11-1993

On-Line and Dynamic Shortest Paths Through Graph Decompositions

Hristo N. Djidjev
Rice University

Grammati E. Pantziou
Dartmouth College

Christos D. Zaroliagis
Computer Technology Institute, Patras, Greece

ON-LINE AND DYNAMIC SHORTEST PATHS THROUGH GRAPH DECOMPOSITIONS

Hristo N. Djidjev
Grammati E. Pantziou
Christos D. Zaroliagis

Technical Report PCS-TR93-200

11/93

On-line and Dynamic Shortest Paths through Graph Decompositions*

HRISTO N. DJIDJEV [1] GRAMMATI E. PANTZIOU [2,3]
CHRISTOS D. ZAROLIAGIS [3]

(1) Department of Computer Science, Rice University, P.O. Box 1892, Houston, TX 77251, USA
(2) Department of Mathematics and Computer Science, Dartmouth College, Hanover NH 03755, USA
(3) Computer Technology Institute, P.O. Box 1122, 26110 Patras, Greece

Abstract

We give here fast and efficient solutions to a fundamental underlying problem, encountered frequently in the development of many dynamic real-world systems. This problem concerns the maintainance of all pairs shortest path information in a planar digraph. We give both sequential and parallel algorithms, that exploit the topology of the input digraph G, and which can also detect a negative cycle, either if it exists or it is created during dynamic changes in G. The parallel algorithms presented here are the first known ones for solving this problem. Our results can be extended to hold for digraphs with genus $g(n) = o(n)$, where n is the number of vertices of the digraph.

1 Introduction

Let G be an n-vertex digraph with real valued edge costs but no negative cycles. The *length* of a path p in G is the sum of the costs of all edges of p and the *distance* between a pair v, w of vertices of G is the minimum length of a path between v and w. The path of minimum length between v, w is called a *shortest path* between v and w. Finding shortest path information in graphs is a very important and intensively studied graph problem with many applications. Recent papers [4, 7, 12, 13, 14, 15, 17, 22, 27, 28, 29] investigate different versions of the problem according to the structure of the input graph and the model of computation. All of the above-mentioned results, however, relate to the static version of the problem, i.e. the graph and the costs on its edges do not change over time. In contrast, we consider here a *dynamic environment*, where edges can be deleted and their costs can be modified. More precisely, we investigate the following *on-line and dynamic shortest path problem:* given G (as above), build a data structure (i.e. preprocess G so as to reduce both time and space) in order to be able to answer very fast on-line single-pair and/or single-source shortest path or distance queries. Also the data structure should be efficiently updated and maintain shortest path information after a modification of G.

*This work is partially supported by the EEC ESPRIT Basic Reasearch Action No. 7141 (ALCOM II) and by the Ministry of Education of Greece. The work of the second author is also partially supported by the NSF postdoctoral fellowship No. CDA-9211155. Email: hristo@cs.rice.edu, pantziou@cs.dartmouth.edu, zaroliag@grpatvx1.bitnet

The dynamic version of the shortest paths problem has applications in the so called *vehicle routing* problem. Assume that you are in a vehicle located somewhere in the traffic network of a city, and you want to know at any time the shortest route to the nearest hospital, drugstore, hotel, etc, or to find the shortest route or distance to a specific place. Note that the underlying traffic network may change dynamically: some roads may be closed (because of works or accidents), certain roads may change behaviour at rush hours, or some other ones may change direction. This problem gives rise to the development of a software system loaded to a very fast computer, where a number of operators receive on-line queries from the drivers, get the appropriate answers and transmit them back to the drivers.

There has been a growing interest in dynamic problems in the recent years [2, 10, 18, 19, 30]. The goal is to design efficient data structures that are not only capable of answering a series of queries, but also can easily be updated after a modification of the input graph. Such an approach has immediate applications to a variety of problems which are of both theoretical and practical value. More precisely, dynamic algorithms for graph problems have applications to simulation of traffic networks, high level languages for incremental computations, incremental data flow analysis, interactive network design, maintainance of maximum flow in a network [3, 36, 37, 38], just to name a few.

There are a few previously known algorithms for the dynamic shortest path problem. For general digraphs, the best previous algorithms in the case of updating the data structure after edge insertions/deletions were due to [9] and require $O(n^2)$ update time after an edge insertion and $O(n^2 \log n)$ update time after an edge deletion. Some improvements of these algorithms have been achieved in [2] with respect to the amortized cost of a sequence of edge insertions, if the edge costs are integers. For the case of planar digraphs the best dynamic algorithms are due to [11] for the case of edge cost updates. The preprocessing time and space is $O(n \log n)$ ($O(n)$ space can be achieved, if the computation is restricted to finding distances only.) A single-pair query can be answered in $O(n)$ time, while a single-source query takes $O(n\sqrt{\log\log n})$ time. An update operation to this data structure, after an edge cost modification or deletion, can be performed in $O(\log^3 n)$ time. In parallel computation we are not aware of any previous results related to dynamic structures for maintaining shortest path information in the case of edge cost updates. On the other hand, in parallel computation efficient data structures for answering very fast on-line shortest path or distance queries have been proposed in [7], but they do not support dynamization. (We should mention here that much of the recent work in either static [4, 27] or dynamic algorithms [11] for shortest paths, is based on the recursive separator decomposition idea introduced by Frederickson [12] and Lingas [28], in sequential and parallel computation respectively.)

In this paper, we give efficient sequential and parallel algorithms for solving the on-line and dynamic shortest path problem, in the case where the input digraph is planar. More precisely, for the sequential model of computation, we have achieved the following results which are clear improvements over the best previous ones, in all cases where $q = o(n)$. Here q is the minimum number of faces that cover all vertices of the planar digraph among all embeddings of the graph. (Note that $q = 1$ if the digraph is outerplanar.)

Theorem 1 *Given an n-vertex planar digraph G with real-valued edge costs but no negative cycles, there exists an algorithm for the on-line and dynamic shortest path problem on G that supports edge cost modification and edge deletion with the following performance characteristics: (i) preprocessing time and space $O(n + q \log q)$; (ii) single-pair distance query time $O(q + \log n)$; (iii) single-pair shortest path query time $O(L + q + \log n)$ (where L is the number of edges of the path); (iv)* *single-source shortest path tree query time $O(n + q\sqrt{\log\log q})$; (v) update time (after an edge cost modification or edge deletion) $O(\log n + \log^3 q)$. In the case where the computation is restricted to finding distances only the space can be reduced to $O(n)$.*

As it can be derived by [8, 23], the *expected value* for q is $O(1)$ for random graphs which are planar. Then, our algorithms achieve the following expected performance: $O(n)$ preprocessing time and space, $O(\log n)$ (resp. $O(L + \log n)$) distance (resp. shortest path) query time, $O(n)$ single-source query time, and $O(\log n)$ update time. For comparison, see the best previous results of [11] stated above.

In the CREW PRAM model of parallel computation our result is the following.

Theorem 2 *Given an n-vertex planar digraph G with real-valued edge costs but no negative cycles, there exists an algorithm for the on-line and dynamic shortest path problem that supports edge cost modification and edge deletion with the following performance characteristics: (i) preprocessing time $O(\log^2 n + \log^4 q)$ using $O(n + q^2/\log^3 q)$ processors and space $O(n + q^2)$; (ii) single-pair shortest distance query time $O(\log n)$ using a single processor; (iii) single-pair shortest path query time $O(L + \log n)$ (where L is the number of edges of the path), using a single processor; (iv) single-source shortest-path tree query in $O(n)$ time using a single processor; (v) update time (after an edge cost modification or edge deletion) $O(\log n + \log^2 q)$ using $O(\log n + q^2/\log q)$ processors.*

There are no comparative previous results for the parallel case. Similarly to sequential computation, our parallel algorithms have also very efficient expected performance. Note that the CREW PRAM is a very popular and well-established model in the case of shared memory parallel machines [20, 21]. Moreover, as it has been proved in [24, 25, 26], the PRAM provides an excellent model for fault-tolerant parallel computing. Our results can be directly applied to any distributed memory parallel machine model (using e.g. [1]), even in the case of processor faults [26].

Our solution is based on the following ideas:

(a) The input planar digraph is decomposed into a small number, $O(q)$, of outerplanar subgraphs (called *hammocks*) satisfying certain separator conditions [14, 29]. Note that an embedding of the input digraph in order to achieve such a (hammock) decomposition does not need to be provided by the input as it is proved in [13, 22].

(b) A multi-level decomposition strategy is employed for the efficient solution of the problem for the case of *outerplanar* digraphs that uses the so-called *beneficial separators*. Our separator decomposition is adaptive in the sense that at any step, the computation of the next level beneficial separators can be tuned up so as a certain parameter will never exceed a predefined value.

(c) A data structure is constructed during the decomposition of the outerplanar digraph and is updated after each edge cost modification. This data structure contains information about the shortest paths between properly chosen $O(n)$ pairs of vertices. It also has the properties that the shortest path between any pair of vertices is a composition of $O(\log n)$ of the predefined paths and that any edge of the graph belongs to $O(\log n)$ of those paths (n is the size of the outerplanar digraph).

We mention also the following extensions and generalizations to our results discussed in the paper.

(i) Our algorithms can detect a negative cycle, either if it exists in the initial graph, or if it is created after an edge cost modification.

