Fast Algorithms for Maintaining Shortest Paths in Outerplanar and Planar Digraphs * (FCT'95 ver.)

Hristo N. Djidjev[1], Grammati E. Pantziou[2] and Christos D. Zaroliagis[3]

[1] Computer Science Dept, Rice University, P.O. Box 1892, Houston, TX 77251, USA
[2] Computer Science Dept, University of Central Florida, Orlando FL 82816, USA
[3] Max-Planck Institut für Informatik, Im Stadtwald, 66123 Saarbrücken, Germany

Abstract. We present algorithms for maintaining shortest path information in dynamic outerplanar digraphs with sublogarithmic query time. By choosing appropriate parameters we achieve continuous trade-offs between the preprocessing, query, and update times. Our data structure is based on a recursive separator decomposition of the graph and it encodes the shortest paths between the members of a properly chosen subset of vertices. We apply this result to construct improved shortest path algorithms for dynamic planar digraphs.

1 Introduction

The design and analysis of algorithms for dynamic graph problems is one of the most active area of current algorithmic research. For solving a dynamic graph problem one has to design an efficient data structure that not only allows fast answering to a series of queries, but that can also be easily updated after a modification of the input data. 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 two vertices v and w of G is the minimum length of a path between v and w. The path of minimum length between v and w is called a *shortest path* between v and w. Finding shortest path information in graphs is an important and intensively studied problem with many applications. The dynamic version of the problem has also been studied recently [3, 4] and is stated as follows: Given G (as above), build a data structure that will enable fast on-line shortest path or distance queries. In case of edge cost modification of G, update the data structure in an appropriately short time. We will refer to the above, as the *dynamic shortest path* (DSP) problem. The DSP problem has a lot of applications, including dynamic maintenance of a maximum st-flow in a network [7], computing a feasible flow between multiple sources and sinks as well as finding a perfect matching in bipartite planar graphs [8].

In this paper, we investigate the DSP problem for particular classes of digraphs, namely those of outerplanar and planar digraphs. An efficient solution

* This work was partially supported by the EU ESPRIT BRA No. 7141 (ALCOM II), by the EU Cooperative Action IC-1000 (ALTEC) and by the NSF grant No. CCR-9409191. Email: hristo@cs.rice.edu, pantziou@cs.ucf.edu, zaro@mpi-sb.mpg.de.

to the DSP problem for outerplanar digraphs has been given in [3] where a data structure is constructed in $O(n)$ time and space and then any distance query is answered in $O(\log n)$ time. The data structure is updated after an edge cost modification in $O(\log n)$ time. (A different approach leading to the same performance characteristics is claimed in [1].) On the other hand, if constant query time is required one can not do much better than the naive approach, i.e. an $O(n^2)$ time and space preprocessing of the input digraph (using e.g. the algorithm in [6]), such that a distance query is answered in $O(1)$ time and a shortest path one in $O(L)$ time, where L is the number of edges of the path. Updating this data structure after an edge cost modification takes $O(n^2)$ time which is equivalent to recomputing the data structure from scratch.

Hence, the interesting question arising is: Can we do better than these two extremes? In particular, can we build a dynamic data structure for the DSP problem on outerplanar digraphs in $o(n^2)$ time and space, such that a query can be answered in $o(\log n)$ time and also the data structure can be updated in appropriately short (say sublinear) time after an edge cost modification?

In this paper, we give an affirmative answer to the above question. More precisely, we present two families of algorithms with $o(\log n)$ query time that achieve an interesting trade-off between preprocessing, update, and query bounds, depending on the choice of a particular parameter ε, $0 < \varepsilon < (1/2)$. The results are stated in Theorems 1 and 2. For the case of constant ε, our results as well as their comparison with previous work are summarized on Table 1. (We would like to mention here that the preprocessing bound has been improved very recently [2] through a completely different method. However, the algorithms presented here are much simpler compared with the ones in [2].)

