowth with separation vertices v_1 and v_2. By definition, J, i.e. J_1 with the edge (v_1, v_2) added, must be outerplanar.

We will show that the vertices of J_1 form a subpath of some path P_i^2 of decomposition D_2. Since no path in D_2 has external or internal non-trivial paths attached to it, this amounts to showing that the endpoints v and w of P_i^2 can be included in J_1 only as separation vertices (i.e. they cannot be internal to J_1). Assume to the contrary that at least one endpoint of P_i^2, say v, is in J_1 but is not a separation vertex (see Fig. 3). Since v was declared dividing at Step 1.5, it is the beginning of at least two more paths (except P_i^2). There are several cases to consider. If at least one of the paths is non-trivial (path $v - v_1$), then the second path cannot include v_1: if it did, then it must necessarily be simple edge (dashed line in Fig. 3), otherwise outerplanarity would be violated. But then v_1 is dividing and path $v - v_1$ is connected to two consecutive vertices of the ear of D_1 from which the trivial path $v - v_1$ evolved. This possibility was

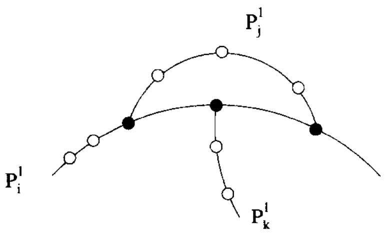

Fig. 2. ● = Dividing vertex of Stage 1 of algorithm *Find_Outgrowths*.

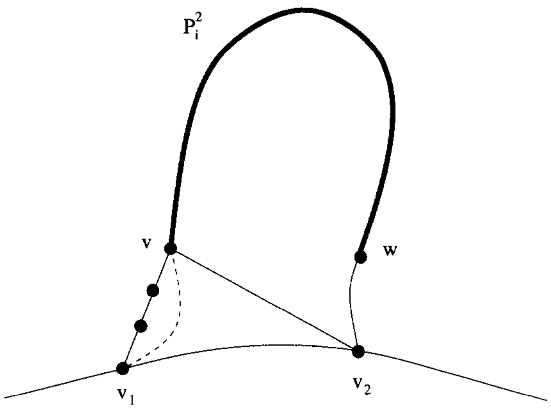

Fig. 3. Proof of Lemma 3.2.

ruled out at Step 1.4 and thus we come to a contradiction. Since all paths beginning at v must either be totally included in J_1 or pass through either v_1 or v_2, we conclude that v is directly connected to a vertex in path $w - v_2$. But then similar arguments result in the necessity to have w directly connected to a vertex in path $v - v_1$, again a violation of outerplanarity unless w is directly connected to v. This is absurd however since, now, P_i^2 has endpoints to be consecutive vertices of another path. We conclude that v can be included in J_1 only as a separation vertex. Observe that this construction does not rule out the possibility of a path to have external or internal edges attached to it and thus further splitting may be necessary. $\square$

In the beginning of the second stage we have a new decomposition into paths where each non-trivial path P_i^2 is a maximal candidate of being an outerplanar outgrowth. A path P_i^2 could have however *external* or *internal edges* attached to it that may destroy the outerplanarity and hence only portions (if any) of the subgraph induced on P_i^2 could be outerplanar outgrowths. This second stage resolves this problem.

Now referring to all paths P_i^2 of D_2, call *dividing vertices of Stage* 2 the endpoints of:

(i) P_i^2 and of all external edges with respect to P_i^2.

(ii) All internal edges that intersect with other internal or external edges. Two internal edges with endpoints w_1, w_2 and u_1, u_2 intersect, if exactly one endpoint of the second edge (e.g. exactly one of u_1 and u_2) lies between the endpoints of the first one on the path P_i^2. In the case where one is external and only for the purpose of identifying possible intersections, consider it as being connected to one endpoint of P_i^2 and handle it as being internal. Equivalently, if we consider the numbers assigned to the nodes of P_i^2 at the end of Stage 1, two edges intersect if the corresponding intervals intersect and no interval is a subset of the other.

(iii) All internal edges that do not intersect with other edges (internal or external handled as above) and whose endpoints are separated by at least one dividing vertex of cases (i) and (ii).

We are now ready to give Stage 2 of algorithm *Find_Outgrowths*.

Algorithm *Find_Outgrowths. Stage* 2:

BEGIN

2.1. For all paths P_i^2 of decomposition D_2 (produced by Stage 1) in parallel, locate the dividing vertices of types (i) to (iii) defined above.

2.2. Subgraphs of P_i^2 that are separated by two consecutive dividing vertices are outerplanar outgrowths (see Fig. 4). Delete each such subgraph and substitute it by a single edge that connects these dividing vertices. In order to be able to easily reconstruct the outgrowths, label the edge by the numbers of the vertices of the corresponding outgrowth.

END.

Stage 2 of the algorithm is based on the following lemma.

Lemma 3.3. *The subgraphs induced on each of the portions into which the dividing vertices separate a path P_i^2 are outerplanar outgrowths (provided of course, that they are not simple edges). The separation vertices of an outgrowth are the two dividing vertices that define the portion.*

Proof. Let Q be the subgraph induced on a portion defined by two consecutive dividing vertices. By definition, Q must include all edges of the path P_i^2 that have both endpoints on Q. Observe now that no two of those edges intersect, since intersecting edges define dividing vertices and consequently the presence of such edges would result in further splitting of P_i^2. Moreover, there are no edges connecting a vertex in the given portion with the rest of the graph. If there were such an edge, then it would fall into cases (i) or (iii) and its endpoints would also be dividing vertices. The graph Q defined in this way, is clearly outerplanar and is separated from the rest of the graph by the two dividing vertices. It is maximal, since the addition of any vertices must necessarily include at least one of the dividing vertices that define Q. This vertex now becomes internal to Q (i.e. it is no longer a separation vertex of Q). If this vertex was found to be dividing because it falls into case (i), then a portion of at least one more path must be included in the outerplanar outgrowth which by Lemma 3.2 is impossible. If it is of type (ii), then it is the endpoint of an intersecting edge which must also be part of Q, otherwise a link with the rest of the graph is introduced. Now this edge

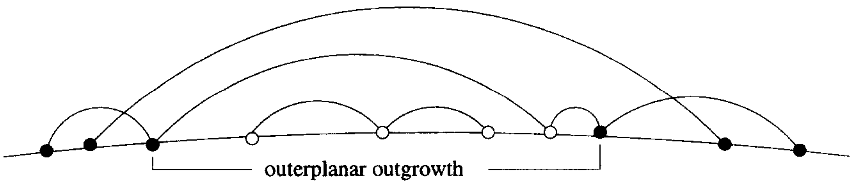

Fig. 4. ● = Dividing vertex of Stage 2 of algorithm *Find_Outgrowths*.

intersects with another which has an endpoint between the endpoints of the first. This endpoint must also be included in Q since in order to separate it from Q at least two more separation vertices are needed. For the same reason the intersecting edge must also be included in Q, thus destroying the outerplanarity. If the vertex was found to be dividing because it falls in case (iii), then it is the one endpoint of an edge which has a dividing vertex of types (i) or (ii) between its endpoints. As before this dividing vertex must be included in Q and consequently at least one pair of intersecting edges, again a contradiction. $\square$

Locating dividing vertices of type (i) is straightforward. Also, if we determine dividing vertices of type (ii), it is easy to determine dividing vertices of type (iii). In order however to locate the dividing vertices of type (ii) we need to locate the intersecting edges. Considering the numbering of the vertices of P_i^2, we conclude that this problem is equivalent to the following one: Given a set of intervals on $[1,n]$ determine all intervals that intersect with at least another interval.

In our problem we need to determine all paths that are simple edges and intersect with other edges. The endpoints of the edges are the boundaries of the intervals, considering the path P_i^2 to be the line of numbers. This is actually a geometric problem and we manage to solve it in parallel time $O(\log p)$ using $O(p)$ processors, where p is the number of intervals involved. We note here that the literature of parallel geometry is rich in related problems that study intersections of line segments in the plane (see e.g. [2, 23]). However the versions of the problems studied there, usually report *all* intersections and carry therefore the burden of the size of the output either on the time or the processor bounds. In our case we merely want to report intervals that have some intersection.

We now briefly discuss how to solve this problem using a data structure mentioned in [32] called "priority search tree". A priority search tree of the points (x_i, y_i), $i = 1,\ldots,p$, is a binary tree whose nodes keep the points sorted from left to right according to their x coordinate and which is a heap w.r.t. the y coordinate (i.e. a node always keeps a value that is greater than the value kept in its children). Now let $I = (x_1, x_2)$ be an interval chosen from a set S of p intervals of $[1,n]$. We say that I has a *right intersection* (resp. *left intersection*) in S, if there is an interval $I' = [x_1', x_2'] \in S$ such that $x_1 < x_1' < x_2$ and $x_2' > x_2$ (resp. $x_1 < x_2' < x_2$ and $x_1' < x_1$). The following lemma holds.