(ii) Using the ideas of [13, 22], our results can be extended to hold for any digraph with genus bounded by any function $g(n) = o(n)$. In such a case an embedding of the graph does not need to be provided by the input.

(iii) Although our algorithms do not directly support edge insertion, they are so fast that even if they are run from scratch, they still provide better results compared with those of [9].

(iv) Our results find applications in the problem of the maintainance of the all pairs reachability information in a planar digraph G. We can achieve further improvements over the recent results of [33] for the case of edge insertions/deletions, by treating the reachability problem (i.e. whether a vertex v is reachable from a vertex z via a directed path) as a degenerated version of the shortest path problem.

The paper is organized as follows. In section 2 we give some definitions, preliminary results, and basic algorithms used later in the paper. In section 3 we consider sequential and parallel algorithms for outerplanar digraphs and in section 4 we obtain our basic results for planar digraphs. In section 5 we consider generalizations and applications of our results.

2 Preliminaries and Basic Algorithms

Let $G = (V(G), E(G))$ be a planar digraph with real edge weights but no negative cycles. A *separation pair* is a pair of vertices whose removal separates G into two disjoint subgraphs. Note that if G is outerplanar and biconnected, a separation pair is either an edge or a pair of vertices belonging to the same interior face. A *separator* S for a graph G is a pair of sets $(V(S), D(S))$ where $D(S)$ is a set of separation pairs and $V(S)$ is the set of the endpoints of $D(S)$. A $k(n)$*-separator* in an n-vertex graph G is a separator S such that $|V(S)| = k(n)$. A separation pair in an outerplanar biconnected graph G_o is called *suitable* if separates G_o into two subgraphs G_1 and G_2 such that $|V(G_1)| = \varepsilon|V(G_o)|$ and $|V(G_2)| = (1-\varepsilon)|V(G_o)|$ for some previously fixed $(1/2) < \varepsilon \leq (2/3)$. A $(k(n), c)$*-beneficial separator* S in G_o ($|V(G_o)| = n$) is a $k(n)$-separator in which each subgraph H produced after the removal of S satisfies the following two conditions: (B1) $|V(H)| \leq 4n/k(n)$ and (B2) $|(V(H) \times V(H)) \cap D(S)| \leq c$, for some $c \geq 3$. We will call the separation vertices (pairs) of $V(H) \cap V(S)$ $((V(H) \times V(H)) \cap D(S))$ separation vertices (pairs) *attached to* H. A tree is called *convergent (divergent)* if the edges of the tree point from a node to its parent (children). A *DSP* (divergent-shortest-path) tree rooted at some vertex x in a digraph G, denoted as DSP(x, G), is a divergent tree of shortest paths such that: (P1) for each $v \in G$, the path from x to v in DSP(x, G) is the shortest one in G; and (P2) DSP(x, G) includes any vertex of G reachable from x.

For each (directed) edge $\langle v, w\rangle$ of G we define the *label* $S(v, w)$ of the edge as follows: $S(v, w) = \{u | \langle v, w\rangle$ is the first edge in a shortest path from v to $u\}$. Each $S(v, w)$ is described as a union of a minimum number of subintervals of $[1, n]$, assuming $V = \{1, 2, ..., n\}$. (A subinterval is allowed to wrap around n.) Edge labels are used in the succinct encoding of all pairs shortest paths information in what it is called *compact routing tables* [14, 35]. Compact routing tables have been used for keeping shortest path information (edge labels) in a space-efficient way, either in sequential [13, 14] or in parallel computation [22, 29]. Here, we shall consider their use only for outerplanar digraphs. If G is outerplanar then each $S(v, w)$ is a single interval $[a, b]$ [16]. It is clear that the total size of a compact routing table for an outerplanar graph is $O(n)$. Computation of edge labels is very useful for finding any shortest path information. Given edge labeling information in an n-vertex outerplanar graph G_o, then a convergent or divergent tree of shortest paths rooted at some vertex x can be constructed either in $O(n)$ sequential time, or in $O(\log n)$ parallel time using $O(n/\log n)$ processors on an EREW PRAM [14, 29]. Also, given an n-vertex outerplanar digraph G_o along with divergent shortest path trees rooted at a constant number of distinguished vertices a_i, a *compressed version* $C(G_o)$ of G_o is an outerplanar digraph of $O(1)$ size and can be generated in either $O(n)$ sequential time, or in $O(\log n)$ parallel time using $O(n)$ CREW PRAM processors.

Furthermore, for every pair a_i, a_j in G_o $(i \neq j)$, if a shortest path from a_i to a_j exists in G_o, then there is a corresponding (compressed) one in $C(G_o)$ of equal cost [14, 29].

Let G_o be an outerplanar digraph provided with a $(k(n), c)$-beneficial separator S $(c = O(1))$. Let also $SR(G_o)$ be the graph obtained from G_o as follows: remove S from G_o, substitute each subgraph produced by its compressed version and then join again the subgraphs. We call $SR(G_o)$ the *sparse representative* of G_o. Clearly, $SR(G_o)$ has size $O(k(n))$.

A *hammock decomposition* is a decomposition of G into certain outerplanar digraphs called *hammocks*. This decomposition is defined relative to a given set of faces that cover all vertices of G. Let q be the minimum number of such faces (among all embeddings of G), i.e. q is the minimum cardinality among all the so called *face-on-vertex coverings* of G. It has been proved in [14, 29] that a planar digraph G can be decomposed into $O(q)$ hammocks either in $O(n)$ sequential time, or in $O(\log n \log^* n)$ parallel time using $O(n)$ CREW PRAM processors. Also, by [13, 22], we have that an embedding of G does not need to be provided by the input in order to compute a hammock decomposition of $O(q)$ hammocks. Hammocks satisfy certain separator conditions and hammock decomposition employs 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) the hammock decomposition spans all the edges of G, i.e. each edge belongs to only one hammock; and (iii) the number of hammocks produced is order of the minimum possible among all possible decompositions. This decomposition allows us to *reduce* the solution of a given problem Π on G, into the solution of Π in an outerplanar digraph.

In parallel computation we will make use of the following recent result [4]: given an n-vertex planar digraph, we can compute shortest path trees from s sources in $O(\log^2 n)$ time using $O(sn/\log n)$ EREW PRAM processors. A preprocessing phase is needed which takes $O(\log^4 n)$ time using $O(n^{1.5}/\log^3 n)$ CREW PRAM processors using $O(n^2)$ space.

In the sequel, we can assume w.l.o.g. that G_o is biconnected and the vertices are named consecutively around the exterior face. This does not affect shortest path information for three reasons: (i) renaming is (clearly) not a problem; (ii) the edges added to enforce biconnectivity, can be assigned a very large weight, e.g. the sum of the absolute values of all edge costs of G_o, such that they will not be used by any shortest path; and (iii) the conversion can be done in $O(n)$ sequential time, or in $O(\log n \log^* n)$ parallel time using $O(n/\log n \log^* n)$ CREW PRAM processors [14, 29]. As we will see, these bounds does not affect our preprocessing algorithms.

2.1 Computing a beneficial separator

Suppose we are given an outerplanar digraph $G_o = (V(G_o), E(G_o))$, where $n = |V(G_o)|$. The algorithm for computing a $(k(n), c)$-beneficial separator, $2 \leq k(n) \leq \beta \cdot n^{\mu}$, in G_o, where β, μ are constants, and $0 < \mu < 1$, is as follows. (In the sequel, with the expression "for any $k(n)$", we will mean "for any value of $k(n)$ in the range $[2, \beta \cdot n^{\mu}]$".)

```
PROCEDURE Beneficial_Separator($G_o, k(n), c, S$)
BEGIN
    1. Convert $G_o$ into an undirected outerplanar graph $G_u$ and then triangulate each interior face
of $G_u$. Let initially $V(S) = D(S) = \emptyset$.
    2. If the number $n_{sep}$ of separation pairs attached to $G_o$ is less or equal to $c$, then let $p = \{p_1, p_2\}$
be a suitable separation pair in $G_o$ which separates $G_u$ into two subgraphs $G_1$ and $G_2$ with no more
than $2n/3$ vertices each. Otherwise ($n_{sep} > c$), let $p = \{p_1, p_2\}$ be a separation pair that separates
```

G_o into subgraphs G_1 and G_2 each containing no more than 2/3 of the number of separation pairs attached to G.

3. Let $V(S) = V(S) \cup \{p_1, p_2\}$ and $D(S) = D(S) \cup \{p\}$.

4. Run steps 2 and 3 recursively on each one of G_1 and G_2, until $k(n)$ separation pairs are produced.

5. Remove all edges added for the triangulation of G_u in step 2.

END.

Lemma 2.1 *Procedure Beneficial_Separator($G_o, k(n), c, S$) correctly finds a $(k(n), c)$-beneficial separator S of an n-vertex biconnencted outerplanar digraph G_o either in $O(n)$ sequential time, or in $O(\log n \log k(n)))$ parallel time using $O(n)$ CREW PRAM processors.*

Proof: In the appendix. ∎

2.2 Adaptive separator decomposition

The following algorithm generates an adaptive separator decomposition in G_o (by finding successive $(k(n), c)$-beneficial separators in a recursive way) and builds the two main data structures: the *separator tree*, $S(G_o)$, which is a tree of maximum degree $k(n)$ and binary trees T_S used to support binary search in G_o.