	[3]	Naive approach	This paper	This paper
Dynamic	Yes	No	Yes	Yes
Preprocessing Time & Space	$O(n)$	$O(n^2)$	$O(n \log\log n)$	$O(n^{1+\varepsilon})$
Single-Pair Dist. Query	$O(\log n)$	$O(1)$	$O(\log\log n)$	$O(1)$
Single-Pair SP Query	$O(L + \log n)$	$O(L)$	$O(L + \log\log n)$	$O(L)$
Update Time	$O(\log n)$	$O(n^2)$	$O(n^{2\varepsilon})$	$O(n^{2\varepsilon})$

TABLE 1: Comparison of results for outerplanar digraphs in the case where ε is an arbitrary constant $0 < \varepsilon < (1/2)$.

Our approach is actually a (non-trivial) generalization of the method given in [3] and is based on: (i) a multilevel decomposition strategy based on graph separators; and (ii) on a sparsification of the input digraph where we keep shortest path information between properly chosen $\Theta(n)$ pairs of vertices.

As a main application of our algorithm we give faster algorithms for shortest path problems in planar digraphs. In the final section of this paper we list also other extensions and generalizations of our results.

2 Preliminaries and Data Structures

Let $G = (V(G), E(G))$ be a connected n-vertex digraph with real edge costs but no negative cycles. A *separation pair* is a pair (x, y) of vertices whose removal divides G into two disjoint subgraphs G_1 and G_2. We add the vertices x, y and the edges $\langle x, y\rangle$ and $\langle y, x\rangle$ to both G_1 and G_2. Let $0 < \alpha < 1$ be a constant. An *α-separator* S of 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 vertices of $D(S)$ such that the removal of $V(S)$ leaves no connected component of more than αn vertices. We will call the separation vertices (pairs) of S that belong to any such resulting component H and separate it from the rest of the graph separation vertices (pairs) *attached to H*. It is well known that if G is outerplanar then there exists a 2/3-separator of G which is a single separation pair.

In the sequel, we assume w.l.o.g. that G_o is a biconnected n-vertex outerplanar digraph. Note that if G_o is not biconnected we can add an appropriate number of additional edges of very large costs in order to convert it into a biconnected outerplanar digraph (see e.g. [6]).

An l-decomposition of G_o, where l is an arbitrary positive integer, is a decomposition of G_o into $O(l)$ subgraphs such that (i) each subgraph is of size $O(n/l)$, and (ii) the number of separation pairs attached to each subgraph is $O(1)$. We say that the $O(n/l)$ separation pairs whose removal divides G_o into $O(l)$ subgraphs are *associated with G_o*.

Let λ be an arbitrary function over positive integers whose values are positive integers. Suppose that we compute a $\lambda(n)$-decomposition of G_o and then recursively find a $\lambda(n_i)$-decomposition of each resulting connected component G_i of n_i vertices until we get subgraphs of constant size. We associate a tree with the above decomposition as follows: at each level of recursion, the node associated with G_i is parent of the roots of the trees corresponding to the components of the $\lambda(n_i)$-decomposition of G_i. We call the resulting tree a *λ-decomposition tree* of G_o. Note that each subgraph G_i has $O(1)$ separation pairs attached to it and $O(n_i/\lambda(n_i))$ separation pairs associated with it. We describe a recursive algorithm that constructs a λ-decomposition tree $DT(G_o)$ of G_o that will be used in the construction of a suitable data structure for maintaining shortest path information in G_o. At each level of the recursion, the algorithm recursively decomposes an n_i-vertex graph G_i into $O(\lambda(n_i))$ subgraphs and builds the corresponding part of the decomposition tree.

Let in the algorithm below $\hat{G}$ denote an $\hat{n}$-vertex subgraph of G_o (initially $\hat{G} := G_o$).

ALGORITHM Decomp_Tree($\hat{G}, \lambda, DT(\hat{G})$)
BEGIN

1. For any connected component K of $\hat{G}$ with $|V(K)| > \hat{n}/\lambda(\hat{n})$ do Steps 1.1 and 1.2.

1.1. Denote by S the set of separation pairs (2/3-separators) in $\hat{G}$ found during all previous iterations and denote by n_{sep} the number of separation pairs of S attached to K. Check which of the following cases applies.

1.1.1. If $n_{sep} \leq 3$, then let $p = \{p_1, p_2\}$ be a separation pair of K that divides K into two subgraphs K_1 and K_2 with no more than $2\hat{n}/3$ vertices each.