Lemma 3.4. *Let R be a priority search tree constructed on a set S of p intervals (where the intervals are seen as points). Then, given an interval $I \subseteq [1,n]$ we can test whether I has a right intersection in S in sequential time* $O(\log p)$. *Symmetrically, we treat left intersections by a priority search tree that has reversed the heap property. Moreover, constructing R in parallel can be accomplished in* $O(\log p)$ *time using* $O(p)$ *CREW PRAM processors.*

Proof. Follows from a well-known reduction to the problem of locating all points on a plane that lay in half-infinite band with sides parallel to the x and y-axes, see [33].

In the same reference, the construction of the priority search tree basically requires a sorting of the points according to their x-coordinate and a knock-out tournament in order to establish the heap property of the y-coordinate. It is not difficult to see that both steps can be parallelized within the resource bounds stated in the lemma. □

Therefore we have:

Lemma 3.5. *The problem of interval intersections, for a set S of p intervals, can be solved in* $O(\log p)$ *time by employing* $O(p)$ *CREW PRAM processors.*

Proof. By Lemma 3.4 the tree R on S can be constructed in $O(\log p)$ time using $O(p)$ CREW PRAM processors. Given this tree, we can solve interval intersections in the same resource bounds by letting each processor to query one interval in parallel. □

Theorem 3.1. *Algorithm Find_Outgrowths correctly identifies all outerplanar outgrowths of an n-vertex, m-edge biconnected graph G in* $O(\log n \log\log n)$ *time using* $O(n+m)$ *CREW PRAM processors.*

Proof. In Stage 1 of the algorithm, subsets of ears that possibly form outerplanar outgrowths are identified. Each one of these subsets is converted into one new ear and finally into a set of paths. Next, each path is divided into portions by invoking Stage 2. Lemma 3.3 guarantees that the portions are outerplanar outgrowths. Stage 1 of the algorithm employs an open ear decomposition procedure, a connected components procedure and also at Steps 1.4 and 1.5 a list ranking procedure. Stage 2 can be done in $O(1)$ time except for the identification of the dividing vertices, which by Lemma 3.5 is accomplished in $O(\log n)$ time using $O(n+m)$ CREW PRAM processors. Therefore, the time and processor bounds are dominated by the bounds of the connected components algorithm [5] (which also dominates the bounds for finding an open ear decomposition [31]), and which are those stated in the theorem. □

We end this section by showing how to remove the restriction in algorithm *Find_Outgrowths* as promised in the proof of Lemma 3.2. We remind the reader that the problem was that the algorithm correctly identifies an outerplanar outgrowth, if the first (lower numbered) ear that involves the outgrowth enters from one separation point and exists from the other. The first ear however (the ear P_0) may have one or both endpoints on an outgrowth. In this case one or two outerplanar outgrowths may not be identified correctly. This can be corrected if we first run the algorithm from an arbitrary ear decomposition, identify all but at most two outerplanar outgrowths and then rerun the algorithm from a different ear decomposition with the endpoints of the new P_0 placed in an already discovered outgrowth. This can be done since the parallel ear decomposition algorithm begins from a spanning tree which also defines the first ear (see [31] for details).

3.2. *Pseudo-sequencing*

This subsection shows how to efficiently parallelize the *pseudo-sequencing* procedure. The parallel algorithm presented is based on ideas developed in [16] where also the main theory behind the procedure can be found. The goal is to identify sequences of faces that cover all but a constant part of a hammock. The hammocks thus produced are called *partial hammocks.*

Let G_{u1} be the undirected graph resulting from the pseudo-absorption procedure. Each edge (x, y) in G_{u1} has a (possibly empty) label representing the outerplanar outgrowth generated by separation vertices x and y. Note that all vertices in G_{u1} have degree greater than 2.

Let H be a hammock in a basic face embedded graph $I(G)$ (as defined in Section 2). For each H, there is a corresponding subgraph H' in G_{u1} that has a special structure which is relative to an outerplanar embedding of H'. Assume that H' is biconnected. The non-biconnected case is handled in a preprocessing step to be discussed later. Since every node of G_{u1} has degree at least three and H' is outerplanar, it follows that all faces (in an outerplanar embedding) of H' are bounded by either three or four edges. (There can be a situation where this is not true, involving the attachment vertices of the hammock which however only represent a constant part.) This observation suggests that a hammock is actually a sequence of triangles and rectangles sharing an edge in a way that outerplanarity is preserved. Consequently, two *neighboring faces* of a hammock form a graph that is isomorphic to one of the three graphs P_1, P_2 or P_3 of Fig. 5. We will next try to identify *interior* edges of a hammock. These are the edges that separate two neighboring faces in the same hammock, i.e. the edges (v_1, v_2) of the previous graphs. Note that the remaining (non interior) edges of H' form two paths corresponding to the paths of marked edges in H. Consequently, the endpoints of the edges of these two paths in H' form two sequences of vertices which are (in general) subsets of the corresponding vertex sequences in H.

The following lemma is a combination of Lemmata 6.2–6.4 that were proved in [16].

Lemma 3.6. *Any sequence of seven faces in a hammock (i.e. combination of triangles and rectangles) falls into at least one of ten possible cases. At least one of the graphs P_1, P_2 or P_3 of Fig. 5 appears in any of the ten cases having at least two of its vertices with degree less than 7. This graph can be identified in any of the cases, in* O(1) *time, by applying one of the following degree tests*:

(1) P_1 *with* $deg(v_2) = deg(v_3) = deg(v_4) = 3$.
(2) P_1 *with* $deg(v_2) = deg(v_3) = 3$ *and* $deg(v_4) = 4$.
(3) P_1 *with* $deg(v_2) = 3$ *and* $deg(v_1) = 5$.
(4) P_1 *with* $deg(v_2) = 3$ *and* $deg(v_1) = 6$.
(5) P_1 *with* $deg(v_2) = deg(v_1) = 4$.
(6) P_1 *with* $deg(v_2) = 4$ *and* $deg(v_3) = deg(v_4) = 3$.
(7) P_2 *with* $deg(v_1) = 4$ *and* $deg(v_2) = deg(v_4) = 3$.
(8) P_2 *with* $deg(v_1) = 4$ *and* $deg(v_2) = deg(v_5) = 3$.

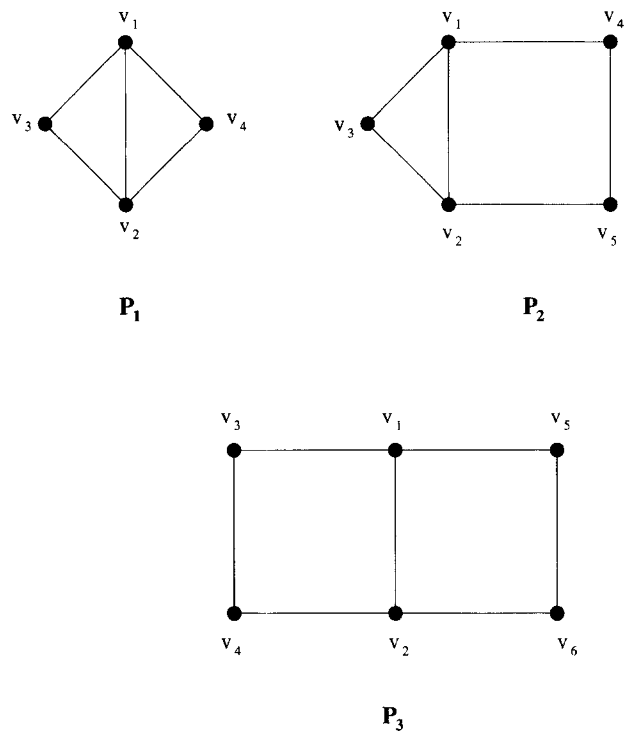

Fig. 5. The graphs used to identify the hammocks.

(9) P_3 *with* $deg(v_1) = deg(v_2) = deg(v_4) = deg(v_6) = 3$.
(10) P_3 *with* $deg(v_1) = deg(v_2) = deg(v_3) = deg(v_6) = 3$.