ALGORITHM Sep_Tree($G_o, k(n)$)

BEGIN

1. Run procedure Beneficial_Separator($G_o, k(n), 3, S$) to find in G_o a $(k(n), 3)$-beneficial separator S.

2. Create a binary search tree T_S with nodes the separation pairs of S. This can be done at each iteration of procedure Beneficial_Separator as follows. Create a node v for each separation pair $p \in S$ produced. Assume that p separates a subgraph G of G_o into two subgraphs $G_1(p)$ and $G_2(p)$. Associate with v the separation pair p and the subgraph G. The children of v are those nodes v_1 and v_2 that correspond to the separation pairs p' and p'' separating $G_1(p)$ and $G_2(p)$ respectively.

3. For each component C of $G_o - S$ run steps 1 to 4 recursively to create a separator tree $S(C)$ corresponding to C, if $|V(C)| > K_0$, or let $S(C) = C$, if $|V(C)| \leq K_0$, for some appropriately chosen constant K_0.

4. Create a separator tree $S(G_o)$ rooted at a new node v_S with children the roots of the subtrees constructed in step 3. Associate with v_S a pointer to T_S.

END.

Let $d(k(n))$ be the depth[1] of the separator tree.

Lemma 2.2 *Algorithm Sep_Tree($G_o, k(n)$) can be implemented to run in $O(n)$ time. A parallel CREW PRAM implementation of the algorithm runs in $O(\log^2 n)$ time using $O(n)$ processors. The space required is $O(n)$.*

Proof (sketch): For any beneficial separator S and any vertex of T_S there exists a distinct separation pair of G_o. Thus the total number of vertices in all trees T_S is $O(n)$. Furthermore, as the number of vertices in $S(G_o)$ equals the number of trees T_S, then $|V(S(G_o))| = O(n)$ and the space

[1]Note for example that, if $k(n) = O(1)$ then $d(k(n)) = O(\log n)$; if $k(n) = O(\log n)$, then $d(k(n)) = O(\log n / \log \log n)$; if $k(n) = O(\sqrt{n})$, then $d(k(n)) = O(\log \log n)$.

required by the algorithm is $O(n)$. Therefore, steps 2 and 4 can be implemented to run in $O(n)$ time, in total. In order to achieve $O(n)$ upper bound for the total running time $S(n)$ for all executions of step 1, we can construct in algorithm Beneficial_Separator the dual graph of G_o (excluding the outer face), which is a tree, and then use the data structure of [34] for linking and cutting trees that will reduce the time for finding one separation pair in algorithm Beneficial_Separator to $O(\log n)$. Then the recurrence for $S(n)$ is $S(n) \leq k(n)S(n/k(n)) + O(k(n)\log n)$ with solution $S(n) = O(n)$. For the parallel implementation we have the following. The recursion depth is $d(k(n))$. Finding a $(k(n), 3)$-beneficial separator S in step 1 takes $O(\log n \log k(n))$ parallel CREW PRAM time, using $O(n)$ processors by lemma 2.1. Steps 2 and 4 clearly need $O(\log k(n))$ parallel time and $O(n)$ processors. Therefore, the parallel CREW PRAM time satisfies the recurrence $T_p(n) = T_p(n/k(n)) + O(\log n \log k(n))$ which gives $T_p(n) = O(\log^2 n)$ for any $k(n)$. ∎

Remark 2.1: Note that the total searching depth of the separator tree $S(G_o)$ along with its associated T_S trees is $O(\log n)$. To see this, just replace every star subgraph of $S(G_o)$ consisting of a node along with its children, by the corresponding T_S tree. Clearly the resulting tree will have depth satisfying the recurrence $H(n) = H(n/k(n)) + O(\log k(n))$, which gives $H(n) = O(\log n)$ for any $k(n)$.

2.3 Computing shortest path trees

Consider the following fundamental problem which will be used as a main subroutine by our algorithms in the next section: suppose that we have any two digraphs G_1 and G_2 separated by a single separation pair (v, w). Assume also that the following shortest path (DSP) trees are given: DSP(v, G_2), DSP(w, G_2), DSP(v, G_1), DSP(w, G_1), and DSP(s, G_1), $s \in G_1$. We want to compute DSP$(s, G_1 \cup G_2)$.

In the sequel, by $d_G(x, y)$ we will denote the distance of a shortest path between vertex x and vertex y in a graph G. We will omit the subscipt in the cases where it is clear (by the context) to which graph the distance is refered to.

Proposition 2.1 *Let a be a node of DSP(v, G_2) such that $d(s,v) + d(v,a) > d(s,w) + d(w,a)$. Then, this inequality is also satisfied by any descendant of a in DSP(v, G_2). Similarly, let b be a node of DSP(w, G_2) such that $d(s,w) + d(w,b) > d(s,v) + d(v,b)$. Then, this inequality is also satisfied by any descendant of b in DSP(w, G_2).*

Proof (sketch): Suppose on the contrary that $d(s,v) + d(v,x) \leq d(s,w) + d(w,x)$ for some descendant x of a in DSP(v, G_2). Let $d(s,t;y)$ denote the distance of the shortest path from s to t through vertex y. We have that, $d(s,x;v) = d(s,a;v) + d(a,x) > d(s,a;w) + d(a,x) = d(s,x;w)$. That is, the shortest path from s to x goes through w, a contradiction. The proof for DSP(w, G_2) is similar. ∎

A node c_v in DSP(v, G_2) will be called *critical with respect to* G_2 or simply G_2*-critical* if: (i) $d(s,v) + d(v,c_v) > d(s,w) + d(w,c_v)$ and (ii) $d(s,v) + d(v,x) \leq d(s,w) + d(w,x)$, where x is the parent of c_v in DSP(v, G_2). G_2-critical nodes c_w in DSP(w, G_2) are defined similarly.

A node c_w in DSP(w, G_1) will be called *critical with respect to* G_1 or simply G_1*-critical* if: (i) $d(s,v) + d_{G_2}(v,w) + d(w,c_w) > d(w,c_w)$ and (ii) $d(s,v) + d_{G_2}(v,w) + d(w,x) \leq d(s,x)$, where x is the parent of c_w in DSP(w, G_1). G_1-critical nodes c_v in DSP(v, G_1) are defined similarly.

A node r_s in DSP(s, G_1) will be called *crucial* if: (i) $d(s,v) + d_{G_2}(v,w) + d(w,r_s) \leq d(s,r_s)$ and (ii) $d(s,v) + d_{G_2}(v,w) + d(w,x) > d(s,x)$, where x is the parent of r_s in DSP(s, G_1).

Proposition 2.2 *Let T_1 be the tree resulted from DSP(v, G_2) after the deletion of all subtrees rooted at G_2-critical nodes c_v, and T_2 be the tree resulted from DSP(w, G_2) after the deletion of all subtrees rooted at G_2-critical nodes c_w. Then $T_1 \cap T_2 = \{x | d(s,v) + d(v,x) = d(s,w) + d(w,x)\}$.*

Proof (sketch): Assume that x is a node that belongs to both T_1 and T_2. By proposition 2.1, x cannot be a descendant of a G_2-critical node c_v in DSP(v, G_2). Therefore, x is an ancestor of a G_2-critical node c_v and $d(s,v) + d(v,x) \leq d(s,w) + d(w,x)$. Also, since x is an ancestor of a G_2-critical node c_w (in DSP(w, G_2)), $d(s,v) + d(v,x) \geq d(s,w) + d(w,x)$. Therefore, the only way for a node x to be in $T_1 \cap T_2$ is to satisfy $d(s,v) + d(v,x) = d(s,w) + d(w,x)$. ∎

We are now ready to prove our main lemma.

Lemma 2.3 *Suppose that we have two digraphs G_1 and G_2 (with m_1 and m_2 edges respectively) separated by a single separation pair (v, w). Assume also that the following shortest path trees are given: DSP(v, G_2), DSP(w, G_2), DSP(v, G_1), DSP(w, G_1), and DSP(s, G_1), $s \in G_1$. Then DSP$(s, G_1 \cup G_2)$ can be computed in $O(m_1 + m_2)$ sequential time, or in $O(1)$ parallel time using $O(m_1 + m_2)$ CREW PRAM processors.*