1.1.2. Otherwise ($n_{sep} > 3$), let $p = \{p_1, p_2\}$ be a separation pair that separates K into subgraphs K_1 and K_2 each containing no more than 2/3 of the number of separation pairs attached to K.

1.2. Add p to S.

2. Find a λ-decomposition tree of each component of $\hat{G}$ by running this algorithm recursively.

3. Create a separator tree $DT(\hat{G})$ rooted at a new node associated with all separation pairs p found in Step 1.1 and whose children are the roots of the λ-decomposition trees of the components of $\hat{G}$.

END.

Following [3], we can implement each recursive step of the algorithm in $O(\hat{n})$ time and space. It is easy to see that by choosing $\lambda(n) = n^{\varepsilon}$, for any $0 < \varepsilon < (1/2)$, the depth of the $DT(G_o)$ is $O((1/\varepsilon)\log\log n)$. Hence, we have:

Lemma 1. *Let $\lambda(n) = n^{\varepsilon}$, where $0 < \varepsilon < (1/2)$ is an arbitrary number. Algorithm Decomp_Tree($G_o, \lambda, DT(G_o)$) constructs a λ-decomposition tree of G_o in $O((1/\varepsilon)n\log\log n)$ time and $O((1/\varepsilon)n\log\log n)$ space.*

Given an outerplanar digraph G_o and a set M of vertices of G_o, *compressing G_o with respect to M* means constructing a new outerplanar digraph of $O(|M|)$ size that contains M and such that the distance between any pair of vertices of M in the resulting graph is the same as the distance between the same vertices in G_o [6].

Definition 1. *Let G_o be an n-vertex outerplanar digraph and let $\{p_1, p_2\}$ be a separator pair of G_o that divides G_o into connected components one of which is G. Let S be a set of $\lambda(n)$ separation pairs that divides G into $O(\lambda(n))$ subgraphs K. Let A_K be the set of separator pairs attached to K, $|A_K| = O(1)$. Construct a digraph $SR(G)$ as follows: remove S from G, compress each resulting subgraph K with respect to $(V(S) \cup \{p_1, p_2\}) \cap V(K)$, and join the resulting subgraphs at vertices $V(A_K)$. We call $SR(G)$ the* sparse representative *of G.*

Remark: Let e be an edge with both of its endpoints in A_K. It is clear that e can be shared by at most two subgraphs K. In the above definition, when subgraphs are joined at the vertices of $V(A_K)$, we keep as the cost of e the smallest of the (possibly) two different costs that e may have in the two subgraphs.

3 The Dynamic Shortest Path Algorithm for Outerplanar Digraphs

3.1 The Preprocessing Phase

The preprocessing algorithm constructs the λ-decomposition tree $DT(G_o)$. Each node of $DT(G_o)$ is associated with a subgraph G of G_o along with the set of

separation pairs associate with it (as they are determined by the decomposition procedure), and also contains a pointer to the sparse representative $SR(G)$ of G. The sparse representative $SR(G)$ is computed for all graphs G of $DT(G_o)$. According to Definition 1, $SR(G)$ consists of the union of the compressed versions of G_i with respect to the separation pairs attached to G plus the associated separation pairs dividing G into the subgraphs G_i, where G_i's are the children of G in $DT(G_o)$. Therefore the size of $SR(G)$ is proportional to the number of separation pairs attached to and associated with G. Note that for each leaf of $DT(G_o)$ we have that $SR(G) \equiv G$, since in this case G is of $O(1)$ size. Moreover, for all graphs G of $DT(G_o)$, the preprocessing algorithm also computes all pairs shortest path (APSP) information between the separation pairs associated with G. Thus, in the query phase, we can answer distance queries in $SR(G)$ in constant time and shortest path queries in time proportional to the number of edges of the path.

In the following, let $\lambda(n) = n^{\varepsilon}$, where $0 < \varepsilon < (1/2)$ is any arbitrary number.

