Improved Algorithms for Dynamic Shortest Paths

Hristo N. Djidjev*
Department of Computer Science, University of Warwick,
Coventry CV4 7AL, UK
Email: hristo@dcs.warwick.ac.uk

Grammati E. Pantziou†
Computer Technology Institute,
P.O. Box 1122, 26110 Patras, Greece
Email: pantziou@cti.gr

Christos D. Zaroliagis‡
Department of Computer Science, King's College,
University of London, Strand, London WC2R 2LS, UK
Email: zaro@dcs.kcl.ac.uk

January 20, 1999

Abstract

We describe algorithms for finding shortest paths and distances in outerplanar and planar digraphs that exploit the particular topology of the input graph. An important feature of our algorithms is that they can work in a dynamic environment, where the cost of any edge can be changed or the edge can be deleted. In the case of outerplanar digraphs, our data structures can be updated after any such change in only logarithmic time. A distance query is also answered in logarithmic time. In the case of planar digraphs, we give an interesting trade-off between preprocessing, query and update times depending on the value of a certain topological parameter of the graph. Our results can be extended to n-vertex digraphs of genus $O(n^{1-\varepsilon})$ for any $\varepsilon > 0$.

Keywords: Shortest path, dynamic algorithm, planar digraph, outerplanar digraph.

---

*The work of this author was partially supported by the NSF grant No. CCR-9409191. Part of the work was done while the author was with the Department of Computer Science, Rice University, Houston, USA.

†The work of this author was partially supported by the EU ESPRIT LTR Project No. 20244 (ALCOM-IT).

‡The work of this author was partially supported by the EU ESPRIT LTR Project No. 20244 (ALCOM-IT). Part of the work was done while the author was with the Max-Planck-Institut für Informatik, Saarbrücken, Germany.

1 Introduction

1.1 The problem and its motivation

There has been a growing interest in dynamic graph problems in the recent years. The goal is to design efficient data structures that not only enable fast answering to a series of queries, but that can also be easily updated after a modification of the input data. Such an approach has immediate applications to a variety of problem domains which are of both theoretical and practical value, including (among others) communication networks, high level languages for incremental computations [40], incremental data flow analysis [6], database and knowledge base systems [1, 39], and programming environments [27].

Finding shortest path information in graphs is an important and intensively studied problem with many applications. Given an n-vertex digraph G with real valued edge costs but no negative cycles, the shortest path problem is to find paths of minimum length between any two vertices of a given set of vertices, where the *length* of a path is the sum of the costs of its edges. The length of a shortest path between vertices v and w is called the *distance* from v to w and will be denoted by $d(v, w)$. There are two main versions of the shortest path problem: the *all-pairs shortest paths* in which we look for shortest paths between every pair of vertices in G; and the *single-source shortest path* in which we look for shortest paths between a specific vertex and all other vertices in G. Recent papers [9, 13, 18, 19, 21, 20, 23, 29, 32, 34] investigate the shortest path problem for different classes of input graphs and models of computation. All of these 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, in this paper we investigate the following *dynamic (all-pairs) shortest path* problem: Given G (as above), build a data structure that will enable fast answering of on-line shortest path or distance queries, where a shortest path (resp. distance) query specifies two vertices and asks for the shortest path (resp. distance) between them. In case of edge deletion or edge cost modification of G, update the data structure efficiently, i.e., without recomputing everything from scratch. Note that the problem must be solved in an on-line fashion, i.e., each operation (query or update) must be performed before the next one is known.

The dynamic version of the shortest path problem has several applications including dynamic maintenance of a maximum s-t flow in a planar network [26], computing a feasible flow between multiple sources and sinks [33], as well as finding a perfect matching in bipartite planar graphs [33]. Dynamic algorithms for shortest paths appear also to be fundamental procedures in incremental computations for data flow analysis and interactive systems design [35, 40].