Proof (sketch): Let $d_{G_1 \cup G_2}(v,w) = \min\{d_{G_1}(v,w), d_{G_2}(v,w)\}$. The proof follows a case analysis.
Case 1.a: $d_{G_1}(v,w) \leq d_{G_2}(v,w)$ and $d_{G_1}(w,v) \leq d_{G_2}(w,v)$. Then DSP$(s, G_1 \cup G_2)$ can be obtained by grafting at v and w the trees T_1 and T_2 respectively (as they are defined in proposition 2.2), and breaking ties arbitrarily between T_1 and T_2. This is justified as follows. The fact that $d_{G_1}(v,w) \leq d_{G_2}(v,w)$ and $d_{G_1}(w,v) \leq d_{G_2}(w,v)$, implies that every shortest path in G_1 stays entirely in G_1 (i.e. does not involve a subpath belonging to G_2 with endpoints the vertices v and w). Then v and w is not an ancestor of each other and every node in $T_1 - (T_1 \cap T_2)$ is in a shortest path from s, and the same happens from every node in $T_2 - (T_1 \cap T_2)$. For the nodes in $T_1 \cap T_2$ we decide arbitrarily to which of the subtrees (of DSP$(s, G_1 \cup G_2)$) rooted at v and w they will belong.
Case 1.b: $d_{G_1}(v,w) > d_{G_2}(v,w)$ and $d_{G_1}(w,v) > d_{G_2}(w,v)$, but $d_{G_1}(s,v) + d_{G_2}(v,w) > d_{G_1}(s,w)$ and $d_{G_1}(s,w) + d_{G_2}(w,v) > d_{G_1}(s,v)$. Then DSP$(s, G_1 \cup G_2)$ can be obtained by grafting at v and w the trees T_1 and T_2 respectively, and breaking ties arbitrarily between T_1 and T_2. (Justification is similar to that one of case 1.a.)
Case 2: $d_{G_1}(v,w) > d_{G_2}(v,w)$ and v is an ancestor of w in DSP(s, G_1). Then v is also an ancestor of w in DSP$(s, G_1 \cup G_2)$. (Since v is an ancestor of w in DSP(s, G_1) we have that $d(s,v) + d_{G_1}(v,w) \leq d(s,w)$.) From the hypothesis $d_{G_1}(v,w) > d_{G_2}(v,w)$, hence $d(s,v) + d_{G_2}(v,w) < d(s,w)$. Therefore v is an ancestor of w in DSP$(s, G_1 \cup G_2)$. Moreover, DSP$(s, G_1 \cup G_2)$ can be constructed as follows. Let T^* be the subtree of DSP(s, G_1) rooted at crucial node r_s which is an ancestor of w. Then DSP$(s, G_1 \cup G_2)$ consists of DSP$(s, G_1) - T^*$ along with tree DSP(v, G_2) grafted at node v of $DSP(s, G_1) - T^*$ and tree T^*_{new} grafted at node w of DSP(v, G_2), where T^*_{new} consists of the nodes of T^* arranged in a DSP way rooted at w. The construction is correct since now, all descendants of r_s in DSP(s, G_1) will be descendants of w in DSP$(s, G_1 \cup G_2)$ (from the definition of crucial nodes r_s). Therefore, it suffices to do the following operations on DSP(s, G_1): prune T^* at r_s, graft DSP(v, G_2) at v and also graft a new DSP tree T^*_{new} which will have now all nodes in T^* arranged in a DSP way from w. This can be done by pruning all subtrees rooted at G_1-critical nodes of DSP(w, G_1). Clearly, T^*_{new} can be constructed in $O(|\text{DSP}(w, G_1)|)$ sequential time, or in $O(1)$ parallel time using $O(|\text{DSP}(w, G_1)|)$ CREW PRAM processors.
Case 3: $d_{G_1}(v,w) > d_{G_2}(v,w)$ and v is not an ancestor of w in DSP(s, G_1) (and vice versa).
Case 3.a: $d(s,v) + d_{G_2}(v,w) \geq d(s,w)$. The construction of DSP$(s, G_1 \cup G_2)$ is similar to the construction described in case 1.a.

Case 3.b: $d(s,v) + d_{G_2}(v,w) < d(s,w)$. Then w is a descendant of v in DSP$(s, G_1 \cup G_2)$ (since in such a case, the shortest path from s to w should go through v) and the construction of the DSP tree is done in the same way as it is described in case 2.

Remark 2.2: Clearly, similar cases to 2 and 3 above hold when the roles of v and w are interchanged.

The resource bounds follows from the fact that the computation involves a constant number of manipulations on trees of size at most $O(m_1 + m_2)$ and also by the fact that the dominating step is the identification of critical and crucial nodes at the various DSP trees. This can be done, for each such tree T, in $O(|T|)$ sequential time in a straightforward way. For the parallel implementation, we assign one processor to every node of T. Each processor checks in $O(1)$ time (from the preprocessing of T) if the two conditions in the definition are satisfied. The processors verifying the conditions, mark the associated nodes as critical (or crucial). Finally, the resulted DSP$(s, G_1 \cup G_2)$ tree should be preprocessed. This means that each node v of the tree should know the distance $d(s,v)$. This can be also achieved in $O(1)$ parallel time as follows. Assign a processor to each node of DSP$(s, G_1 \cup G_2)$. If v also belongs to DSP(s, G_1) then do nothing. If not, then let x be the node of DSP(s, G_1) where a new tree T' has been grafted. Note also that x is the root of T'. Then each processor executes the following: $d(s,v) = d(s,x) + d(x,v)$, where $d(x,v)$ is already known by the preprocessing of T'. The bounds follow. Clearly, lemma 2.3 gives body to a procedure for constructing a DSP tree. Let us call it DSP_Tree$(G_1, G_2, (v,w), s, G_1 \cup G_2)$. ∎

Remark 2.3: It is well known [34] that a convergent shortest path (CSP) tree rooted at a vertex s is a directional dual of a DSP tree rooted at s. Therefore, if we reverse the direction of edges in $G_1 \cup G_2$ and apply the above method, we will have a CSP tree rooted at s. Hence, w.l.o.g. we will assume in the sequel that procedure DSP_Tree also computes the CSP tree rooted at s.

3 Dynamic Algorithms for Outerplanar Digraphs

In this section we shall give algorithms for solving the on-line and dynamic shortest path problem in an outerplanar digraph G_o.

3.1 The data structures and the preprocessing algorithm

The data structures used by our algorithms are the following:

(I) the separator tree $S(G_o)$ along with its associated T_S trees. Each node of $S(G_o)$ contains the following attributes: name, level number, a pointer to its parent, a pointer to the corresponding subgraph G, a pointer to the sparse representative $SR(G)$ of G, and a pointer to the associated T_S tree. Each node of T_S contains the following attributes: name, level number, a pointer to its parent, a pointer to its left child, a pointer to its right child, a pointer to the corresponding separation pair p, a pointer to $G_1(p)$, a pointer to $SR(G_1(p))$, a pointer to $G_2(p)$, a pointer to $SR(G_2(p))$, and a pointer to its corresponding node in $S(G_o)$. (In the sequel, we will use the notation $v.x$ to refer to the attribute x of node v in either tree.)

(II) Each graph $SR(G)$ (except for the one corresponding to the root of $S(G_o)$) is associated (via pointers) with at most 6 convergent and divergent compressed shortest path trees (inside $SR(G)$) rooted at the (at most 6) separation vertices of $SR(G)$, that separate it from its parent subgraph. Note that for those subgraphs G corresponding to a leaf of $S(G_o)$, we have that $G \equiv SR(G)$ and therefore the convergent and divergent shortest path trees for these subgraphs, are the original ones. (We also keep temporarily the compact routing tables which help us to create these original convergent and divergent shortest path trees.)

We next give an algorithm which makes a preprocessing of an outerplanar digraph G_o (by creating the above data structures), in order to answer very fast on-line queries requesting either the shortest path or distance between any two vertices, or the shortest path tree rooted at some given vertex. The basic idea of the preprocessing algorithm is the following. We build a tree data structure based on an adaptive separation decomposition of G_o, by an application of algorithm Sep_Tree. Simultaneously, we keep shortest path information among separation vertices in sparse representatives, for each level of the separator tree produced. The shortest path information needed to be stored, is determined by the following observations: (i) the only shortest path and distances that a separation pair p (corresponding to a node in a T_S tree) needs to know, are the shortest paths and distances to all separation pairs that are ancestors of p in T_S; (ii) each separation pair p of a subgraph G, is also a separation pair of $SR(G)$.

For lack of space, we give here an informal description of the preprocessing algorithm. A more formal one is given in the appendix. The algorithm proceeds in three steps. First, find shortest path information in G_o. Second, construct a separator tree $S(G_o)$ using algorithm Sep_Tree. Third, compute the sparse representative $r.SR(G_o)$ of the root node r of $SR(G_o)$ as follows. For each child v of r, run step 3 recursively on v to compute $v.SR(G)$. Having computed $v.SR(G)$, compute their compressed versions $C(v.SR(G))$. Then using lemma 2.3 and the T_S tree associated with r, compute DSP trees rooted at each separation vertex p attached to $f(x).SR(G)$, where $f(x)$ is the parent of a node x in T_S, $f(x).SR(G) = x.SR(G) \cup x^s.SR(G)$ and x^s is the sibling of x in T_S. Initially, $x = v$ and $v.SR(G) = C(v.SR(G))$. Proceed in a bottom-up fashion in T_S until the computation of $r.SR(G_o)$. Let us call the above algorithm Pre_Outerplanar($G_o, k(n)$).