ALGORITHM Pre_1(G_o)
BEGIN
1. Construct a λ-decomposition tree $DT(G_o)$.
2. Compute the sparse representative $SR(G_o)$ of G_o as follows.
   for each child G of G_o in $DT(G_o)$ **do**
(a) **if** G is a leaf of $DT(G_o)$ **then** $SR(G) = G$
   else find $SR(G)$ by running Step 2 recursively on G.
(b) Construct the sparse representative of G_o as described in Definition 1 by using the sparse representatives of the children of G_o.
(c) Run an APSP algorithm on $SR(G_o)$ storing the shortest path information among the $O(\lambda(n))$ separation vertices in a table.
3. Generate a table with entries $[v, G_l]$, where G_l is the leaf subgraph of $DT(G_o)$ containing v. Construct a similar table for the edges of G_o.
4. Preprocess $DT(G_o)$ (using e.g. the algorithm in [9]) such that lowest common ancestor queries can be answered in $O(1)$ time.

END.

From the discussion preceding the algorithm and Lemma 1, we have:

Lemma 2. *Let $\lambda(n) = n^{\varepsilon}$, where $0 < \varepsilon < (1/2)$ is any arbitrary number. Algorithm Pre_1(G_o) takes $O((1/\varepsilon)n \log\log n)$ time and space.*

3.2 Answering a Query

The query algorithm computes the distance between any two vertices v and z of G_o and proceeds as follows. First, use $DT(G_o)$ to find a subgraph G of G_o such that there exists at least one separation pair associated with G that separates v from z. Let $P_{12} = (p_1, p_2)$ be a separation pair associated with G such that v and p_1, p_2 belong to the same child subgraph of G and also P_{12} separates v from z.

Let $P_{34} = (p_3, p_4)$ be another separation pair associated with G which separates v from z and moreover, z and p_3, p_4 belong to the same child subgraph of G. (Note that P_{12} and P_{34} may coincide.) Let $d(v, z)$ denote the distance between v and z. Then clearly,

$$d(v,z) = \min\{\min\{d(v,p_1) + d(p_1,p_3) + d(p_3,z), d(v,p_1) + d(p_1,p_4) + d(p_4,z)\}, \\ \min\{d(v,p_2) + d(p_2,p_3) + d(p_3,z), d(v,p_2) + d(p_2,p_4) + d(p_4,z)\}\}. \quad (1)$$

Hence, for answering the query it suffices to compute the distances $d(v, p_1)$, $d(p_3, z)$, $d(v, p_2)$, $d(p_4, z)$ and $D(P_{12}, P_{34})$, where $D(P_{12}, P_{34})$ denotes the set of all four distances from a vertex in P_{12} to a vertex in P_{34}. In order to do this we will need the shortest path information stored in the tables of the sparse representatives.

Now we discuss how one can use the information the sparse representatives provide. Let $s = (s_1, s_2)$ be any separation pair *attached to* G. The distance from s_1 to s_2 in $SR(G)$ is, by the preprocessing algorithm, equal to the distance between s_1 and s_2 in G. Note that, in general, the distance from s_1 to s_2 in G might be different from the distance between these vertices in G_o. Before we present the query algorithm, we give a way to determine the distances in G_o between the vertices of certain separation pairs that are used in the computation of the distance between v and z. Let G_v be the subgraph associated with the leaf node of $DT(G_o)$ that contains v. Let $D(G_v)$ be the set of all distances in G_o between the vertices of the separation pairs attached to ancestors of G_v (including G_v itself) in $DT(G_o)$. Then $D(G_v)$ can be found by the following algorithm.

ALGORITHM Attached_Pairs(G_v)
BEGIN

1. Let G' be the parent of G_v in $DT(G_o)$. If $G' = G_o$ then $D(G') := \emptyset$; otherwise compute recursively $D(G')$ by this algorithm.

2. For each separation pair (s'_1, s'_2) attached to G', find $d(s'_1, s'_2)$ and $d(s'_2, s'_1)$ in G_o by using the tables of $SR(G')$ and the information in $D(G')$. Set $D(G_v) := D(G') \cup \{d(s'_1, s'_2), d(s'_2, s'_1)\}$.

END.

Algorithm Attached_Pairs can be used to compute the distances in G_o between the vertices of all separation pairs attached to G_v or to any ancestor of G_v in $DT(G_o)$, so that one can ignore the rest of G_o when computing distances in G_v or in one of its ancestors. (As a consequence, we can also compute the correct distances in $D(P_{12}, P_{34})$ in $O(1)$ time, using the tables of the sparse representative of G.) It is not hard to see that the running time of the above algorithm is $O((1/\varepsilon) \log \log n)$ (i.e. proportional to the depth of $DT(G_o)$).