1.2 Previous results

There are a few previously known algorithms for the dynamic shortest path problem. For general digraphs with real edge costs, the best previous algorithms, in the case of edge insertions, edge deletions and edge cost updates, are given in [15, 36]. The data structure in [15, 36] is updated in $O(n^2)$ time after an edge insertion or edge cost decrease, and in $O(nm + n^2 \log n)$ time after an edge deletion or edge cost increase (m being the current number of edges in the graph). Note that the update time after an edge deletion or edge cost increase is equal to the time required to recompute all pairs shortest paths from scratch [23]. Improvements on these algorithms have been achieved in [5] with respect to the worst-case complexity of a sequence of edge insertions or edge cost decreases (thus providing a better bound per update in the amortized sense), in the special case where the edge costs are nonnegative integers. More specifically, the data structure in [5] can be updated in $O(Cn^3 \log(nC))$ time after a sequence of at most $O(n^2)$ edge insertions or at most $O(Cn^2)$ edge cost decreases, where C is the largest value of an edge cost. (Edge deletions or edge cost increases are not considered in [5].)

For the important case of planar digraphs with nonnegative real edge costs, dynamic algorithms for the shortest path problem are given in [17]. The preprocessing time and space is $O(n \log n)$. (The space can be reduced to $O(n)$, if the computation is restricted to finding distances only.) A shortest path or distance query can be answered in $O(n)$ time. An update operation to this data structure, after an edge cost modification or edge deletion, can be performed in $O(\log^3 n)$ time. This result, however, has been superseded by the $O(n)$ time single-source shortest path algorithm for planar digraphs with nonnegative real edge costs given in [31]. Also in that paper [31] a dynamic algorithm is given for shortest paths in planar digraphs with integral edge costs (which may be negative). More precisely, the algorithm initializes an $O(n)$-size data structure in $O(n^{10/7})$ time. The time for a query or an update operation is $O(n^{9/7} \log(nL))$, where the edge costs are integers larger than $-L$ ($L \geq 1$). This time bound is worst-case for queries as well as for edge cost modifications and edge deletions, while for edge insertions it is amortized. Other dynamic algorithms for the shortest path problem are known for special classes of digraphs [7].

On the other hand, efficient data structures for answering very fast on-line shortest path or distance queries in planar digraphs with real edge costs have been proposed in [13, 21], but they do not support dynamization.

1.3 Our results

In this paper, we give efficient algorithms for solving the dynamic shortest path problem in outerplanar and planar digraphs. Our main result is an algorithm for outerplanar digraphs with real edge costs (but no negative cycles) that has a preprocessing phase during which an $O(n)$ size data structure is constructed in $O(n)$ time. A distance query can be answered in $O(\log n)$ time, and the data structure can be updated in $O(\log n)$ time after an edge cost modification or edge deletion.

Our solution is based on the following idea. We employ a decomposition strategy based on graph separators that divides the outerplanar digraph into subgraphs sharing a constant number of vertices with the rest of the graph. However, maintaining global shortest path information even for separator vertices may be too expensive. Instead, our data structure maintains only local – to a particular subgraph – information regarding shortest paths between separator vertices. This information is encoded into specially constructed, constant size, sparse abstractions of these subgraphs. To answer a query, distances between relevant separator vertices are computed and passed into the relevant local subgraph.

Based on the above result for outerplanar digraphs and using the hammock decomposition approach pioneered by Frederickson [19, 20], we give two algorithms for planar digraphs that are parameterized in terms of a topological measure q of the input digraph. Informally, q represents the minimum number of outerplanar subgraphs (called hammocks), satisfying certain separation properties, into which a planar graph can be decomposed.

Our first result for n-vertex planar digraphs with nonnegative real edge costs is a dynamic algorithm that, after an $O(n)$ time and space preprocessing, answers a distance query in $O(\log n + q)$ time. The data structures set up during preprocessing can be updated (after an edge cost modification or edge deletion) in $O(\log n)$ time.