Lemma 3.1 *Algorithm Pre_Outerplanar($G_o, k(n)$) runs in $O(n)$ time. A parallel CREW PRAM implementation of the algorithm runs in $O(\log^2 n)$ time using $O(n)$ processors. The space requirement is $O(n)$.*

Proof (sketch): Step 1 needs $O(n)$ sequential time, or $O(\log^2 n)$ parallel time using $O(n)$ processors and $O(n)$ space, by [14, 29]. Step 2 needs by lemma 2.2 either $O(n)$ sequential time, or $O(\log^2 n)$ parallel CREW PRAM time, using $O(n)$ processors. The space required by step 2 is $O(n)$. Let $P(n)$ be the sequential time, space and parallel work[2] of step 3. Running step 3 recursively for each child v of r in $S(G_o)$ takes $k(n)P(n/k(n))$ sequential time, space and parallel work. Assume that all shortest path information (as well as compressed versions) concerning $v.SR(G)$ have been computed. Computation of $r.SR(G_o)$ needs $O(k(n)\log k(n))$ sequential time, space and parallel work, by lemma 2.3 and tree $r.T_S$. Therefore, step 3 satisfies the recurrence $P(n) = k(n)P((n/k(n)) + O(k(n)\log k(n))$, whose solution is $P(n) = O(n)$ for any $k(n)$. Using similar reasoning, the total parallel time needed is described by the recurrence $T_p(n) = T_p(n/k(n)) + O(\log^2 k(n))$. Its solution depends on the exact value of $k(n)$. For example, for $k(n) = O(n^\mu)$, $0 < \mu < 1$, we have $T_p(n) = O(\log^2 n)$, while for $k(n) = O(\log^a n)$, $a \geq 1$, we have $T_p(n) = O(\log n \log\log n)$. The bounds follow. ∎

3.2 The update algorithm

In the sequel, we will show how we can update our data structures for answering on-line shortest path queries in outerplanar digraphs, in the case where an edge cost is modified. (Note that

[2] i.e. the product of time and number of processors used, or alternatively the total number of operations.

updating after an edge deletion is equivalent to the updating of the cost of the particular edge with a very large weight, such that this edge will not be used by any shortest path.) The algorithm for updating the cost of an edge e in an n-vertex outerplanar digraph G_o is based on the following idea: the edge will belong to at most $O(\log n)$ subgraphs of G_o, as they are determined by the Sep_Tree algorithm. Therefore, it suffices to update (in a bottom-up fashion) those subgraphs that are on the path from the subgraph $\ell.G$ containing e (where ℓ is a leaf of $S(G_o)$) until the root of $S(G_o)$. Let r be the root of $S(G_o)$. (Clearly, $G_o \equiv r.G$.) Let $f(z)$ denote the parent of a node z in $S(G_o)$ or in some tree T_S, and u^s denote the sibling of a node u in a T_S tree. Note that $u.G \cup u^s.G = f(u).G$, and of course, $u.SR(G) \cup u^s.SR(G) = f(u).SR(G)$. The algorithm for the update operation is the following.

```
ALGORITHM Update_Outerplanar($r.G, k(n), e, w(e)$)
BEGIN
1. Find the child $x$ of $r$ in $S(G_o)$ for which $e \in x.G$.
2. if $x.level = d(k(n))$ then
      Update the edge cost of $e$ with the new one $w(e)$ in $x.G$.
      Compute new convergent and divergent shortest path trees rooted
      at the separation vertices attached to $x.G$.
   else (a) Update_Outerplanar($x.G, k(n), e, w(e)$)
        (b) Find the child $y$ of $r$ in $r.T_S$ for which $e \in y.G$.
        (c) Run step (2.b) recursively on $y$.
        (* Let $(p_1, p_2)$ be the separation pair between $y.SR(G)$ and $y^s.SR(G)$. *)
        (d) for each separation vertex $p$ attached to $f(y).SR(G)$ do
               DSP_Tree($y.SR(G), y^s.SR(G), (p_1, p_2), p, f(y).SR(G)$).
        (e) if $f(y)$ is the root of a $T_S$ tree, but not the root of $S(G_o)$ then
               Generate a compressed vesrion of $f(y).SR(G)$.
END.
```

Lemma 3.2 *Algorithm Update_Outerplanar updates the data structures created by the preprocessing algorithm after an edge cost modification: (i) in $O(k(n))$ sequential time, or in $O(\log n)$ parallel time using $O(k(n))$ CREW PRAM processors, if $k(n) = \Theta(n^\mu)$, $0 < \mu < 1$; (ii) in $O(d(k(n)) \cdot k(n))$ sequential time, or in $O(\log n)$ parallel time using $O(d(k(n)) \cdot k(n))$ CREW PRAM processors, if $k(n) = O(\log^a n)$, $a \geq 0$, a constant.*

Proof (sketch): We need only to determine which leaf of $S(G_o)$ contains the given edge e. This takes $O(\log n)$ time using a single processor (according to remark 2.1). ¿From the construction of $S(G_o)$ and T_S, the rest of $x.G$ and $y.G$ that contain also the edge e are determined from the path between the appropriate leaf and the root of each one of $S(G_o)$ and T_S. Clearly, the total sequential time and CREW PRAM processors used are described by the recurrence $U(n) = U(n/k(n)) + O(k(n))$, since (as it follows by lemma 2.3) the sequential time and number of processors used in order to update a tree T_S with $k(n)$ leaves, as well as to compress the sparse representative associated with its root [14, 29], is $O(k(n))$. Therefore, $U(n) = O(d(k(n)) \cdot k(n))$ if $k(n) = O(\log^a n)$, $a \geq 0$ and $U(n) = O(k(n))$ if $k(n) = \Theta(n^\mu)$, $0 < \mu < 1$. Using similar reasoning, the CREW PRAM time satisfies the recurrence $T(n) = T(n/k(n)) + O(\log k(n))$. ∎

Remark 3.1: An edge e may belong to two subgraphs of G_o. In the case where e belongs to subgraphs $x.G$ and $x'.G$, where x and x' are children of the root of $S(G_o)$, run algorithm Update_Outerplanar on $x.G$ and $x'.G$. Let $U'(n)$ be the total sequential time and CREW PRAM processors used, for updating G_o in this case. Then $U'(n) = 2U(n/k(n)) + O(k(n))$, where $U(m)$ is the time and processors used by algorithm Update_Outerplanar in order to update a digraph of size $O(m)$. Clearly, the resource bounds of lemma 3.2 remain the same.

3.3 The single-pair query algorithm

We shall give an informal description of the query algorithm for finding the shortest path or distance between any two vertices v, z. (A more formal description is given in the appendix.) The algorithm is as follows. Search $S(G_o)$ to find a node v_S such that S separates v from z in G_o. Determine also the subgraphs H and H' that contain v and z respectively. Search the corresponding tree $v_S.T_S$ to find the first separation pair $p = (p_1, p_2)$ in $v_S.T_S$ that separates v and z. Compute the shortest path or distance between v and z, as follows. Let $d(v, z)$ denote the distance of the shortest path $SP(v, z)$ between v and z. Then clearly, $d(v, z) = \min\{d(v, p_1) + d(p_1, z), d(v, p_2) + d(p_2, z)\}$. Hence, it suffices to compute the distances $d(v, p_1)$, $d(p_1, z)$, $d(v, p_2)$ and $d(p_2, z)$ (as well as their corresponding shortest paths). We will show how to compute the shortest path or distance $d(v, p_1)$. (The computation of the other ones is similar.) Since H has at most 6 separation vertices x_i, $1 \leq i \leq 6$, it is clear that $d(v, p_1) = \min_i\{d(v, x_i) + d(x_i, p_1)\}$. Computation of $d(x_i, p_1)$ can be done using the shortest path information associated with the nodes of $v_S.T_S$. The computation of $d(v, x_i)$ can be done as follows. First, find the leaf ℓ of $S(G_o)$, for which $v \in \ell.G$ (where G is the subgraph associated with ℓ). From the convergent and divergent shortest path trees in $\ell.G$, compute $d(v, \ell.x_j)$ where $\ell.x_j$ are the separation vertices of $\ell.G$. Clearly, $d(v, x_i) = \min_j\{d(v, \ell.x_j) + d(\ell.x_j, x_i)\}$ where $1 \leq j \leq 6$. Compute $d(\ell.x_j, x_i)$ step-by-step, by walking the path from ℓ up to v_S and by using the shortest path information associated with the nodes of $S(G_o)$. What left to be explained is how the original shortest path can be obtained. This can be done by uncompressing the compressed shortest path $C(SP(v, p_1))$. The uncompression involves at most a traversal of the subtree of $S(G_o)$ rooted at v_S and a traversal of the path from v_S up to the root of $S(G_o)$. Clearly, the traversal time can not exceed the number of edges of the original path.

Lemma 3.3 *Algorithm Query_Outerplanar($G_o, v, z, k(n)$) finds the shortest path (resp., distance) between any two vertices v and z in an n-vertex outerplanar digraph in $O(L+\log n)$ (resp., $O(\log n)$) time using a single processor.*

Proof: In the appendix. ∎

3.4 The single-source query algorithm

Let $U \subset V$ be a subset of $O(1)$ vertices of G_o with a weight $d_0(u)$ on any $u \in U$. For any vertex v of G the weighted distance $d(U, v)$ is defined by $d(U, v) = \min\{d_o(u) + d(u, v) | u \in U\}$. We assume that $d(U, v) = d_0(v)$ for every $v \in U$. The following algorithm computes $d(U, v)$, $\forall v \in G_o$.

```
ALGORITHM Query_Outerplanar_DSP_Tree($G_o, U$)
BEGIN
    1. Let $S$ be the separator corresponding to the root $r$ of $S(G_o)$. Compute $d(u, s)$ for all vertices
$u \in U$ and $s \in S$ by using the query algorithm.
    2. For any $s \in S$ define $d_0(s) = \min\{d(u, s) | u \in U\} = d(U, s)$.
    3. Run recursively Query_Outerplanar_DSP_Tree($w.G, (S \cup U) \cap w.G$), on each child $w$ of $r$
which is not a leaf of $S(G_o)$. (If $w$ is a leaf, then DSP trees are computed easily since the associated
graph is of $O(1)$ size.)
```

END.