Next we describe the query algorithm. Let v' be a vertex that belongs to the same subgraph G_v of G_o that is a leaf of $DT(G_o)$ and that contains v. Let $p(v)$ be the pair of vertices v, v'. Similarly define a pair of vertices $p(z)$ that contains z and a vertex z' which belongs to the leaf G_z of $DT(G_o)$ containing z. Then (1) shows that $D(p(v), p(z))$ can be found in constant time, given $D(p(v), P_{12})$,

$D(P_{12}, P_{34})$ and $D(P_{34}, p(z))$. The following recursive algorithm is based on the above fact.

ALGORITHM Query_1(G_o, v, z)
BEGIN

1. Find the subgraphs G_v and G_z as defined above.

2. Run the Algorithm Attached_Pairs on G_v and on G_z .

3. Find pairs of vertices $p(v)$ and $p(z)$ as defined above.

4. Find a subgraph G of G_o such that there exist separation pairs P_{12}, P_{34} associated with G (as defined above) that separate $p(v)$ and $p(z)$ in G. Use the information found at step 2 and the tables of $SR(G)$ to compute the correct distances in $D(P_{12}, P_{34})$.

5. Find $D(p(v), P_{12})$ as follows:

5.1. Let G' be the child of G in $DT(G_o)$ that contains $p(v)$ and P_{12}. If G' is a leaf of $DT(G_o)$, then determine $D(p(v), P_{12})$ directly in constant time.

5.2. If G' is not a leaf then find the child G'' of G' that contains $p(v)$. For each separation pair p' attached to G'' do the following. Compute $D(p(v), p')$ by executing Step 5 recursively with $P_{12} := p'$, and then find $D(p(v), P_{12})$ using (1). (Note that $D(p', P_{12})$ can be taken from the tables of $SR(G')$.) Keep as $D(p(v), P_{12})$ the minimum of the computed distances.

6. Find $D(P_{34}, p(z))$ as in Step 5.

7. Use $D(p(v), P_{12})$, $D(P_{12}, P_{34})$, $D(P_{34}, p(z))$ and (1) to find $D(p(v), p(z))$.

END.

Steps 1, 3 and 4 of algorithm Query_1 take $O(1)$ time by the preprocessing of G_o. Step 2 takes $O((1/\varepsilon) \log\log n)$ time (as discussed above). It is not difficult to see that each recursive execution of Step 5 takes $O(1)$ time and the depth of the recursion is bounded by the depth of $DT(G_o)$. Thus, we have:

Lemma 3. *Algorithm Query_1(G_o, v, z) finds the distance between any two vertices v and z of an n-vertex outerplanar digraph G_o in $O((1/\varepsilon) \log\log n)$ time.*

Algorithm Query_1 can be modified in order to answer path queries. The additional work (compared with the case of distances) involves uncompressing the shortest paths corresponding to edges of the sparse representatives of the graphs from $DT(G_o)$. Uncompressing an edge from a graph $SR(G)$ involves a traversal of a subtree of $DT(G_o)$, where at each step an edge is replaced by $|G|^{\varepsilon}$ new edges each possibly corresponding to a compressed path. Obviously this subtree will have no more than L leaves, where L is the number of the edges of the output path. Then the traversal time can not exceed the number of the vertices of a binary tree with L leaves in which each internal node has exactly 2 children. Any such tree has $2L - 1$ vertices. Thus the next claim follows.

Lemma 4. *The shortest path between any two vertices v and z of an n-vertex outerplanar digraph G_o can be found in $O(L + (1/\varepsilon) \log\log n)$ time, where L is the number of edges of the path.*

3.3 Updating the Data Structures

In the sequel, we will show how we can update our data structures for answering shortest path and distance queries in outerplanar digraphs, in the case where an edge cost is modified. 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 $O((1/\varepsilon)\log\log n)$ subgraphs of G_o, as they are determined by Algorithm Pre_1. Therefore, it suffices to update (in a bottom-up fashion) the sparse representatives, as well as their tables, of those subgraphs that are on the path from the subgraph G_l containing e (where G_l is a leaf of $DT(G_o)$) to the root of $DT(G_o)$. Let $parent(G)$ denote the parent of a node G in $DT(G_o)$, and $\hat{G}$ denote any sibling of a node G in a $DT(G_o)$ such that G and $\hat{G}$ have a common separation pair attached to them. (Note also that an edge e can belong to at most one other sibling of G_l.) The algorithm for the update operation is the following.