The importance of the above lemma lies in the fact that it shows firstly that any part of a hammock consisting of seven faces has a constant number of different forms, and secondly that each one of these forms is identifiable by a single processor in constant time after running the ten tests sequentially. So a portion of a hammock is identifiable in constant time by a single processor. After identifying it, we next delete edges that are interior to the hammock. If a vertex, z, of degree 2 results from such deletions, then we perform on it a *contraction operation*, i.e. we remove z, join its two neighbors x and y by an edge (x, y), and assign to this edge a label $\ell_{xy} = (\ell_{xz}, z, \ell_{zy})$, where ℓ_{xz} and ℓ_{zy} are the (possibly empty) labels of the removed edges (x,z) and (z, y), respectively. The edge labels will help us later to rebuild the hammock. Note that a non-empty edge label denotes that the corresponding edge cannot be an interior edge of a hammock. It is important to mention here that after the deletion of an edge and the subsequent contraction of vertices (if any), the hammock is still a sequence of triangles and rectangles and, hence, the same tests can be applied until all the hammock has been shrunk. The following parallel algorithm uses these observations to identify edges that are interior to hammocks.

```
Algorithm Find_Sequences
BEGIN
Repeatedly apply Steps 1–5, for $4.5 \times \lceil \log n \rceil$ times:
  1. Assign one processor to each vertex of $G_{u1}$ and check in turn the 10 cases of
     Lemma 3.6, in order to identify an interior edge of a hammock. (Note that this task
     involves the scanning of an adjacency list of a vertex of length at most six.) Several
     processors may identify the same edge. However, their number will be constant and
     hence breaking ties can be done in constant time.
  2. Create linked lists of identified interior edges as follows. Assign a processor $P_{e_i}$
     to every interior edge $e_i$ identified in Step 1. Each $P_{e_i}$ checks if there is an identified
     interior edge $e_i'$ such that $e_i$ and $e_i'$ belong to the same triangle or rectangle. If this is
     the case, then create a (double) link between them.
  3. In every list $L$ created in Step 2, find an independent set $U$ of size $|U| \geq |L|/3$
     using the algorithm of [22], or the 2-ruling set algorithm of [8]. Mark all $e_i$ that
     belong to $U$.
  4. Delete all marked interior edges and perform the necessary contraction operations
     when a vertex of degree 2 appears.
  5. Update the lists accordingly, by removing the marked elements.
END.
```

Lemma 3.7. *Let G be an n-vertex, m-edge biconnected graph such that for each hammock H in a basic face embedding, its corresponding subgraph H' in G_{u1} is biconnected. Then, algorithm Find_Sequences generates a labeling of the edges that gives a partial hammock decomposition of* $O(\tilde{\gamma}(G))$ *hammocks. The algorithm takes* $O(\log n \log^* n)$ *time and uses* $O(n+m)$ *processors on a CREW PRAM.*

Proof. The correctness of the algorithm comes from Lemma 3.6 and the fact that all contraction operations are performed independently. (It may happen for two identified interior edges that they both bound the same triangle or rectangle. In such a case the simultaneous execution of the two contraction operations does not give the desired edge labeling.) To ensure independence of contraction operations, we first identify (in Step 2) those interior edges whose deletion would cause conflicting contraction operations. These interior edges are connected to lists. It follows that symmetry-breaking in these lists enforces the desired independence. This is done by calling the independent set algorithm of [22], or the 2-ruling set algorithm of [8] in Step 3. Hence, contraction operations are correctly executed. It only remains to argue for the bounds and how the hammocks are generated.

In each iteration at least one edge for every seven faces in every hammock is contracted (i.e. in the worst case all lists consist of only one element). Since each contraction joins two neighboring faces in one, we conclude that at least one-seventh of a hammock is contracted in every iteration. Thus, after at most $4.5 \times \lceil \log n \rceil$ iterations all hammocks will be contracted. All steps, except for Step 3, can be implemented in $O(1)$ time using $O(n+m)$ CREW PRAM processors. Step 3 takes, by [8, 22], $O(\log^* n)$

time using $O(n+m)$ EREW PRAM processors. Hence, the bounds stated in the lemma follow.

Consider now the graph, G_{u2}, resulting from algorithm *Find_Sequences*. Each non-empty label of an edge in this graph, represents a sequence of vertices around the basic face in some embedding. Alternatively, each non empty label defines the one of the two sequences of vertices of a partial hammock. (Having the edge labels of G_{u2} the hammocks are constructed in a way similar to the one described in [16].) Each hammock H' in G_{u2} has now at most six faces, i.e. constant number of edges. Thus the number of partial hammocks generated with respect to a given hammock H is a constant. This means that we have a partial hammock decomposition of $O(\tilde{\gamma}(G))$ hammocks. □

The entire approach above was based on the assumption that the hammocks were biconnected. If however a hammock is not biconnected, the above procedure may delete an external edge of the hammock. To handle the general case, we preprocess G_{u1} to add in dummy vertices and edges so that each hammock becomes biconnected. This addition is done by identifying the two possible subgraphs that a hammock (i.e. an outerplanar graph) must contain in order to be non biconnected (see [16]). This operation is also done by using local degree tests (like the ones given in Lemma 3.6) and is therefore not difficult to parallelize. (It can be actually parallelized in $O(\log n)$ time using $O(n+m)$ CREW PRAM processors, since the tests in this case prevent conflicting operations.) We call this parallel algorithm *Preprocess*. We therefore have:

Theorem 3.2. *Given an n-vertex, m-edge graph G, the decomposition of G into* $O(\tilde{\gamma}(G))$ *partial hammocks can be done in* $O(\log n \log\log n)$ *time using* $O(n+m)$ *CREW PRAM processors.*

Proof. The algorithm to find a partial hammock decomposition of a graph consists of the following steps: First, find the biconnected components of G using the algorithm in [5]. Second, find the outerplanar outgrowths of G (algorithm *Find_Outgrowths*). Third, make the appropriate preprocessing such that the hammocks of the graph are biconnected (algorithm *Preprocess*). Fourth, find the sequences that make up the partial hammocks (algorithm *Find_Sequences*). By Theorem 3.1, Lemma 3.7 and the discussion preceding the theorem, it is clear that the dominating steps in terms of time are the first two ones, which in turn are based on the algorithm of [5] (for finding connected components in parallel) as it was also discussed in the proof of Theorem 3.1. □

Our hammock-on-ears decomposition algorithm can be implemented faster on a CREW PRAM, if G belongs to the class of *linearly contractible* graphs [25]. (A class $\mathscr{C}$ of graphs is called linearly contractible if: (a) for all $G=(V,E)$ in $\mathscr{C}$, $|E| \leqslant c|V|$, where c is a constant; and (b) $\mathscr{C}$ is closed under taking of minors.) Examples of this class of graphs are planar graphs and graphs of constant genus. In [25] it is shown that if G is linearly contractible, then connected and biconnected components (and hence an open ear decomposition) can be found in $O(\log n \log^* n)$ time using an optimal number of EREW PRAM processors.

Also, if a CRCW PRAM is used, then connected and biconnected components, as well as an open ear decomposition, can be found in $O(\log n)$ time with $O(n+m)$ processors using the algorithm in [38]. However, it seems that algorithm *Find_Sequences* cannot take advantage of this model in order to be implemented faster.

The above discussion leads to the following:

Corollary 3.1. *Given an n-vertex, m-edge graph G, the decomposition of G into* $O(\tilde{\gamma}(G))$ *partial hammocks can be done in* $O(\log n \log^* n)$ *time using* $O(n+m)$ *CRCW PRAM processors. Moreover, if G belongs to the class of linearly contractible graphs, then the decomposition can be accomplished in* $O(\log n \log^* n)$ *time using* $O(n+m)$ *CREW PRAM processors.*

4. Applications

In this section we give a high level description of our algorithms for the problems mentioned in the introduction. Recall the general scheme for problem solving using the hammock-on-ears decomposition technique. In the sequel, let $G = (V,E)$ be a sparse biconnected digraph with real-valued arc costs. (The non-biconnected case can be easily handled. Consider for example the APSP problem: it suffices to find APSP information in every biconnected component and then to observe that a shortest path between vertices in different biconnected components has to pass through the *cutpoints* [27] of G. Which cutpoints to consider, can be found by constructing the so-called block-cutpoint tree of G [27]. The construction can be accomplished within the resource bounds for finding the biconnected components. The other problems are treated similarly.) We will use $\tilde{\gamma}$ instead of $\tilde{\gamma}(G)$ for notational simplicity.

4.1. Computing all pairs shortest paths information

The algorithm for computing APSP information in G, encoded in compact routing tables, is as follows:

Algorithm *Sparse_Apsp*

BEGIN

1. Find a hammock-on-ears decomposition of (the undirected version of) G into $O(\tilde{\gamma})$ hammocks.

2. Rename the vertices of G, in a way such that all vertices belonging to a hammock form two consecutively numbered sequences around its external face. (This is needed for the encoding of APSP information into compact routing tables.)

3. Find APSP information in each hammock.

4. Find APSP in G between each pair of attachment vertices of the hammocks as follows. For each hammock H, generate its compressed version $C(H)$. This is done

using the algorithm in [36] to find the distance between each pair of attachment vertices of H. The graph $C(H)$ is just a complete digraph on the attachment vertices in H (which are at most four), with the cost of each arc being the distance in H between its endpoints. Then, generate a *compressed version* $C(G)$ of G by replacing each hammock H by its compressed version $C(H)$ and deleting all but one least expensive copy of any multiple arc. The compressed graph $C(G)$ will have $O(\tilde{\gamma})$ vertices and arcs. Find APSP in $C(G)$ by running another algorithm.

5. For each pair of hammocks determine succinct shortest paths information for each vertex in one hammock to all vertices in the other hammock.