Lemma 3.4 *Algorithm Query_Outerplanar_DSP_Tree runs in $O(n)$ time using a single processor.*

Proof (sketch): Correctness of the algorithm comes from the fact that two outerplanar subgraphs of G_o have only one separation pair in common. Let $D(n)$ be the running time of the algorithm. Then, $D(n) \leq k(n)D(n/k(n)) + O(|S| \cdot |U| \cdot \log n) = k(n)D(n/k(n)) + O(k(n)\log n)$, which gives $D(n) = O(n)$ for any $k(n)$. ∎

Having the distances, we can produce a shortest path tree rooted at any $u \in U$ in $O(n)$ time as follows. Each vertex x (except for u) checks its neighbors and selects as its parent that vertex y which satisfies: $d(u,x) = d(u,y) + c(x,y)$ ($c(x,y)$ is the weight of edge $\langle x,y \rangle$). If v is any vertex of the graph, then the single-source shortest path tree rooted at v can be computed in $O(n)$ time by running Query_Outerplanar_DSP_Tree($G_o, \{v\}$) with $d_0(v) = 0$.

3.5 Main results and negative edge costs

Theorems 1 and 2 (for the case of outerplanar digraphs, i.e. $q = 1$) follow from lemmata 3.1, 3.2, 3.3 and 3.4, if we set $k(n) = O(1)$. In the case of negative cycle costs we have the following. The initial digraph G_o can be tested for a negative cycle either in $O(n)$ sequential time, or in $O(\log n \log^* n)$ parallel time using $O(n/\log n \log^* n)$ CREW PRAM processors, by results of [22]. Now, assume that we want to modify the cost $c(v,w)$ of an edge $\langle v,w \rangle$, to $c'(v,w)$ in G_o. Before running the Update_Outerplanar algorithm, run the algorithm Query_Outerplanar to find the distance $d(w,v)$. If $d(w,v) + c'(v,w) < 0$, then halt and announce not acceptance of this edge cost modification. Otherwise, continue in the known way. Clearly, the above procedures for either testing the initial digraph, or testing the acceptance of the edge cost modification, does not affect the resource bounds of either our reprocessing or of our update algorithm, respectively.

4 Dynamic Algorithms for Planar Digraphs

The algorithms for maintaining all pairs shortest paths information in a planar digraph G are based on the hammock decomposition idea and on the algorithms of the previous section. The preprocessing algorithm for G is as follows: find a hammock decomposition of G into $O(q)$ hammocks. Construct local data structures for each hammock H using algorithm Pre_Outerplanar. Also, in each hammock H construct convergent and divergent shortest path trees rooted at each attachment vertex of H. Compress each hammock into a O(1)-sized graph such that the shortest paths between its attachment vertices are preserved. This results into a sparse representative G_q of G, which is a planar graph of size $O(q)$. Finally, run the preprocessing algorithms of [11] or [4] for constructing a data structure in G_q. ¿From results of [4, 11, 14, 29] discussed in section 2 and lemmata of section 3 (with $k(n) = O(1)$), we have the preprocessing bounds stated in theorems 1 and 2.

A single-pair query between any two vertices v and z can be answered as follows (using the above data structures). If v and z do not belong to the same hammock, then their distance $d(v,z) = \min_{i,j}\{d(v,a_i) + d(a_i,a'_j) + d(a'_j,z)\}$ where a_i and a'_i respectively are the attachment vertices of the hammocks in which v and z belong to. If both v and z belong to the same hammock H, then note that the shortest path between them does not necessarily have to stay in H. Hence, first compute (using algorithm Query_Outerplanar) their distance $d_H(v,z)$ inside H. After that compute $d_{ij}(v,z) = \min_{i,j}\{d(v,a_i) + d(a_i,a_j) + d(a_j,z)\}$. Clearly, $d(v,z) = \min\{d_H(v,z), d_{ij}(v,z)\}$.

In sequential computation we need $O(q)$ time for querying in G_q's data structure [11] and $O(\log n)$ time for querying in each hammock (lemma 3.3). In parallel computation, a distance query in G_q's data structure is answered in $O(1)$ time. The query bounds, in theorems 1 and 2, follow.

The update algorithm is straightforward. Let e be the edge that its cost has been modified. We have two data structures that should be updated. The first one concerns the hammock H where e belongs to. This is done by the algorithm Update_Outerplanar. The second data structure is that of the digraph G_q and can be updated in $O(\log^3 q)$ sequential time by [11], or in $O(\log^2 q)$ parallel time using $O(q^2/\log q)$ CREW PRAM processors by [4]. The algorithm for answering a single-source query is based on a generalization of lemma 2.3 and for lack of space is given in the appendix. The bounds of theorems 1 and 2 follow by the above discussion.

The case of negative edge costs is handled in a similar way with that of outerplanar digraphs. The initial digraph can be tested for a negative cycle either in $O(n + q^2)$ sequential time, or in $O(\log^2 n + \log^4 q)$ parallel time using $O(n+q^2/\log^3 q)$ CREW PRAM processors [22]. The procedure for accepting or not an edge cost modification is similar to the one for outerplanar digraphs.

5 Extensions of our results and further applications

Our results can be extended to hold for any digraph with genus bounded by a function $g(n) = o(n)$, by the fact that the hammock decomposition technique applies to *any* digraph G [13, 22]. Actually, the number of hammocks produced is a small constant factor times $g(n) + q'$, where G is supposed to be embedded on an orientable surface of genus $g(n)$ such that all vertices are covered by q' of the faces, and $g(n), q'$ are the minimum possible. (Note that the methods in [13, 22] do not require such an embedding to be provided by the input in order to produce the hammock decomposition.) If the digraph is provided with a separator decomposition, then all of our results hold as they are. Otherwise, the bounds depending on q (i.e. on the number of hammocks) are getting a little bit larger, but they still compare favorably with either the results of [9], or by running the best off-line algorithm from srcatch.

Another important application of our results here is the maintainance of the all pairs reachability information in a planar digraph G. We treat the reachability problem as a degenerated version of the shortest path problem. (Assign a weight of 1 to all edges in G. If $d(z, v) \neq \infty$ then v is reachable from z.) The results of [33] provide fully dynamic algorithms for this problem. A data structure can be constructed in $O(n \log n)$ time using $O(n)$ space. A reachability query is answered in $O(n^{2/3} \log n)$ time. Also in the same time the data structure can be updated after an edge deletion. In case of an edge insertion, the data structure can be updated in $O(n^{2/3} \log n)$ amortized time (which can be $O(n \log n)$ in the worst case, i.e. equivalent to running the algorithm from scratch). Our methods achieve the following bounds: (a) we can construct a data structure in $O(n + q \log q)$ time, using $O(n)$ space. A query can be answered in $O(\log n + q)$ time, while an update of the data structure after an edge deletion is done in $O(\log n + \log^3 q)$ time; (b) we can construct a data structure in $O(\log^2 n + \log^4 q)$ time using $O(n + q^{1.19})$ CREW PRAM processors. The space required is $O(n + q^2)$. A query can be answered in $O(\log n)$ time, while the data structure can be updated after an edge deletion in $O(\log n)$ time using $O(\log n + q^{1.19})$ CREW PRAM processors. (The bounds in parallel computation come from results in [4, 22].)

References

[1] H. Alt, T. Hagerup, K. Mehlhorn and F. Preparata, "Deterministic Simulation of Idealized Parallel Computers

on More Realistic Ones", *SIAM J. Comp.* , Vol.16, N.5, October 1987, pp.808-835.

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

[3] M. Carroll and B. Ryder, "Incremental Data Flow Analysis via Dominator and Attribute Grammars", *Proc. 15th ICALP*, 1988.

[4] E. Cohen, "Efficient Parallel Shortest-paths in Digraphs with a Separator Decomposition", *Proc. 5th ACM Symp. on Parallel Algorithms and Architectures*, 1993.

[5] R. Cole and U. Vishkin, "Deterministic Coin Tossing with Applications to Optimal Parallel List Ranking", *Inform. and Control* Vol.70, 1986, pp.32-53.

[6] H. Djidjev, "Linear Algorithms for Graph Separation Problems", *Proc. 1st SWAT88*, 1988, LNCS, Vol.318, pp.216-221, Springer-Verlag.

[7] H. Djidjev, G. Pantziou and C. Zaroliagis, "Computing Shortest Paths and Distances in Planar Graphs", in *Proc. 18th ICALP*, 1991, LNCS, Vol. 510, pp. 327-339, Springer-Verlag.

[8] P. Erdos and J. Spencer, "Probabilistic Methods in Combinatorics", Academic Press, 1974.

[9] S. Even and H. Gazit, "Updating distances in dynamic graphs", *Methods of Operations Research*, Vol.49, 1985, pp.371-387.