ALGORITHM Update_1$(G_o, e, w(e))$
BEGIN
1. Find a leaf G of $DT(G_o)$ for which $e \in E(G)$.
2. Update the cost of e in G with the new cost $w(e)$.
3. If e belongs also to some $\hat{G}$ then update the cost of e in $\hat{G}$.
4. **While** $G \neq G_o$ **do**
   (a) Update $SR(parent(G))$ by using the new versions of $SR(G)$ and $SR(\hat{G})$ and then by running an APSP algorithm on it.
   (b) $G := parent(G)$.
END.

The first three steps of the above algorithm require $O(1)$ time. Let $U(n)$ be the maximum time required by Step 4. Then it is clear that after updating recursively the child subgraph of G_o containing edge e, we need $O(\lambda(n))$ time to recompute $SR(G_o)$ plus $O(\lambda^2(n))$ time to recompute the APSP tables of $SR(G_o)$. Hence, $U(n) \leq U(n/\lambda(n)) + O(\lambda^2(n))$. Letting $\lambda(n) = n^\varepsilon$, we have:

Lemma 5. *Let $\lambda(n) = n^\varepsilon$, where $0 < \varepsilon < (1/2)$ is an arbitrary number. Algorithm Update_1 updates after an edge cost modification the data structures created by the preprocessing algorithm in $O(f(\varepsilon)n^{2\varepsilon})$ time, where $f(\varepsilon) = 1$, if ε is a constant (independent of n), or $f(\varepsilon) = (1/\varepsilon)\lceil\log\log n\rceil$, if ε depends on n.*

Summarizing all the results in Section 3, we get:

Theorem 1. *Given an n-vertex outerplanar digraph G_o with real-valued edge costs but no negative cycles and an arbitrary number $0 < \varepsilon < (1/2)$, there exists an algorithm for maintaining all pairs shortest paths information in G_o under any edge cost modification, with the following performance characteristics: (i) preprocessing time and space $O((1/\varepsilon)n\log\log n)$; (ii) single-pair distance query time $O((1/\varepsilon)\log\log n)$; (iii) single-pair shortest path query time*

$O(L + (1/\varepsilon)\log\log n)$ *(where L is the number of edges of the path); (iv) update time (after an edge cost modification)* $O(f(\varepsilon)n^{2\varepsilon})$*, where* $f(\varepsilon) = 1$*, if* ε *is independent of* n*, or* $f(\varepsilon) = (1/\varepsilon)\lceil\log\log n\rceil$ *otherwise.*

4 Improving More on the Query Time

In this section we shall describe how the algorithms presented in the previous section can be modified such that a distance query is answered in $O(1)$ time. In the following, let $\lambda(n) = n^{\varepsilon}$, for some arbitrary number $0 < \varepsilon < (1/2)$.

We change the first step of the preprocessing algorithm as follows. Instead of dividing each child subgraph H of G_o into $\lambda(n/\lambda(n))$ subgraphs, we can divide it into $\lambda(n)$ subgraphs. This will reduce the depth of $DT(G_o)$ to $O(1/\varepsilon)$. However, notice that now all the descendant subgraphs of G_o which are leaves of $DT(G_o)$ are of size $O(n^{\varepsilon})$ and there are $O(n^{1-\varepsilon})$ of them. We shall run on these subgraphs an APSP algorithm. Call the new preprocessing algorithm Pre_2(G_o). Hence, we have the following:

Lemma 6. *Algorithm Pre_2(G_o) runs in* $O((1/\varepsilon)n + n^{1+\varepsilon})$ *time and uses* $O((1/\varepsilon)n + n^{1+\varepsilon})$ *space.*

A query is answered in the same way as before. But since now the depth of $DT(G_o)$ is $O(1/\varepsilon)$, we can answer a query in this time.