6. For each hammock determine 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.)

END

Theorem 4.1. *Given an n-vertex sparse digraph G with real-valued arc costs (but no negative cycles), that can be decomposed into an asymptotically minimum number of* $O(\tilde{\gamma})$ *hammocks, we can encode all pairs shortest paths information into compact routing tables in* $O(\log^2 n)$ *time using* $O(n\tilde{\gamma} + M_s(\tilde{\gamma}))$ *CREW PRAM processors and* $O(n\tilde{\gamma})$ *space. In the case where G is planar, the encoding can be done in* $O(\log^2 n + \log^4 \tilde{\gamma})$ *time using* $O(n\tilde{\gamma})$ *CREW PRAM processors.*

Proof. Note that Step 1 of algorithm *Sparse_Apsp* corresponds to major step 1 of the general scheme, Steps 2 and 3 correspond to major step 2, Step 4 corresponds to major step 3 and Steps 5 and 6 correspond to major step 4. Now, for the resource bounds the following hold in a CREW PRAM. Step 1 needs $O(\log n \log\log n)$ time using $O(n)$ processors by Theorem 3.2. Step 2 can be done in $O(\log n)$ time using $O(n)$ processors by sorting [7], in a way similar to that described in [16] (i.e. perform a sorting to the list of vertices obtained by arbitrarily concatenating the edge labels of G_{u2} and then by arbitrarily concatenating to them the vertices of G_{u2}). Step 3 can be implemented, overall hammocks, in $O(\log^2 n)$ time using $O(n)$ processors by [36]. Step 4 needs $O(\log n)$ time with $O(n/\log n)$ processors for the generation of the graphs $C(H)$ [36], and $O(\log^2 \tilde{\gamma})$ time using $M_s(\tilde{\gamma})$ processors for APSP in $C(G)$ by running the algorithm of [26]. Step 5 needs for all hammocks $O(\log n)$ time using $O(n\tilde{\gamma})$ processors by [36]. Finally, Step 6 (again by [36]) needs $O(\log^2 n)$ time using $O(n)$ processors. Hence, the claimed bounds follow. The space bound comes from the discussion in Section 2.

If G is planar, then we use in Step 4 the algorithm of [6] to find APSP in $C(G)$ and the algorithm of [36] for generating the graphs $C(H)$ which are outerplanar and of $O(1)$ size. (This latter step takes, overall hammocks, $O(\log n)$ time using $O(n)$ processors.) □

We will now discuss two alternative encodings of APSP information that require less work and space. The first alternative encoding makes partial use of compact routing tables, while the second one does not use compact routing tables at all.

Theorem 4.2. *Given an n-vertex sparse digraph G with real-valued arc costs (but no negative cycles), that can be decomposed into an asymptotically minimum number of* $O(\tilde{\gamma})$ *hammocks, we can compute all pairs shortest paths information using an alternative encoding (compact routing tables for hammocks and global tables for shortest paths among the attachment vertices of the hammocks), in* $O(\log^2 n)$ *time using* $O(n+M_s(\tilde{\gamma}))$ *CREW PRAM processors and* $O(n+\tilde{\gamma}^2)$ *space. In the case where G is planar, this encoding takes* $O(\log^2 n+\log^4 \tilde{\gamma})$ *time using* $O(n+\tilde{\gamma}^2/\log^4 \tilde{\gamma})$ *CREW PRAM processors.*

Proof. The method for finding such an encoding is to run only the first four steps of algorithm *Sparse_Apsp*. Then, using the table for APSP information between every pair of attachment vertices in $C(G)$, a shortest path or distance between two vertices v and w in G is computed as follows. If both vertices belong to the same hammock H we have first to compute the distance between them inside H and then compare it with the minimum among the distances $d(v,a_i)+d(a_i,a_j)+d(a_j,w)$, where $a_i \neq a_j$ and a_i, $1 \leqslant i \leqslant 4$, are the four attachment vertices of H. If w belongs to another hammock H', then we have to choose those i and j that minimize the quantity $d(v,a_i)+d(a_i,b_j)+d(b_j,w)$, where b_j, $1 \leqslant j \leqslant 4$, are the attachment vertices of H'.

Having this encoding, a (divergent or convergent) shortest path tree in G, rooted at some vertex, s, can be computed as follows. Find the shortest path tree rooted at s, in the hammock containing s. In every other hammock H, find shortest path trees rooted at the attachment vertices of H. (From the discussion in Section 2, both computations take $O(\log n)$ time with $O(n/\log n)$ processors overall hammocks.) Finally, from the global table for $C(G)$, we have the shortest path trees in $C(G)$ rooted at the four attachment vertices of the hammock containing s. Having all these shortest path trees, it follows easily by the discussion in the previous paragraph that the desired shortest path tree can be constructed in $O(\log n)$ time using $O(n/\log n)$ processors. □

The second alternative encoding follows the first alternative encoding, but now in each hammock we do not build compact routing tables. Instead, we create the data structures presented recently in [4, 15]. In [15] it is shown how to preprocess an outerplanar digraph in $O(\log n)$ time using $O(n)$ CREW PRAM processors such that subsequently a shortest path, between any two vertices, is computed in $O(\log n)$ time using $O(\max\{1, L/\log n\})$ CREW PRAM processors. (L is the number of arcs of the reported path.) Also, in [4] it is shown how a digraph of constant treewidth is preprocessed in $O(\log^2 n)$ time using $O(n/\log^2 n)$ EREW PRAM processors, such that afterwards the distance, between any two vertices, is computed in $O(\alpha(n))$ time using a single processor. (Note that an outerplanar graph has treewidth 2 and $\alpha(n)$ is the inverse of Ackermann's function.) Therefore, using the above data structures in each hammock (instead of compact routing tables), we have:

Theorem 4.3. *Given an n-vertex sparse digraph G with real-valued arc costs (but no negative cycles), which can be decomposed into an asymptotically minimum number of*

$O(\tilde{\gamma})$ hammocks, we can compute an encoding of all pairs shortest paths in $O(\log^2 n)$ time using $O(n+M_s(\tilde{\gamma}))$ CREW PRAM processors and $O(n+\tilde{\gamma}^2)$ space. Having this encoding a distance between any two vertices can be found in $O(\alpha(n))$ time by a single processor, while the corresponding shortest path can be found in $O(\log n)$ time using $O(\max\{1, L/\log n\})$ processors. In the case where G is planar, we can compute this encoding in $O(\log^2 n + \log^4 \tilde{\gamma})$ time using $O(n+\tilde{\gamma}^2/\log^4 \tilde{\gamma})$ CREW PRAM processors.

Note that the total work required to set up these alternative encodings, in the case of planar digraphs, is far from optimality (w.r.t. the results in [16, 18]) by a polylogarithmic factor only.

4.2. Detecting negative cycles

In this section we give an efficient algorithm for finding a negative cycle in G. The algorithm is based on the hammock-on-ears decomposition technique and on the detection of a negative cycle in an outerplanar digraph. We shall first discuss the outerplanar case and then the general one.

4.2.1. An optimal work algorithm for finding negative cycles in outerplanar digraphs

We will show, in the current subsection, how to solve optimally the negative cycle problem provided that the input digraph is outerplanar and biconnected. (If it is not biconnected, we apply our algorithm to every biconnected component.) Our method is based on a novel extension of the fundamental tree-contraction technique [1, 29] to the tree of interior faces of the outerplanar digraph.

Let $G_o = (V, E)$ be the input outerplanar digraph and let $\hat{G}_o$ be its undirected embedded version. It is well known that the dual graph of $\hat{G}_o$ is a tree, T^F, called the *tree of faces*. (The exterior face is excluded in this construction.) Let us assume that T^F is binary. (At the end of the section we will show how to overcome this assumption.) Any reference to a node of T^F will be considered also as a reference to the face f of $\hat{G}_o$ it represents, and vice versa. We shall refer to the endpoints of the edges bounding a face f in $\hat{G}_o$, as the vertices of f. The subgraph induced by a face f, is the subgraph of G_o consisting of the vertices of f and those arcs whose corresponding (undirected) edges in $\hat{G}_o$ bound f. Let f_1 and f_2 be two neighboring faces in $\hat{G}_o$, i.e. they have an edge in common. By $f_1 \cup f_2$, we shall denote the subgraph of G_o consisting of the union of the subgraphs induced by f_1 and f_2.

Root T^F arbitrarily. For a face (or union of neighboring faces) f, let $s_1(f)$ and $s_2(f)$ be the endpoints of that edge in $\hat{G}_o$ whose dual (tree) edge connects f with its parent node in T^F. We will call $s_1(f)$ and $s_2(f)$ the *associated vertices* of f and denote them as $A(f) = \{s_1(f), s_2(f)\}$.

A shortest path from a vertex x to a vertex y in a subgraph H of G_o will be denoted by $SP(x, y; H)$. Let also $D(x, y; G)$ denote the set $\{SP(x, y; G), SP(y, x; G)\}$ and let $D((x_1, x_2), (y_1, y_2); G)$ denote the set $\{D(x_i, y_j; G) \mid i, j \in \{1, 2\}\}$. In the following, we will use the well-known fact that a separator of G_o is a pair of vertices $\{v, w\}$.

Consider the following *problem* Π: Let G be an outerplanar digraph and let $\{v, w\}$ be a separator, separating G into two subgraphs G_1 and G_2. Suppose also that in each G_i, $i = 1, 2$, there is no negative cycle. Find a negative cycle in G (if it exists).

Solution of Π: Since there is no negative cycle in each G_i, a possible negative cycle $N(G)$ for G will consist of a path in G_1 joined with a path in G_2. Also, $N(G)$ will have v and w as two of its vertices. It follows that to find $N(G)$, it suffices to find $SP(v, w; G_1)$ and $SP(w, v; G_2)$ (or $SP(w, v; G_1)$ and $SP(v, w; G_2)$). The union of these two paths will give the possible $N(G)$. Therefore we have the following.