The parameter q ranges from 1 up to $\Theta(n)$, depending on how complex the topological properties of the input graph become. It defines a natural hierarchy of planar graphs [22] that appears to be very important since it generalizes outerplanar graphs (for which $q = 1$) and has been proved crucial in the design of space-efficient methods for message routing in communication networks [22]. Hence, our result is always competitive with the best previous ones, and it is better in all cases where $q = o(n)$. Classes of graphs with a small value of q are the planar graphs which satisfy the $o(n)$-interval property as they are defined in [22]. Yet another class of graphs are the graphs describing global area networks. Typically such graphs can be represented as a tree plus a small number of non-tree edges and thus have a small value of q. Also, our algorithms seem to be very efficient for the class of all appropriately *sparse* graphs. As it has been established in [16, 30] random $G_{n,p}$ graphs with threshold function $1/n$ are planar with probability one and have *expected value* for q equal to $O(1)$.

Our second algorithm for planar digraphs is based on the hammock decomposition, our result for outerplanar digraphs, and on some recent results achieved in [4, 12] regarding non-dynamic shortest paths in planar digraphs. It has slower preprocessing and update time compared to our first algorithm, but has a sublinear query time regardless of the value of q. More precisely, the preprocessing phase of our second algorithm takes $O(n+q^{3/2})$ time and space, after which a distance query can be answered in $O(\log n + q^{1/2})$ time. The data structures set up during preprocessing can be updated (after an edge cost modification or edge deletion) in time $O(\log n + q^{3/2})$.

All of our algorithms can answer a shortest path query in additional $O(L)$ time, where L is the number of edges of the reported path. We mention also the following extensions and generalizations of our results discussed in the paper.

(i) We have constructed parallel versions of our algorithms for the CREW PRAM model of parallel computation (Section 5). We are not aware of any previous parallel (NC) algorithms for the dynamic version of the shortest path problem for planar digraphs.

(ii) In the case of outerplanar digraphs, our algorithms can detect a negative cycle either if it exists in the initial digraph, or if it is created after an edge cost modification (Section 3). Moreover, our data structures allow us to solve in linear time the single-source shortest path problem in outerplanar digraphs with real edge costs (but no negative cycles), thus matching the bounds given in [19, 21] for the same problem.

(iii) Using the ideas in [20], our results can be extended to hold for any digraph whose genus is $O(n^{1-\varepsilon})$ for any $\varepsilon > 0$.

(iv) Although our algorithms do not directly support edge insertion, they are efficient enough so that even if the preprocessing algorithm is run from scratch after any edge insertion, they still compare favorably with the best previous results. 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.

The paper is organized as follows. In Section 2 we give some definitions and preliminary results. In Section 3 we present our algorithms for outerplanar digraphs. Our results for planar digraphs are given in Section 4. Finally, in Section 5 we describe the extensions and generalizations of our results. Preliminary portions of this work appeared in [14].

2 Preliminaries

We assume that the reader is familiar with standard graph-theoretic terminology as contained e.g., in [2, 25]. A graph is called *outerplanar* if it can be embedded in the plane such that all of its vertices lie on one face. Since we concentrate in this paper on the shortest path problem, we shall consider directed graphs (digraphs) whose edges are associated with real edge costs. Let $G = (V(G), E(G))$ be such a digraph which we also assume is connected. (When we discuss connectivity or biconnectivity issues for a digraph G, we shall ignore the direction of the edges.) By $G + e$ we will denote the graph $(V(G), E(G) \cup \{e\})$, where e is an edge with endpoints in $V(G)$. For a subgraph H of G and vertices $v, w \in V(H)$, we shall denote by $d_H(v, w)$ the distance of a shortest path from v to w in H. For $V' \subseteq V(G)$, the subgraph of G *induced* by V' is defined by $G[V'] = (V', E(G) \cap (V' \times V'))$, i.e., the set of edges of $G[V']$ is the set of all edges of G with both endpoints in V'. A *separation pair* of G is a pair (x, y) of vertices whose removal (along with their incident edges) results in a graph with at least two connected components. We shall refer to this operation as *division* of G using the pair (x, y).

Let $0 < \alpha < 1$ be a constant. An *α-separator* of G is a set $S \subseteq V(G)$ whose removal leaves no connected component with more than $\alpha|V(G)|$ vertices. It is well known that if G is outerplanar, then there exists a 2/3-separator of G which is a single separation pair [8]. Such a separation pair can be found by triangulating all internal faces of G and finding a separator edge in the dual graph of the resulting embedding, excluding the outer face, which dual graph is a tree.