[10] D. Eppstein, Z. Galil, G. Italiano and A. Nissenzweig, "Sparsification - A Technique for Speeding Up Dynamic Graph Algorithms", *Proc. 33rd Symp. on FOCS*, 1992, pp.60-69.

[11] E. Feuerstein and A.M. Spaccamela, "Dynamic Algorithms for Shortest Paths in Planar Graphs", in *Proc. 17th Workshop on Graph-Theoretic Concepts in Computer Science (WG91)*, LNCS, Vol.570, pp.187-197, Springer-Verlag, 1991.

[12] G.N. Frederickson, "Fast algorithms for shortest paths in planar graphs, with applications", *SIAM J. on Computing*, 16 (1987), pp.1004-1022.

[13] G.N. Frederickson, "Using Cellular Graph Embeddings in Solving All Pairs Shortest Path Problems", *Proc. 30th Annual IEEE Symp. on FOCS*, 1989, pp.448-453; also CSD-TR-897, Purdue University, August 1989.

[14] G.N. Frederickson, "Planar Graph Decomposition and All Pairs Shortest Paths", *J. ACM*, Vol.38, No.1, January 1991, pp.162-204; also TR-89-015, ICSI, Berkeley, March 1989.

[15] M. Fredman, "New bounds on the complexity of the shortest path problem", *SIAM J. Comp.* , 5(1976), pp.83-89.

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

[17] M. Fredman and R. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms", *JACM*, 34(1987), pp. 596-615.

[18] Z. Galil and G. Italiano, "Fully Dynamic Algorithms for Edge-connectivity Problems", *Proc. 23rd ACM STOC*, 1991, pp.317-327.

[19] Z. Galil and G. Italiano, "Maintaining Biconnected Components of Dynamic Planar Graphs", *Proc. 18th ICALP*, 1991, LNCS Vol. 510, pp.339-350, Springer-Verlag.

[20] J. JáJá, "An Introduction to Parallel Algorithms", Addison-Wesley, 1992.

[21] R. Karp and V. Ramachandran, "Parallel Algorithms for Shared-Memory Machines", *Handbook of Theoretical Computer Science*, Ed. J. van Leeuwen, pp. 869-941, 1990, Elsevier Science Publishers.

[22] D. Kavvadias, G. Pantziou, P. Spirakis and C. Zaroliagis, "Hammock-on-Ears Decomposition: A Technique for Parallel and On-line Path Problems", Computer Technology Institute Technical Report, CTI-TR-93.05.22, Patras, May 1993.

[23] D. Kavvadias, G. Pantziou, P. Spirakis and C. Zaroliagis, "On the expected number of hammocks in a graph", preprint, October 1993.

[24] Z. Kedem, K. Palem and P. Spirakis, "Efficient Robust Parallel Computations", *Proc. 22nd ACM STOC*, pp.138-148, 1990.

[25] Z. Kedem, K. Palem, A. Raghunathan and P. Spirakis, "Combining Tentative and Definite Algorithms for Very Fast Dependable Parallel Computing", *Proc. 23nd ACM STOC*, pp.381-390, 1991.

[26] Z. Kedem, K. Palem, A. Raghunathan and P. Spirakis, "Resilient Parallel Computing on Unreliable Parallel Machines", chapter 8 in *Lectures on Parallel Computation*, Ed. A. Gibbons & P. Spirakis, pp.149-175, Cambridge University Press, 1993.

[27] P. Klein and S. Subramanian, "A linear-processor polylog-time algorithm for shortest paths in planat graphs", *Proc. 34th IEEE Symp. on FOCS*, 1993.

[28] A. Lingas, "Efficient Parallel Algorithms for Path Problems in Planar Directed Graphs", *Proc. SIGAL'90*, LNCS 450, pp.447-457, 1990, Springer-Verlag.

[29] G. Pantziou, P. Spirakis and C. Zaroliagis, "Efficient Parallel Algorithms for Shortest Paths in Planar Digraphs", *BIT* 32 (1992), pp.215-236.

[30] M. Rauch, "Fully Dynamic Biconnectivity in Graphs", *Proc. 33rd IEEE Symp. on FOCS*, 1992, pp.50-59.

[31] N. Santoro and R. Khatib, "Labeling and implicit routing in networks", *Computer Journal*, 28 (1985), pp.5-8.

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

[33] S. Subramanian, "A Fully Dynamic Data Structure for Reachability in Planar Digraphs", *Proc. 1st European Symp. on Algor. (ESA93)*, LNCS, pp.372-383, 1993, Springer-Verlag.

[34] R.E. Tarjan, "Data Structures and Network Algorithms", SIAM, 1983.

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

[36] J. Westbrook, "Algorithms and Data Structures for Dynamic Graph Problems", PhD Dissertation, CS-TR-229-89, Dept of Computer Science, Princeton University, 1989.

[37] M. Yannakakis, "Graph Theoretic Methods in Database Theory", *Proc. ACM conference on Principles of Database Systems*, 1990.

[38] D. Yellin and R. Strom, "INC: a language for incremental computations", *Proc. ACM SIGPLAN conf. on Programming Language Design and Implementation*, 1988.

APPENDIX

A.1 Proofs

Proof (sketch) of lemma 2.1: First of all notice that all tentative separation pairs of G_u are separation edges, since G_u is triangulated. Let H be the subgraph produced after the removal of the c separation pairs from G_i, $i = 1, 2$. It is clear that if $c = 3$ and the separation pairs belong to the same face, then H cannot be further separated. Therefore, we assume that $c \geq 3$ and the c separation pairs do not belong to the same face of G_u. Suppose on the contrary that there is no separation pair, separating the c pairs. Now, H will consists of at least $c + 1$ vertices. Since G_u is triangulated, it follows that there exists at least one edge in H connecting two nonconsecutive (according to their clockwise naming) vertices. But such an edge is a separation edge, a contradiction. The sequential resource bound of the algorithm comes from results in [6]. For the parallel implementation we have the following. Step 1 needs $O(\log n)$ time and $O(n)$ processors in either model [29]. The depth of the recursion in step 4 is $O(\log_{1/\varepsilon} k(n))$. Each recursion step needs $O(\log n)$ time with $O(n)$ CREW PRAM processors. (Assign one processor in every edge of the graph and select the one which satisfies the suitable separation condition. In case of ties choose one arbitrarily.) The bounds follow. Also note that at level i of recursion, each component has size at most $O(\varepsilon^i n)$. Since the depth of the resursion is $O(\log_{1/\varepsilon} k(n))$, we have that no component has size larger than $O(n/k(n))$. ∎

Proof (sketch) of lemma 3.3: The correctness is clear by the description of the algorithm.

Searching $S(G_o)$, as well as its associated T_S trees takes in total $O(\log n)$ time by remark 2.1. Let $Q(n)$ be the time for computing the distances $d(v,p_1), d(v,p_2), d(p_1,z)$ and $d(p_2,z)$. Then $Q(n) = Q'(n) + O(\log k(n))$, where $Q'(n)$ is the time to compute all distances $d(v,x_i)$. $Q'(n)$ satisfies the recurrence $Q'(n) = Q'(n/k(n)) + O(\log k(n))$, which gives $Q'(n) = O(\log n)$ and consequently $Q(n) = O(\log n)$ for any $k(n)$. Once we have decided (from the computation of the distances) which is the correct compressed shortest path, we need $O(L)$ time to output the original one (as it was explained in the informal description of the algorithm). The bounds follow. ∎

A.2 The preprocessing algorithm for outerplanar digraphs

```
ALGORITHM Pre_Outerplanar($G_o, k(n)$)
BEGIN
1. Find shortest path (edge labeling) information in $G_o$.
2. Construct a separator tree $S(G_o)$ using algorithm Sep_Tree($G_o, k(n)$).
3. Compute the sparse representative $r.SR(G_o)$ as follows.
(* Let $v$ be any child of $r$ in $r.T_S$ *)
(a) if $v$ is a leaf in $r.T_S$ then Create_SR($v.G$)
    else run step 3 recursively on each $v$.
(b) $f(v).SR(G) = v.SR(G) \cup v^s.SR(G)$.
(c) (* Let $(p_1,p_2)$ be the separation pair between $v.SR(G)$ and $v^s.SR(G)$. *)
(d) for each separation vertex $p$ attached to $f(v).SR(G)$ do
        DSP_Tree($v.SR(G), v^s.SR(G), (p_1,p_2), p, f(v).SR(G)$).
END.
```

```
PROCEDURE Create_SR($x.G$)
BEGIN
if $x.level = d(k(n))$ then compute convergent and divergent shortest path trees rooted at
  the separation vertices attached to $x.G$. Let $x.SR(G) = x.G$.
else
  for each child $y$ of $x$ in $S(G_o)$ do
          Create_SR($y.G$).
          Aux_SR($y.G$).
END.
```

```
PROCEDURE Aux_SR($y.G$)
BEGIN
(* Let $z, z^s$ be the two children of $y$ in $y.T_S$ *)
1. if $z$ is a leaf in $y.T_S$ then let $z.SR(G)$ be its corresponding sparse representative.
   else  Aux_SR($z.G$).
         Aux_SR($z^s.G$).
2. $f(z).SR(G) = z.SR(G) \cup z^s.SR(G)$.
3. (* Let $(p_1,p_2)$ be the separation pair between $z.SR(G)$ and $z^s.SR(G)$. *)
   for each separation vertex $p$ attached to $f(z).SR(G)$ do
       DSP_Tree($z.SR(G), z^s.SR(G), (p_1,p_2), p, f(z).SR(G)$).
```

```
4. if $f(z) = y$ then
        Store a pointer to $f(z).SR(G)$ and generate a compressed version $C(f(z).SR(G))$.
        $f(z).SR(G) = C(f(z).SR(G))$.
END.
```

A.3 The single-pair query algorithm for outerplanar digraphs

ALGORITHM Query_Outerplanar($G_o, v, z, k(n)$)
BEGIN

1. Search $S(G_o)$ (starting form the root) to find a node v_S such that S separates v from z in G_o. Determine also those children v_1 and v_2 of v_S such that $v \in v_1.G$ and $z \in v_2.G$.