Proposition 4.1. *Let G be an outerplanar digraph and let $\{v, w\}$ be a separator, separating G into two subgraphs G_1 and G_2. Suppose also that in each G_i, $i = 1, 2$, there is no negative cycle and that the two shortest paths between v and w are known. Then, G can be tested for a negative cycle in* $O(1)$ *time.*

The main idea of our algorithm is the following. We assume that at a certain point, a set of neighboring faces has been tested for a negative cycle. In the case of a positive answer, the negative cycle is output and the algorithm stops. Otherwise, this set of faces is joined to a neighboring set of faces that has also been tested, in order to form a new set. Detection of a negative cycle in this union is done according to the rules implied by Proposition 4.1. Thus, the algorithm proceeds in a bottom-up fashion.

We say that an interior face f has been *evaluated* in the tree of faces T^{F}, iff in the subgraph of G_o induced by its descendant nodes in T^{F} we have tested if there is a negative cycle and in the case of a negative answer, we have computed shortest path information between certain pairs of vertices in f. The main goal is to evaluate the root face of T^{F}.

The parallel tree-contraction algorithm [1, 29] evaluates the root of a tree T processing a logarithmic number of binary trees $T_0, T_1, \ldots, T_k$, where $k = O(\log|T|)$, $T_0 = T$ and T_k contains only one node. Also $|T_\alpha| \leqslant \varepsilon |T_{\alpha-1}|$, $1 \leqslant \alpha \leqslant k$, for some constant $0 < \varepsilon < 1$. The tree T_α is obtained by $T_{\alpha-1}$ by applying a local operation, called *SHUNT* [29], to a subset of the leaves of $T_{\alpha-1}$. The *SHUNT* operation consists in turn by two other operations, called *prune* and *bypass* [1].

Prune operation: Let l be a leaf in tree $T_{\alpha-1}$. Let also v and l' be its parent and sibling, respectively. Then by "pruning l" we denote the deletion of l from $T_{\alpha-1}$.

Bypass operation: Let l' be the unique child of a non-root node v in $T_{\alpha-1}$. Then by "bypassing l'" we denote the joining of l' and v into a new node v'.

Our algorithm follows the execution of the tree-contraction algorithm, as it is described e.g. in [1] or [29], by executing a number of main steps. This means that half of the leaves of the tree of faces T^{F} perform a *SHUNT* operation (main step) and this is repeated for a logarithmic number of times. It is worth noting that as the algorithm proceeds and the tree is contracted, each leaf of T^{F} corresponds to a set of neighboring faces whose removal does not disconnect G_o. Conversely, an internal node of T^{F} corresponds to a set of faces whose removal disconnects G_o. To complete the description of our algorithm we have to show how the information concerning the negative cycle is maintained, i.e. what information is exchanged and/or updated during a *SHUNT* operation. We will need the following two lemmata.

Lemma 4.1. *Let G be an outerplanar digraph and let $\{v,w\}$ be a separator of G, separating it into two subgraphs G_1 and G_2. Let also a,b be vertices of G_1 and let c,d be vertices of G_2. Suppose that the following shortest paths in G_1, G_2 are known*: $D(a,b;G_1)$, $D(v,w;G_1)$, $D(v,w;G_2)$, $D(c,d;G_2)$, $D((a,b),(v,w);G_1)$ *and* $D((v,w),(c,d);G_2)$. *Then, in* $O(1)$ *time we can compute* $D(a,b;G)$, $D(c,d;G)$, *and* $D((a,b),(c,d);G)$.

Proof. It is easy to see (cf. Fig. 6) that:

$$SP(a,b;G) = \min_{length}\{SP(a,b;G_1), SP(a,v;G_1) + SP(v,w;G_2) + SP(w,b;G_1)\}$$

$$SP(c,d;G) = \min_{length}\{SP(c,d;G_2), SP(c,w;G_2) + SP(w,v;G_1) + SP(v,d;G_2)\}$$

$$SP(a,d;G) = \min_{length}\{SP(a,v;G_1) + SP(v,d;G_2) + SP(a,w;G_1) + SP(w,d;G_2)\}$$

where "$\min_{length}$" denotes minimum in length and "+" denotes path concatenation. Similar expressions hold for the rest of the requested shortest paths. Note that each one of the above computations involves a constant number of additions and comparisons of the lengths of at most 4 shortest paths, and after that, a constant number of pointer manipulations (to get the shortest path itself). The claimed bound follows now by the fact that the number of the requested shortest paths is also constant. □

Lemma 4.2. *Let G be an outerplanar digraph and let $\{v,w\}$ be a separator of G, separating it into two subgraphs G_1 and G_2. Let a,b,c and d be vertices of G_2. Suppose that the following shortest paths in G_1 and G_2 are known*: $D(a,b;G_2)$, $D(v,w;G_1)$, $D(c,d;G_2)$, $D((a,b),(c,d);G_2)$, $D((a,b),(v,w);G_2)$ *and* $D((c,d),(v,w);G_2)$. *Then, in* $O(1)$ *time we can compute* $D(a,b;G)$, $D(c,d;G)$ *and* $D((a,b),(c,d);G)$.

Proof. It is easy to see (cf. Fig. 7) that:

$$SP(a,b;G) = \min_{length}\{SP(a,b;G_2), SP(a,v;G_2) + SP(v,w;G_1) + SP(w,b;G_2)\}$$

$$SP(c,d;G) = \min_{length}\{SP(c,d;G_2), SP(c,v;G_2) + SP(v,w;G_1) + SP(w,d;G_2)\}$$

$$SP(a,c;G) = \min_{length}\{SP(a,c;G_2), SP(a,v;G_2) + SP(v,w;G_1) + SP(w,c;G_2)\}$$

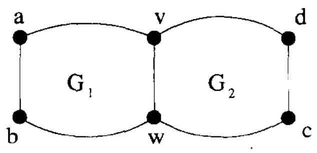

Fig. 6. Proof of Lemma 4.1.

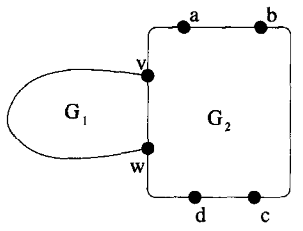

Fig. 7. Proof of Lemma 4.2.

Similar expressions hold for the other requested shortest paths. Using a similar argument with the one used in the proof of Lemma 4.1, it is clear that the above computations can be done in O(1) time. □

We will distinguish between two types of *SHUNT* operations depending on what the sibling of the tree node performing this operation is. In the following, all references to a tree node z will be considered also as references to the subgraph of G_o corresponding to z. The description of the α-th execution of the *SHUNT* operation is as follows.

Type-1: Suppose that leaf l_i performs a *SHUNT* operation and assume also that its sibling l_i' is a leaf (see Fig. 8). From previous executions, the following shortest paths are known: $D(A(l_i); l_i)$, $D(A(l_i); f_i)$, $D(A(f_i); f_i)$, $D(A(l_i'); f_i)$, $D(A(l_i'); l_i')$, $D(A(f_i), A(l_i); f_i)$, and $D(A(l_i'), A(l_i); f_i)$ (provided that these paths exist, since some of the six associated vertices of l_i, f_i, l_i' may coincide). During the prune operation examine if there exists a negative cycle in $l_i \cup f_i$, using Proposition 4.1. If it exists then stop, report that a negative cycle was found and output the cycle. Otherwise, compute $D(A(l_i'); l_i \cup f_i)$ and $D(A(f_i); l_i \cup f_i)$ using Lemma 4.2, and continue with the bypass operation resulting into a new node f_i', where $f_i' = (l_i \cup f_i) \cup l_i'$. Check again (using Proposition 4.1) if there exists a negative cycle in f_i' and if not, compute $D(A(f_i); f_i')$ using Lemma 4.1. Finally, and in the case where a negative cycle was not found, set $s_1(f_i') = s_1(f_i)$ and $s_2(f_i') = s_2(f_i)$.