Throughout the paper let G_{op} denote an n-vertex outerplanar digraph with real edge costs. We can assume w.l.o.g. that G_{op} is biconnected. (For the shortest path problem investigated in this paper it is easy to overcome this restriction: if G_{op} is not biconnected, then we can add an appropriate number of edges to make it biconnected and assign to these edges a very large cost such that they will not be used by any shortest path in G_{op}; see e.g., [19].) It is easy to verify that biconnectivity of G_{op} ensures that when we use a separation pair (x, y) to divide G_{op} we are left with exactly two connected components.

Our data structure for shortest path computations will be based on a recursive decomposition of an outerplanar graph using a modification of the division operation defined above. In the new operation, when a graph G_{op} is divided into subgraphs G_1 and G_2 using a separation pair (p_1, p_2), p_1 and p_2 will be added to both G_1 and G_2 as well as new edges between p_1 and p_2 (if not already there). These edges, whose addition ensures biconnectivity of G_1 and G_2, will later receive appropriate costs so that the distances between p_1 and p_2 and the other vertices of G_{op} can be more easily computed and stored. This leads us to the following definition.

Let (p_1, p_2) be any separation pair of G_{op}. *Splitting G_{op} into G_1 and G_2 at (p_1, p_2)* consists of dividing G_{op} using (p_1, p_2) into two components with vertex sets V_1 and V_2, and defining the graphs $G_1 = G_{op}[V_1 \cup \{p_1, p_2\}] + \langle p_1, p_2 \rangle + \langle p_2, p_1 \rangle$ and $G_2 = G_{op}[V_2 \cup \{p_1, p_2\}] + \langle p_1, p_2 \rangle + \langle p_2, p_1 \rangle$.

Let $M \subset V(G_{op})$ be a set of vertices in G_{op}. Define the *compression of G_{op} with respect to M* to be a new outerplanar digraph of size $O(|M|)$ that contains M and such that the distance between any pair of vertices of M in the resulting digraph is the same as the distance between the same vertices in G_{op}. Such a compression was first considered in

[19] for the case $|M| = 4$ and can be found in $O(n)$ time. The method described in that paper can be easily generalized for any bounded value of $|M|$. For the rest of the paper we will assume that the size of M is $O(1)$.

We are now ready to define the key notion of a *sparse representative.*

Definition 2.1 *Let G_{op} be an outerplanar digraph, let $M \subset V(G_{op})$ and let $p = (p_1, p_2)$ be a separation pair of G_{op}. Define a digraph $SR(G_{op})$ as follows: split G_{op} into subgraphs G_1 and G_2 using p and assign to edges $\langle p_1, p_2 \rangle$ and $\langle p_2, p_1 \rangle$ costs equal to the shortest distances in G_{op} between the corresponding vertices; compress each subgraph G_i ($i = 1, 2$) with respect to $(M \cup \{p_1, p_2\}) \cap V(G_i)$; join the resulting (compressed) digraphs at vertices p_1, p_2. We call $SR(G_{op})$ the* sparse representative *of G_{op} with respect to M and p.*

From the methods in [19] and the above definition, it follows that $SR(G_{op})$ is an outerplanar digraph of $O(|M|)$ size, and that if the compressed versions of G_1 and G_2 with respect to $M \cap V(G_1)$ and $M \cap V(G_2)$, respectively, are given, then $SR(G_{op})$ can be constructed in $O(|M|)$ time.

Sparse representatives will be used to store distances between any two members of a set of "relevant" vertices in a subgraph. These relevant vertices, corresponding to the vertices of M, will be the vertices in the separation pairs belonging to G_{op} (and incident to vertices not in G_{op}) that have been used for a recursive decomposition of the original outerplanar graph during the preprocessing step.