2. Search the tree $v_S.T_S$ (starting from the root) to find a node w such that: (i) the separation pair $w.p = (p_1, p_2)$ separates v and z, and (ii) $w.level$ is of minimum possible.

3. $d(v,z) = \min\{d(v,p_1) + d(p_1,z), d(v,p_2) + d(p_2,z)\}$. (A similar expression holds for computing the shortest path $SP(v,z)$.)

Computation of the shortest path or distance $d(v,p_1)$: (The computation of the other shortest paths or distances is similar.) The steps of the computation will be given in a top-down fashion. Let $x.a_i$, $1 \leq i \leq 6$, the separation vertices attached to $x.G$, where x is a tree node of either $S(G_o)$ or a T_S tree.

$$d(v,p_1) = \min_i\{d(v,v_1.a_i) + d(v_1.a_i,p_1)\}$$

(a) *Computation of* $d(v_1.a_i,p_1)$: Let w' be that child of w, for which $v_1.a_i \in w'.SR(G)$. From the preprocessing of the convergent shortest path tree rooted at p_1 in the graph $w'.SR(G)$, output $d(v_1.a_i,p_1)$ as well as the compressed shortest path $C(SP(v_1.a_i,p_1))$.

(b) *Computation of* $d(v,v_1.a_i)$:

$$\begin{aligned} d(v,v_1.a_i) &= \min\{d_{v_1.SR(G)}(v,v_1.a_i), \min_{(j,k)}\{d_{v_1.SR(G)}(v,v_1.a_j) \\ &\quad + d(v_1.a_j,v_1.a_k) + d_{v_1.SR(G)}(v_1.a_k,v_1.a_i)\}\} \end{aligned}$$

(* COMMENT: Note that there are at most 2 such pairs (a_j,a_k) and that $d_{v_1.SR(G)}(v_1.a_k,v_1.a_i)$ is already known from the divergent shortest path tree rooted at $v_1.a_k$ in $v_1.SR(G)$. *)

(b.1) *Computation of* $d_{v_1.SR(G)}(v,v_1.a_i)$: This is done by the following procedure.

```
PROCEDURE D_Inside($d_{v_1.SR(G)}(v, v_1.a_i)$)
BEGIN
    1. Find the child $u_1$ of $v_1$ for which $v \in u_1.G$.
    2. D_Inside($d_{u_1.SR(G)}(v, u_1.a_i)$).
    3. $d_{v_1.SR(G)}(v, v_1.a_i) = \min_{1 \leq j \leq 6}\{d_{u_1.SR(G)}(v, u_1.a_i) + d_{v_1.SR(G)}(u_1.a_j, v_1.a_i)\}$.
(* COMMENT: All distances (and compressed shortest paths) $d_{v_1.SR(G)}(u_1.a_j, v_1.a_i)$ are known from
the convergent shortest path tree rooted at $v_1.a_i$ in $v_1.SR(G)$. *)
END.
```

(b.2) *Computation of* $d(v_1.a_j,v_1.a_k)$ *outside* $v_1.SR(G)$:

(b.2.a) *Inside* $v_S.SR(G)$: Let $p' = (p'_1,p'_2)$ be the separation pair associated with the root of $v_S.SR(G)$ and let $v \in u.G$, where u and u^s are the two children of the root.

PROCEDURE D_Outside_Partial($d_{v_S.SR(G)}(v_1.a_j, v_1.a_k)$)

```
BEGIN
    1. D_Outside_Partial($d_{u.SR(G)}(v_1.a_j, v_1.a_k)$).
    2. $d_{v_S.SR(G)}(v_1.a_j, v_1.a_k) = \min\{d_{u.SR(G)}(v_1.a_j, v_1.a_k), \min\{d_{u.SR(G)}(v_1.a_j, p'_1) + d_{u^s.SR(G)}(p'_1, p'_2) + d_{u.SR(G)}(p'_2, v_1.a_k), d_{u.SR(G)}(v_1.a_j, p'_2) + d_{u^s.SR(G)}(p'_2, p'_1) + d_{u.SR(G)}(p'_1, v_1.a_k)\}\}$
END.
```

(b.2.b) *General solution* : Let R be the root of $S(G_o)$ and $v \in u.G$, where u is a child of R.

```
PROCEDURE D_Outside_General($d(v_1.a_j, v_1.a_k)$)
BEGIN
    1. D_Outside_Partial($d_{v_S.SR(G)}(v_1.a_j, v_1.a_k)$).
    2. $d(v_1.a_j, v_1.a_k) = \min\{d_{v_S.SR(G)}(v_1.a_j, v_1.a_k), \min_{(m,l)}\{d_{v_S.SR(G)}(v_1.a_j, v_S.a_m) +$ D_Outside_General($d_{f(v_s).SR(G)}(v_S.a_m, v_S.a_l)$) $+ d_{v_S.SR(G)}(v_S.a_l, v_1.a_k)\}\}$
END.
END. (* End of algorithm Query_Outerplanar. *)
```

A.4 The single-source query algorithm for planar digraphs

The algorithm for answering a query asking for the shortest path tree rooted at a specified vertex s is based on a generalization of lemma 2.3. Let H be the hammock where s belongs to and let a_i, $1 \leq i \leq 4$ be its attachment vertices. Let G' be a subgraph of G. A node c_{a_i} in DSP(a_i, G') is called *critical with respect to s* ($s \in V(H)$) if: (i) $d(s, a_i) + d(a_i, c_{a_i}) > d(s, a_j) + d(a_j, c_{a_i})$, $a_i \neq a_j$, and (ii) $d(s, a_i) + d(a_i, x) \leq d(s, a_j) + d(a_j, x)$, where x is the parent of c_{a_i} in DSP(a_i, G'). A node c_s in DSP(s, H) is called *crucial with respect to a_i* if: (i) $d(s, a_i) + d(a_i, a_j) + d(a_j, c_s) \leq d_H(s, c_s)$, $a_i \neq a_j$, and (ii) $d(s, a_i) + d(a_i, a_j) + d(a_j, x) \leq d_H(s, x)$, where x is the parent of c_s in DSP(s, H). It is not difficult to see that lemma 2.3 holds for any $O(1)$ number of separation pairs (between G_1 and G_2) along with the above generalized definitions of critical and crucial nodes. The algorithm is as follows.

```
ALGORITHM Query_Planar_DSP_Tree($G, s$)
BEGIN
    1. Compute DSP$(s, H)$ and DSP$(a_i, (G - H) \cup \{a_j | j \neq i\})$.
    2. Determine for each tree DSP$(a_i, (G - H) \cup \{a_j | j \neq i\})$ its critical nodes $c_{a_i}$ with respect to $s$.
    3. Prune each DSP$(a_i, (G - H) \cup \{a_j | j \neq i\})$ at nodes $c_{a_i}$, resulting in trees $T_i$.
    4. If $a_j \in T_i$ $(i \neq j)$, then find crucial nodes $c_s$ in DSP$(s, H)$ with respect to $a_i$.
    5. Prune DSP$(s, H)$ at crucial nodes $c_s$. Let the subtrees rooted at nodes $c_s$ be denoted as $T^*$.
    6. Graft $T_i$ at its associated node $a_i$ in DSP$(s, H)$ (for those $T_i$ which this is applicable). Let, the new tree resulted, be denoted by DSP$(s, H) \cup T_i$.
    7. Arrange the nodes of each $T^*$ in a DSP way rooted at their associated $a_j$ node. This is done in the same way as it is described in the proof of lemma 2.3. Hence, new trees $T^*_{new}$ are created.
    8. Graft each $T^*_{new}$ at its associated node $a_j$ in DSP$(s, H) \cup T_i$.
```

END.

Lemma: *Algorithm Query_Planar_DSP_Tree constructs a shortest path tree rooted at a specified vertex of an n-vertex planar digraph in $O(n + q\sqrt{\log\log q})$ time using a single processor, in sequential computation. Algorithm Query_Planar_DSP_Tree, in either model of parallel computation, constructs a shortest path tree rooted at a specified vertex of an n-vertex planar digraph in $O(n)$ time using a single processor.*

Proof (sketch): By generalization of lemma 2.3 and [11]. In parallel computation the algorithm Query_Planar_DSP_Tree runs in $O(n)$ time using a single processor, since the trees DSP$(a_i, (G - H) \cup \{a_j | j \neq i\})$ have already been computed (or updated) from the preprocessing (updating) of G. ∎