Type-2: Suppose that leaf l_k performs a *SHUNT* operation, and f_j, the sibling of l_k, is an internal node of T^F (see Fig. 9). From previous executions, the following shortest paths are known: (i) $D(A(l_k); l_k)$; (ii) $D(A(l_k); f_k)$, $D(A(f_k); f_k)$, $D(A(f_j); f_k)$, $D(A(f_k), A(l_k); f_k)$, $D(A(f_j), A(l_k); f_k)$, and $D(A(f_k), A(f_j); f_k)$ (i.e. the shortest paths among all six vertices $A(l_k)$, $A(f_k)$ and $A(f_j)$ inside f_k); and (iii) $D(A(f_j); f_j)$, $D(A(f_i); f_j)$, $D(A(l_j); f_j)$, $D(A(f_j), A(l_j); f_j)$, $D(A(f_j), A(f_i); f_j)$ and $D(A(f_i), A(l_j); f_j)$ (i.e. the shortest paths among all six vertices $A(f_j)$, $A(f_i)$ and $A(l_j)$ inside f_j). (If some of the above ten associated vertices coincide, the corresponding shortest paths do not exist.) During the prune operation examine if there exists a negative cycle in $l_k \cup f_k$, using Proposition 4.1. If it exists then stop, report that a negative cycle was found and output the cycle. Otherwise, compute $D(A(f_j); l_k \cup f_k)$ and $D(A(f_k); l_k \cup f_k)$ using Lemma 4.2. After that continue with the bypass operation. This results into a new node f_{kj}, where $f_{kj} = (l_k \cup f_k) \cup f_j$. Check again (using Proposition 4.1) if there

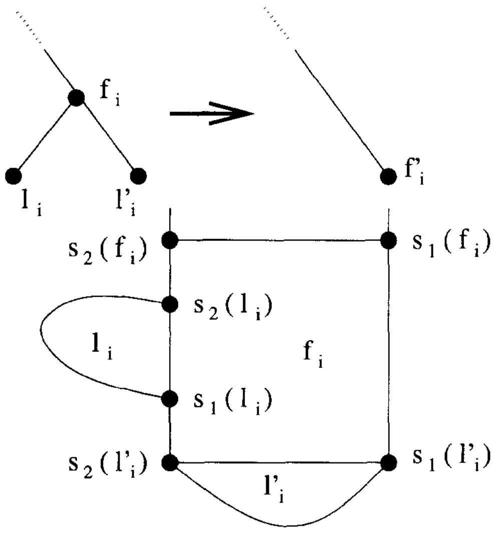

Fig. 8. The Type-1 *SHUNT* operation.

exists a negative cycle in f_{kj} and if not, compute $D(A(f_k); f_{kj})$, $D(A(l_j); f_{kj})$ and $D(A(f_k), A(l_j); f_{kj})$ using one application of Lemma 4.1. Then compute $D(A(f_i); f_{kj})$ and $D(A(f_i), A(f_k); f_{kj})$ using a second application of Lemma 4.1. Furthermore, compute $D(A(f_i), A(l_j); f_{kj})$ using Lemma 4.2. Finally, and in the case where a negative cycle was not found, set $s_1(f_{kj}) = s_1(f_k)$ and $s_2(f_{kj}) = s_2(f_k)$.

This completes the description of the *SHUNT* operations.

Lemma 4.3. *The SHUNT operations are correct and are executed in* O(1) *time.*

Proof. Consider first the *Type*-1 *SHUNT* operation. It is clear by Proposition 4.1 that if there is no negative cycle in any one of l_i, f_i, or l'_i, then a negative cycle (if exists) would be either in $l_i \cup f_i$, or in $f_i \cup l'_i$, or in f'_i. From the description of this *SHUNT* operation, it follows that the first and the third cases are correctly checked. Note however, that the second case is also checked implicitly in the third one. This is true because a negative cycle in $f_i \cup l'_i$ would involve $D(A(l'_i); f_i)$ and $D(A(l'_i); l'_i)$, according to Proposition 4.1. But notice that the length of any shortest path in $D(A(l'_i); l_i \cup f_i)$ is always less than or equal to the length of its corresponding shortest path in $D(A(l'_i); f_i)$. Therefore, if a negative cycle exists in any one of $l_i \cup f_i$, $f_i \cup l'_i$, or f'_i, then it is correctly detected. In the case that a negative cycle does not exist, observe that f'_i represents a set of faces whose removal does not disconnect G_o. Hence, it suffices for any further computation to find the shortest paths between the associated vertices of f'_i (inside f'_i), since by Proposition 4.1 a portion of a negative cycle that passes through f'_i would definitely involve these vertices as well as the shortest paths between them.

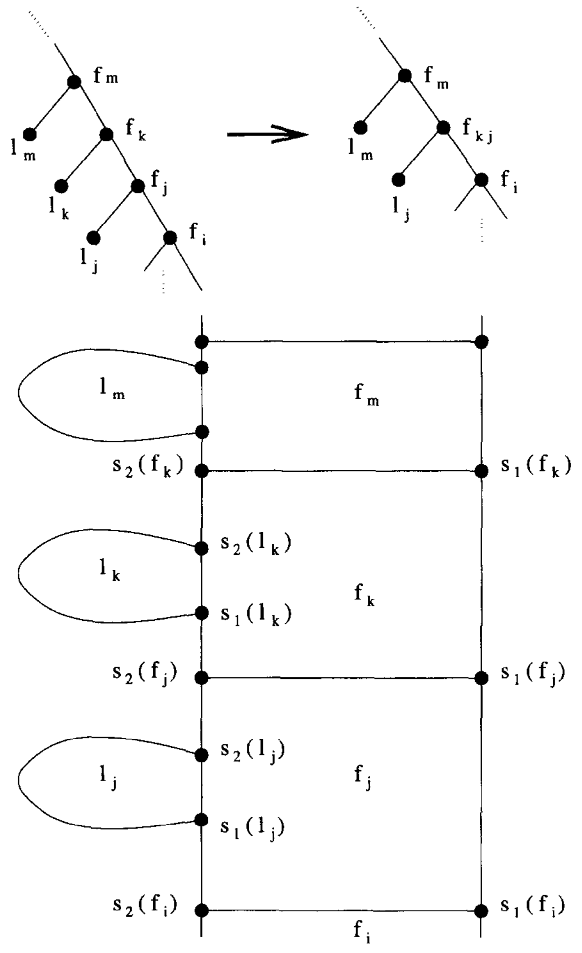

Fig. 9. The Type-2 *SHUNT* operation.

Consider now the *Type*-2 *SHUNT* operation. Assume as before that there are no negative cycles in any one of l_k, f_k and f_j. Using a similar argument as before, we can show that a negative cycle (if exists) is correctly detected in $l_k \cup f_k$, or in $f_k \cup f_j$, or in f_{kj}. If such a cycle does not exist, then observe that f_{kj} represents a set of faces whose removal disconnects G_o. Therefore, according to Proposition 4.1, a portion of a negative cycle which passes through f_{kj} would definitely pass through the associated vertices of it and/or through the associated vertices of the two subgraphs corresponding to its two children in the tree, and also would involve the shortest paths among them. Hence, in this case it is sufficient to compute shortest paths among the above mentioned associated vertices (inside f_{kj}).

The resource bound follows clearly by the description of the *SHUNT* operations, Proposition 4.1 and Lemmata 4.1 and 4.2. □

Now, we will show how to overcome the assumptions made in the beginning of this section and prepare G_o for the application of the parallel tree-contraction method using the *SHUNT* operations described above. We call this, the *initialization step* of our algorithm. This step includes construction of T^{F} and binarization, and initial computation of shortest paths in the subgraphs of G_o induced by each internal face of $\hat{G}_o$. The initialization step consists of the following stages.

1. Find an embedding $\hat{G}_o$ of G_o. Triangulate each interior face of $\hat{G}_o$ (if $\hat{G}_o$ is not already triangulated) by adding appropriate edges to $\hat{G}_o$ and corresponding arcs to G_o. Associate with these new arcs a very large cost (e.g. the sum of the absolute values of all arc costs in G_o) such that they will not be used by any shortest path. Then, construct the tree of faces T^{F} (which will be binary since every face is a triangle) and root T^{F} arbitrarily.

2. In each face (and in each subgraph of G_o induced by) f:

(a) Find the associated vertices $s_1(f)$ and $s_2(f)$, and compute the clockwise and counterclockwise distances from $s_1(f)$ to every other vertex in f (i.e. the sum of the costs of the arcs in clockwise and counterclockwise order).

(b) Compute the sets of shortest paths $D(A(f);f)$ and $D(A(f);p(f))$, where $p(f)$ is the face corresponding to the parent node of f in T^{F}.

(c) Let l be the face corresponding to the left child of f and let h be the face corresponding to the right child of f in T^{F}. Compute the following shortest paths: $D(A(f),A(h);f)$, $D(A(f),A(l);f)$ and $D(A(h),A(l);f)$.

This completes the description of the initialization step and the description of our algorithm. Let us call the algorithm presented in this section *Out_Neg_Cycle*. If G_o is not biconnected, apply *Out_Neg_Cycle* to every biconnected component.