To deal with planar graphs we will make use of a graph decomposition technique, called hammock decomposition, introduced by Frederickson in [19, 20]. The *hammock decomposition* of an n-vertex planar digraph G is a decomposition of G into certain outerplanar digraphs called *hammocks.* This decomposition is defined with respect 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 in the plane). It has been proved that a planar digraph G can be decomposed into $q = O(q')$ hammocks either in $O(n)$ sequential time [19], or in $O(\log n \log^* n)$ parallel time and $O(n \log n \log^* n)$ work on a CREW PRAM [34]. Hammocks satisfy the following properties: (i) each hammock has at most *four* vertices in common with the rest of the graph called *attachment vertices*; (ii) the hammock decomposition spans all the edges of G, i.e., each edge belongs to exactly one hammock; and (iii) the number of hammocks produced is the minimum possible (within a constant factor) among all possible decompositions. The hammock decomposition allows us to *reduce* the solution of a given problem Π on a planar digraph to a solution of Π on an outerplanar digraph. We will see an application of this approach in Section 4.

2.1 Constructing a separator decomposition

In this section, we give an algorithm that generates a decomposition of G_{op} (by finding successive separators in a recursive way) that will be used in the construction of a suitable data structure for maintaining shortest path information in G_{op}. Our goal will be that, at each level of recursion, (i) the sizes of the connected components resulting after the deletion of the previously found separator vertices are appropriately small, and (ii) the number of separation vertices attached to each resulting component is $O(1)$. The following algorithm finds such a partition and constructs the associated *separator tree*, $ST(G_{op})$.

In the algorithm below let G denote a subgraph of G_{op} (initially $G := G_{op}$) and let S denote the set of separation pairs in G_{op} found during all iterations (initially $S = \emptyset$).

ALGORITHM Sep_Tree($G, ST(G)$)
BEGIN

1. If $|V(G)| \leq 4$, then stop. Otherwise, compute the number n_{sep} of separation pairs of S whose vertices belong to G and determine which of the following cases applies.

   1.1 If $n_{sep} \leq 3$, then let $p = \{p_1, p_2\}$ be a separation pair of G that splits G into two subgraphs G_1 and G_2 with no more than $2|V(G)|/3$ vertices each.

   1.2 Otherwise ($n_{sep} > 3$), let $p = \{p_1, p_2\}$ be a separation pair that splits G into two subgraphs G_1 and G_2 each containing no more than $2n_{sep}/3$ separation pairs.

2. Add p to S and run Sep_Tree($G_i, ST(G_i)$) for $i = 1, 2$. Create a separator tree $ST(G)$ rooted at a new node v associated with p and G and whose children are the roots of $ST(G_1)$ and $ST(G_2)$.

END.

Observe that the nodes of $ST(G_{op})$ are associated with subgraphs of G_{op}, which we will call *descendant subgraphs* of G_{op}. With each descendant subgraph G a distinct separation pair is associated (the one that splits G in Step 1.1, or 1.2), which we call the *separation pair associated with* G. We call the separation pairs that separate G from the rest of G_{op} *separation pairs attached to* G. From the description of the algorithm, the following statement is immediate.

Proposition 2.1 *Any descendant subgraph G of G_{op} at level $2i$ in $ST(G_{op})$ has no more than 4 separation pairs attached to it and the number of its vertices is no more than $(2/3)^i n$. Any edge of G_{op} belongs to at most two leaf subgraphs of G_{op}.*

Algorithm Sep_Tree can be easily implemented to run in $O(n \log n)$ time and $O(n)$ space. We show by the following lemma that there exists a more efficient implementation in $O(n)$ time and space.

Lemma 2.1 *Algorithm Sep_Tree($G_{op}, ST(G_{op})$) can be implemented to run in $O(n)$ time and $O(n)$ space. The depth of the resulting separator tree $ST(G_{op})$ is $O(\log n)$.*

Proof: Each recursive step of algorithm Sep_Tree takes $O(1)$ time plus time necessary to compute the number n_{sep} and to find the separation pair p in Steps 1.1 and 1.2. The computation of n_{sep} can take $O(1)$ time per recursive step, if in Step 1.1 or in Step 1.2 we keep, for each component K into which S splits G, a list of the separation pairs attached to K. Since we do not allow the number of the separation pairs in any list to exceed 4, we can compute $n_{sep} = |K|$ in $O(1)$ time. Furthermore, in this way we can compute all separation pairs produced by Step 1.2 during the whole execution of the algorithm in $O(n)$ time, since we can trivially update any such list in $O(1)$ time when a new separation pair is attached to K. Hence, the total time needed by all steps of the algorithm is $O(n)$ plus the time required to find all separation pairs p in Step 1.1. Therefore, to complete the proof of the Lemma, we only need to show that the time required to find all separation pairs p from Step 1.1 is $O(n)$.