The data structures can be updated (after an edge cost modification) using the same approach as in Section 3.3. This means that we need $O(1/\varepsilon)$ iterations and for each $SR(G)$ of a descendant subgraph G we have to run an APSP algorithm for updating the shortest path information among the separation pairs associated with it. Also we have to run the APSP algorithm to the leaf subgraph of G_o containing the edge whose cost has been modified. This will give us a total of $O((1/\varepsilon)n^{2\varepsilon})$ time for updating our data structures.

The above discussion leads to the following.

Theorem 2. *Given an n-vertex outerplanar digraph G_o with real-valued edge costs but no negative cycles and an arbitrary number $0 < \varepsilon < (1/2)$, there exists an algorithm for maintaining all pairs shortest paths information in G_o under any edge cost modification with the following performance characteristics: (i) preprocessing time and space $O((1/\varepsilon)n + n^{1+\varepsilon})$; (ii) single-pair distance query time $O(1/\varepsilon)$; (iii) single-pair shortest path query time $O(L+1/\varepsilon)$ (where L is the number of edges of the path); (iv) update time (after an edge cost modification) $O((1/\varepsilon)n^{2\varepsilon})$.*

5 Extensions of our Results

The algorithms for the DSP problem for outerplanar digraphs we described in this paper can be used for constructing faster algorithms for planar digraphs. The approach used is the same as the one in [3] and is (partially) based on

the hammock decomposition technique introduced by Frederickson in [5, 6]. This technique allows the reduction of the shortest paths problems on planar digraphs with nice topology to similar problems on outerplanar digraphs. Our results for planar digraphs can be obtained by incorporating the results of Theorems 1 and 2 into the algorithms of [3]. (We omit details due to space limitations and the interested reader is referred to [3].)

We mention also the following extensions and generalizations of our results: (i) We can handle efficiently edge deletions. (Note that deletion of an edge e is equivalent to assigning a very large cost to e so that no shortest path will use e.) (ii) Our algorithms can detect a negative cycle (in a way similar to that described in [3]), either if it exists in the initial digraph, or if it is created after an edge cost modification. (iii) Using the ideas of [5], our results can be used to design improved algorithms for the DSP problem on digraphs with small genus. (iv) Although our algorithms do not directly support edge insertion, they are fast enough so that even if the preprocessing algorithm is run from scratch after any edge insertion, they still provide better performance compared with the naive approach. Moreover, our algorithms can support a special kind of edge insertion, called *edge re-insertion*. That is, we can insert any edge that has previously been deleted within the resource bounds of the update operation.

References

1. H. Bondlaender, "Dynamic Algorithms for Graphs with Treewidth 2", *Proc. 19th WG'93*, LNCS 790, pp.112-124, Springer-Verlag, 1994.
2. S. Chaudhuri and C. Zaroliagis, "Shortest Path Queries in Digraphs of Small Treewidth", *Proc. 22nd ICALP*, LNCS, Springer-Verlag, 1995, to appear.
3. H. Djidjev, G. Pantziou and C. Zaroliagis, "On-line and Dynamic Algorithms for Shortest Path Problems", *Proc. 12th STACS*, LNCS 900, pp.193-204, Springer-Verlag, 1995.
4. E. Feuerstein and A.M. Spaccamela, "Dynamic Algorithms for Shortest Paths in Planar Graphs", *Theor. Computer Science*, 116 (1993), pp.359-371.
5. G.N. Frederickson, "Using Cellular Graph Embeddings in Solving All Pairs Shortest Path Problems", *Proc. 30th Annual IEEE Symp. on FOCS*, 1989.
6. G.N. Frederickson, "Planar Graph Decomposition and All Pairs Shortest Paths", *J. ACM*, Vol.38, No.1, January 1991, pp.162-204.
7. R. Hassin, "Maximum flow in (s, t)-planar networks", *Inform. Proc. Lett.*, 13(1981), p.107.
8. G. Miller and J. Naor, "Flows in planar graphs with multiple sources and sinks", *Proc. 30th IEEE Symp. on FOCS*, 1989, pp.112-117.
9. B. Schieber and U. Vishkin, "On Finding Lowest Common Ancestors: Simplification and Parallelization", *SIAM J. Computing*, 17(6), pp.1253-1262, 1988.