Theorem 4.4. *Given an n-vertex outerplanar digraph G_o with real-valued arc costs, algorithm Out_Neg_Cycle detects and outputs a negative cycle in G_o, if it exists, in* $\mathrm{O}(\log n \log^* n)$ *time using* $\mathrm{O}(n/\log n \log^* n)$ *CREW PRAM processors and* $\mathrm{O}(n)$ *space. A sequential implementation of the algorithm runs in* $\mathrm{O}(n)$ *time.*

Proof. The correctness of the algorithm comes from Lemma 4.3. Now for the resource bounds we have that each main step (*SHUNT* operation) of the algorithm needs O(1) time and $\mathrm{O}(|T_\alpha^{\mathrm{F}}|)$ CREW PRAM processors by Lemma 4.3, where $|T_\alpha^{\mathrm{F}}|$ is the size of the tree of faces during the α-th phase. Hence, by [1], this results in a total of $\mathrm{O}(\log n)$ time using $\mathrm{O}(n/\log n)$ CREW PRAM processors. The bounds of the initialization step are as follows: Stage 1 needs $\mathrm{O}(\log n \log^* n)$ time and $\mathrm{O}(n/\log n \log^* n)$ EREW PRAM processors by [12]. Stage 2(a) can be done in $\mathrm{O}(\log n)$ time using $\mathrm{O}(n/\log n)$ EREW PRAM processors, using the algorithm of [9]. Finally Stages 2(b) and 2(c) can be easily done having the information of Stage 2(a). Note also that within the same bounds as those stated in the theorem, we can find the biconnected components of the input digraph [25]. The claimed sequential bound follows directly by the parallel ones. □

4.2.2. *The general case*

The algorithm for finding a negative cycle in a sparse digraph G is the following.

```
Algorithm Sparse_Neg_Cycle
BEGIN
  1. Find a hammock-on-ears decomposition of (the undirected version of) $G$ into $O(\tilde{\gamma})$ hammocks.
  2. Detect if there is any negative cycle in some hammock using algorithm Out_Neg_Cycle. If a negative cycle is found in any hammock, then output one such cycle and stop.
  3. Compress each hammock into an $O(1)$-sized graph such that the shortest paths between its attachment vertices are preserved and then compress $G$ into a digraph of size $O(\tilde{\gamma})$. Examine if there is any negative cycle in this graph.
  4. If a negative cycle is found, then output the cycle taking into account the subpaths contained in each hammock. Otherwise, output that there is no negative cycle in $G$.
END.
```

Theorem 4.5. *A negative cycle in an n-vertex sparse digraph G with real-valued arc costs, that can be decomposed into an asymptotically minimum number of* $O(\tilde{\gamma})$ *hammocks, can be found in* $O(\log^2 n)$ *time with* $O(n + M_s(\tilde{\gamma}))$ *CREW PRAM processors using* $O(n + \tilde{\gamma}^2)$ *space.*

Proof. The correctness of the algorithm as well as the space bound are clear. Note that Step 1 needs $O(\log n \log\log n)$ time using $O(n)$ processors by Theorem 3.2. Step 2 needs, for all hammocks, $O(\log n \log^* n)$ time using $O(n/\log n \log^* n)$ CREW PRAM processors by Theorem 4.4. The implementation of Step 3 is similar to the one described in Step 4 of algorithm *Sparse_Apsp*. The compression of a hammock needs $O(\log^2 n)$ time with $O(n)$ processors [36]. The detection of a negative cycle in the compressed digraph is performed by running the algorithm of [26], thus taking $O(\log^2 \tilde{\gamma})$ time and $O(M_s(\tilde{\gamma}))$ processors. Step 4 can be executed in $O(1)$ time using $O(n)$ processors. □

Corollary 4.1. *A negative cycle in an n-vertex planar digraph G with real-valued arc costs, that can be decomposed into an asymptoticaly minimum number of* $O(\tilde{\gamma})$ *hammocks, can be found in* $O(\log^2 n + \log^4 \tilde{\gamma})$ *time with* $O(n + \tilde{\gamma}^2/\log^4 \tilde{\gamma})$ *CREW PRAM processors using* $O(n+\tilde{\gamma}^2)$ *space. Sequentially, a negative cycle can be found in* $O(n + \tilde{\gamma}^{1.5} \log \tilde{\gamma})$ *time.*

Proof. The parallel bounds are achieved, if in Step 3 of algorithm *Sparse_Neg_Cycle* we run the algorithm of [6] (instead of the one in [26]). For the sequential implementation we have that: Step 1 is performed in $O(n)$ time (by running a straightforward sequential implementation of our algorithm presented in Section 3, or the algorithm of [16]). Step 2 needs $O(n)$ time by Theorem 4.4. The first part of Step 3 (hammock compression) is described in [17] and needs $O(n)$ time. For the second part (detection of a negative cycle in the compressed version of G), we run the algorithm of [34] which needs $O(\tilde{\gamma}^{1.5} \log \tilde{\gamma})$ time. Finally, Step 4 obviously runs in $O(n)$ time. The bounds follow. □

4.3. Computing all pairs reachability information

We treat the APR problem as a degenerated version of the corresponding APSP problem. (To every arc $\langle v, w\rangle$ of G assign a cost of 1, and if there is no arc $\langle w, v\rangle$ in G, then add it with cost ∞. Clearly, a vertex t is reachable by s iff their distance is not ∞.) Succinct encoding of reachability information can be stored into compact routing tables by defining for each arc $\langle v, w\rangle$ its compact label $R(v, w)$ as the set of vertices u such that there is a directed path from v to u with first arc $\langle v, w\rangle$. If we additionally want to output such paths, we must enforce a tie-breaking rule, to be applied when a vertex u belongs also to another set $R(v,z)$ and z is a neighbor of v. Therefore, by Theorems 4.1–4.3, we have the following.

Corollary 4.2. *Given an n-vertex sparse digraph G that can be decomposed into an asymptotically minimum number of* $O(\tilde{\gamma})$ *hammocks, we can find on a CREW PRAM*: (i) APR *information (encoded into compact routing tables) in* $O(\log^2 n)$ *time using* $O(n\tilde{\gamma} + M_r(\tilde{\gamma}))$ *processors and* $O(n\tilde{\gamma})$ *space*; (ii) APR *information (using alternative encodings) in* $O(\log^2 n)$ *time by employing* $O(n+M_r(\tilde{\gamma}))$ *processors and using* $O(n+\tilde{\gamma}^2)$ *space.*

Corollary 4.3. *Given an n-vertex planar digraph G that can be decomposed into an asymptotically minimum number of* $O(\tilde{\gamma})$ *hammocks, we can find on a CREW PRAM*: (i) APR *information (encoded into compact routing tables) in* $O(\log^2 n+\log^4 \tilde{\gamma})$ *time using* $O(n\tilde{\gamma})$ *processors and* $O(n\tilde{\gamma})$ *space*; (ii) APR *information (using alternative encodings) in* $O(\log^2 n + \log^4 \tilde{\gamma})$ *time by employing* $O(n + \tilde{\gamma}^2/\log^4 \tilde{\gamma})$ *processors and using* $O(n + \tilde{\gamma}^2)$ *space.*

5. Conclusions and further results

We have presented here a technique, called hammock-on-ears decomposition, that decomposes any graph into outerplanar subgraphs (called hammocks) satisfying certain separator properties. The interesting feature of this technique is that it can be used to design algorithms parameterized in terms of certain topological properties of the input (di)graph, i.e. in terms of $\tilde{\gamma}$. We have also shown that the hammock-on-ears decomposition technique is well-suited for parallel computation and can be advantageously used to improve the parallel bounds for all pairs shortest paths and related problems. Moreover, we managed to beat the transitive closure bottleneck (that appears to be inherent on these problems) for all sparse digraphs with a small value of $\tilde{\gamma}$. There are certain classes of graphs for which the value of $\tilde{\gamma}$ is small. Examples are: outerplanar graphs, graphs which satisfy the k-interval property [19] and graphs with small genus which, when embedded in their genus surface, have a small number of faces that cover all vertices.

We can achieve further improvements to the applications discussed in this paper, in the case where the input digraph G is provided with a balanced $O(n^{\mu})$-separator decomposition, $0 < \mu < 1$. Although it is known that a digraph with genus $0 < \gamma < n$ has a separator of size $O(\sqrt{n\gamma})$ [13, 21] and that such a separator can be computed in $O(n)$ time without an embedding of G to be provided [13], it is not known yet how to compute such a separator in NC. But there are some families of graphs for which a balanced $O(n^{\mu})$-separator decomposition can be computed in NC. Consider for example the d-dimensional grid which has a trivial balanced $O(n^{(d-1)/d})$-separator decomposition. Also in [35] it is shown that the class of k-overlap graphs embedded in d dimensions (which includes planar graphs) have a separator of size $O(k^{1/d}n^{(d-1)/d})$ that can be computed in NC. Hence, in the case where an $O(n^{\mu})$-separator decomposition is either provided with the input or it can be computed, we have the following results.

Corollary 5.1. *Let G be an n-vertex digraph with real-valued arc costs (but no negative cycles). Let also G be provided with a balanced* $O(n^{\mu})$*-separator decomposition and can be decomposed into an asymptotically minimum number of* $O(\tilde{\gamma})$ *hammocks. Then, we can find the following on a CREW PRAM:* (i) APSP *information (encoded into compact routing tables) in* $O(\log^2 n + \log^3 \tilde{\gamma})$ *time using* $O(n\tilde{\gamma} + (\tilde{\gamma}^2 + \tilde{\gamma}^{2\mu+1})/\log^3 \tilde{\gamma})$ *processors and* $O(n\tilde{\gamma})$ *space;* (ii) APSP *information (using alternative encodings) in* $O(\log^2 n + \log^3 \tilde{\gamma})$ *time by employing* $O(n + (\tilde{\gamma}^2 + \tilde{\gamma}^{2\mu+1})/\log^3 \tilde{\gamma})$ *processors and using* $O(n + \tilde{\gamma}^2)$ *space.*

Corollary 5.2. *Let G be an n-vertex digraph with real-valued arc costs. Assume that G is provided with a balanced* $O(n^{\mu})$*-separator decomposition and that it can be decomposed into an asymptotically minimum number of* $O(\tilde{\gamma})$ *hammocks. Then, there is an algorithm for finding a negative cycle in G in* $O(\log^2 n + \log^3 \tilde{\gamma})$ *time by employing* $O(n + (\tilde{\gamma}^2 + \tilde{\gamma}^{2\mu+1})/\log^3 \tilde{\gamma})$ *CREW PRAM processors. A sequential implementation of this algorithm runs in* $O(n + \min\{\tilde{\gamma}^{3\mu} + \tilde{\gamma}^{1+\mu} \log \tilde{\gamma}, \tilde{\gamma}^2\})$ *time.*

Corollary 5.3. *Let G be an n-vertex digraph that can be decomposed into an asymptotically minimum number of* $O(\tilde{\gamma})$ *hammocks. Let also G be provided with an* $O(n^{\mu})$*-separator decomposition. Then,* APR *information can be computed in the same resource bounds with those given for* APSP *in Corollary* 5.1.