Construct the dual graph $T(G_{op})$ of G_{op} (excluding the outer face), which is a tree. By using the data structure of Sleator and Tarjan [38] for dynamic trees, which maintains in $O(\log l)$ time a forest of trees of size l under a sequence of cut and link operations, Goodrich and Tamassia show in [24] that the *centroid* edge that divides a k vertex binary tree from such a forest into two subtrees with at most $(2/3)k+1$ vertices each can be found in $O(\log l)$ time. With the algorithm of [24], a separation pair of G_{op} in Step 1.1 can be found in $O(\log n)$ time. Thus, the maximum time $T(n)$ needed to find all separation pairs satisfies the recurrence

$$T(n) \leq \max\{T(n_1) + T(n_2) \mid n_1 + n_2 = n + 2, \quad n_1, n_2 \leq 2n/3\} + O(\log n), \quad n > 1$$

whose solution is $T(n) = O(n)$. Since by Proposition 2.1 $ST(G_{op})$ is a balanced tree, it has a logarithmic depth. ∎

3 Dynamic Algorithms for Outerplanar Digraphs

In this section we will give algorithms for solving the dynamic shortest path problem for the special case of outerplanar digraphs. We will use these algorithms in Section 4 for solving shortest path problems in planar digraphs.

3.1 The data structures and the preprocessing algorithm

The data structures used by our algorithms are the following:

(I) The separator tree $ST(G_{op})$. Each node of $ST(G_{op})$ is associated with a descendant subgraph G of G_{op} along with its (associated) separation pair as determined by algorithm Sep_Tree and also contains a pointer to the sparse representative $SR(G)$ of G. We will often not distinguish between a node in the separation tree and its associated graph.

(II) A sparse representative for each node of the separator tree. Specifically, for any subgraph G of $ST(G_{op})$, we store the sparse representative $SR(G)$ of G with respect to the set of all separation vertices attached to G (corresponding to set M from Definition 2.1) and the separation pair associated with G (corresponding to the pair p that splits G in that definition). Thus, $SR(G)$ is a $O(1)$ size compressed version of G that contains all separation vertices incident to G and adjacent to a vertex not in G. Note also that: (a) since the size of $SR(G)$ is $O(1)$, we can compute distances with respect to G between vertices in $SR(G)$ in constant time; (b) for each leaf of $ST(G_{op})$ we have that $SR(G) \equiv G$, since in this case G is of $O(1)$ size.

In the following sections we will use the properties of the separator decomposition to show that the shortest path information encoded in the sparse representatives of the descendant subgraphs of G_{op} is sufficient to compute the distance between any two vertices of G_{op} in $O(\log n)$ time and that, after any edge cost modification or edge deletion, all affected sparse representatives can be updated also in $O(\log n)$ time. Our preprocessing algorithm for constructing the above data structures is given below. This algorithm will also be used in Section 4 as a subroutine for preprocessing arbitrary planar graphs.

ALGORITHM Pre_Outerplanar(G_{op})
BEGIN

1. Run algorithm Sep_Tree($G_{op}, ST(G_{op})$) to construct a separator tree $ST(G_{op})$ in G_{op}.
2. Run procedure Sparse_Representative(G_{op}) (given below) to compute sparse representatives for G_{op} and its descendant subgraphs in $ST(G_{op})$.
3. Construct tables A and B such that for each $v \in V(G_{op})$ $A[v]$ stores a pointer to a leaf subgraph of $ST(G_{op})$ containing v and for each $e \in E(G_{op})$ $B[e]$ stores pointers to the two leaf subgraphs containing e.
4. Preprocess $ST(G_{op})$ such that lowest common ancestor queries can be answered in $O(1)$ time.