The hammock-on-ears decomposition technique has also been used recently in the solution of the dynamic version of the APSP problem [15]. We believe that the technique has potential and will find applications to other problems, too.

Acknowledgements

We are indebted to the anonymous referees for their careful reading of the paper and their helpful and constructive comments which improved the presentation. We are also grateful to Dimitris Sofotassios for pointing out the geometric approach to the intervals problem, and to Hristo Djidjev, Greg Frederickson, Christos Papadimitriou and Moti Yung for their insightful comments and encouragement.

References

[1] K. Abrahamson, N. Dadoun, D. Kirkpatrick and T. Przytycka, A simple parallel tree contraction algorithm, *J. Algorithms* **10** (1989) 287–302.

[2] A. Aggarwal, B. Chazelle, L. Guibas, C. Ó'Dúnlaing and C. Yap, Parallel computational geometry, *Algorithmica* **3**(3) (1988) 293–328.

[3] R. Ahuja, T. Magnanti and J. Orlin, *Network Flows* (Prentice-Hall, Englewood Cliffs, NJ, 1993).

[4] S. Chaudhuri and C. Zaroliagis, Optimal parallel shortest paths in small treewidth digraphs, *Proc. 3rd European Symp. on Algorithms*, Lecture Notes in Computer Science, Vol. 979 (Springer, Berlin, 1995) 31–45.

[5] K.W. Chong and T.W. Lam, Finding connected components in $O(\log n \log\log n)$ time on an EREW PRAM, *Proc. 4th ACM-SIAM Symp. on Discrete Algorithms* (1993) 11–20.

[6] E. Cohen, Efficient parallel shortest-paths in digraphs with a separator decomposition, *Proc. 5th ACM Symp. on Parallel Algorithms and Architectures* (1993) 57–67.

[7] R. Cole, Parallel merge sort, *Proc. 27th IEEE Symp. on Foundations of Computer Science* (1986) 511–516.

[8] R. Cole and U. Vishkin, Deterministic coin tossing with applications to optimal parallel list ranking, *Inform. and Control* **70** (1986) 32–53.

[9] R. Cole and U. Vishkin, Approximate parallel scheduling. Part I: the basic technique with applications to optimal parallel list ranking in logarithmic time, *SIAM J. Comput.* **17**(1) (1989) 128–142.

[10] D. Coppersmith and S. Winograd, Matrix multiplication via arithmetic progressions, *J. Symbol. Comput.* **9**(3) (1990) 251–280.

[11] T. Cormen, C. Leiserson and R. Rivest, *Introduction to Algorithms* (MIT Press, Cambridge, MA, McGraw-Hill, New York, 1990).

[12] K. Diks, T. Hagerup and W. Rytter, Optimal parallel algorithms for the recognition and colouring of outerplanar graphs, *Proc. 15th Symp. on Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science, Vol. 379 (Springer, Berlin, 1989) 207–217.

[13] H. Djidjev, A linear algorithm for partitioning graphs of fixed genus, *SERDICA* **11** (1985) 369–387.

[14] H. Djidjev, G. Pantziou and C. Zaroliagis, Computing shortest paths and distances in planar graphs, *Proc. 18th Internat. Coll. on Automata, Languages and Programming*, Lecture Notes in Computer Science, Vol. 510 (Springer, Berlin, 1991) 327–338.

[15] H. Djidjev, G. Pantziou and C. Zaroliagis, On-line and dynamic algorithms for shortest path problems, *Proc. 12th Symp. on Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science, Vol. 900 (Springer, Berlin, 1995) 193–204.

[16] G.N. Frederickson, Using cellular graph embeddings in solving all pairs shortest path problems, *Proc. 30th IEEE Symp. on Foundations of Computer Science* (1989) 448–453; also *J. Algorithms*, to appear.

[17] G.N. Frederickson, Planar graph decomposition and all pairs shortest paths, *J. ACM* **38**(1) (1991) 162–204.

[18] G.N. Frederickson, Searching among intervals and compact routing tables, *Proc. 20th Internat. Coll. on Automata, Languages and Programming*, Lecture Notes in Computer Science, Vol. 700 (Springer, Berlin, 1993) 28–39.

[19] G.N. Frederickson and R. Janardan, Designing networks with compact routing tables, *Algorithmica* **3** (1988) 171–190.

[20] H. Gazit and G. Miller, A deterministic parallel algorithm for finding a separator in planar graphs, Technical Report CMU-CS-91-103, Carnegie-Mellon University, 1991.
[21] J. Gilbert, J. Hutchinson and R. Tarjan, A separator theorem for graphs of bounded genus, *J. Algorithms* **5** (1984) 391–407.
[22] A. Goldberg, S. Plotkin and G. Shannon, Parallel symmetry-breaking in sparse graphs, *SIAM J. Discrete Math.* **1**(4) (1988) 434–446.
[23] M. Goodrich, Intersecting line segments in parallel with an output-sensitive number of processors, *Proc. 1st ACM Symp. on Parallel Algorithms and Architectures* (1989) 127–136.
[24] J.L. Gross and T.W. Tucker, *Topological Graph Theory* (Wiley, New York, 1987).
[25] T. Hagerup, Optimal parallel algorithms for planar graphs, *Inform. and Comput.* **84** (1990) 71–96.
[26] Y. Han, V. Pan and J. Reif, Efficient parallel algorithms for computing all pair shortest paths in directed graphs, *Proc. 4th ACM Symp. on Parallel Algorithms and Architectures* (1992) 353–362.
[27] F. Harary, *Graph Theory* (Addison-Wesley, Reading, MA, 1969).
[28] J. JáJá, *An Introduction to Parallel Algorithms* (Addison-Wesley, Reading, MA, 1992).
[29] R. Karp and V. Ramachandran, Parallel algorithms for shared-memory machines, in: J. van Leeuwen, ed., *Handbook of Theoretical Computer Science*, Vol. A (Elsevier, Amsterdam, 1990) 869–941.
[30] E.L. Lawler, *Combinatorial Optimization: Networks and Matroids* (Holt, Rinehart and Winston, New York, 1976).
[31] Y. Maon, B. Schieber and U. Vishkin, Parallel ear decomposition search (EDS) and st-numbering in graphs, *Theoret. Comput. Sci.* **47** (1986) 277–298.
[32] E.M. McCreight, Priority search trees, *SIAM J. Comput.* **14** (1985) 257–276.
[33] K. Mehlhorn, *Data Structures and Algorithms 3: Multidimensional Searching and Computational Geometry* (Springer, Berlin, 1984).
[34] K. Mehlhorn and B. Schmidt, A single source shortest path algorithm for graphs with separators, *Proc. Conf. on Fundamentals of Computation Theory*, Lecture Notes in Computer Science, Vol. 158 (Springer, Berlin, 1983) 302–309.
[35] G. Miller, S.H. Teng and S. Vavasis, A unified geometric approach to graph separators, *Proc. 32nd IEEE Symp. on Foundations of Computer Science* (1991) 538–547.
[36] G. Pantziou, P. Spirakis and C. Zaroliagis, Efficient parallel algorithms for shortest paths in planar digraphs, *BIT* **32** (1992) 215–236.
[37] T. Reps, M. Sagiv and S. Horwitz, Interprocedural dataflow analysis via graph reachability, Technical Report TR 94-14, Datalogisk Institut, University of Copenhagen, 1994.
[38] Y. Shiloach and U. Vishkin, An O($\log n$) parallel connectivity algorithm, *J. Algorithms* **3** (1982) 57–67.
[39] P. Spirakis and A. Tsakalidis, A very fast, practical algorithm for finding a negative cycle in a digraph, *Proc. 13th Internat. Coll. on Automata, Languages and Programming*, Lecture Notes in Computer Science, Vol. 226 (Springer, Berlin, 1986) 397–406.
[40] C. Thomassen, The graph genus problem is NP-complete, *J. Algorithms* **10** (1989) 568–576.
[41] D. Yellin and R. Strom, INC: a language for incremental computations, *ACM Trans. Prog. Lang. Systems* **13**(2) (1991) 211–236.
[42] J. van Leeuwen and R. Tan, Computer networks with compact routing tables, in: G. Rozenberg and A. Salomaa, ed., *The Book of L* (Springer, Berlin, 1986) 259–273.