END.

Next we describe the procedure used in Step 2 for computing sparse representatives. It assumes that $ST(G_{op})$ has already been computed. The procedure takes as input a subgraph G of G_{op} and computes $SR(G)$ as well as $SR(G')$ for any G' that is a descendant subgraph of G in $ST(G_{op})$.

PROCEDURE Sparse_Representative(G)
BEGIN

1. **for** each child H of G in $ST(G_{op})$ **do**
   if H is a leaf of $ST(G_{op})$ **then** $SR(H) = H$
   else run Sparse_Representative(H).
2. Construct $SR(G)$ according to Definition 2.1 using the sparse representatives of the children of G.

END.

The following lemma analyzes the preprocessing algorithm.

Lemma 3.1 *Algorithm Pre_Outerplanar(G_{op}) runs in $O(n)$ time and uses $O(n)$ space.*

Proof: Step 1 needs $O(n)$ time and space by Lemma 2.1. Let $P(n)$ be the maximum time required by Step 2. Then, $P(n)$ satisfies the recurrence

$$P(n) \leq \max\{P(n_1) + P(n_2) \mid n_1 + n_2 = n + 2, \quad n_1, n_2 \leq 2n/3\} + O(1), \quad n > 1,$$

whose solution is $P(n) = O(n)$. Step 3 can be easily implemented in $O(n)$ time by performing a traversal of all leaf subgraphs of $ST(G_{op})$ and assigning to $A[v]$ the pointer to the first subgraph containing vertex v in this traversal. Similarly, during the same traversal, we can assign to $B[e]$ pointers pointing to the at most two (by Proposition 2.1) leaf subgraphs containing edge e. Step 4 also needs $O(n)$ time by [37]. The space required is proportional to the size of $ST(G_{op})$ since each sparse representative has $O(1)$ size. Therefore the space needed for the above data structures is $O(|ST(G_{op})|) = O(n)$. The bounds follow. ∎

3.2 The query algorithm

The main idea of the query algorithm for finding the distance between any two vertices v and z of G_{op} is as follows. First search $ST(G_{op})$ to find a descendant subgraph G of G_{op} such that the separation pair $p = (p_1, p_2)$ associated with G separates v from z. Then, for the distance $d(v, z)$ between v and z (in G_{op}) we have:

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

Hence, to find $d(v, z)$ it suffices to compute the distances $d(v, p_1)$, $d(p_1, z)$, $d(v, p_2)$ and $d(p_2, z)$. There are two problems regarding the computation of these distances. First, v and z may not belong to $SR(G)$ and hence the required distances between any of these two vertices and p_1, p_2 are not immediately available. Second, even if v and z belong to $SR(G)$, the required distances may not be the correct ones. By the preprocessing algorithm, the distance between any two vertices in $SR(G)$ represents the length of a shortest path that stays inside G. The actual shortest path, however, may leave G at one separation vertex and enter G through another. In the following, we shall show how to overcome these two problems by using the shortest path information encoded in the sparse representatives.

We start with the second problem. Assume that $G \neq G_{op}$. Let $M(G)$ be the set of separation pairs attached to G. Denote by $D(M(G))$ the set of all distances $d(x_1, x_2)$ and $d(x_2, x_1)$ in G_{op}, where (x_1, x_2) is any separation pair in $M(G)$. Let H and W be descendant subgraphs of G_{op}. Then, $D(M(H) \cap M(W))$ denotes the set of distances $d(x, y)$ and $d(y, x)$ in G_{op}, where (x, y) is a separation pair in $M(H) \cap M(W)$. To tackle the second problem it suffices to show how $D(M(G))$ can be efficiently computed.

Before showing this, we will first show how distances in G_{op} between any two vertices of $SR(G)$ can be computed in $O(1)$ time using $D(M(G))$ and the distances in $SR(G)$ computed by the preprocessing algorithm. By Proposition 2.1, we know that $|M(G)| \leq 4$ and hence $|D(M(G))| = O(1)$. Now, consider any two vertices u, v in $SR(G)$. Then, either $d(u, v) = d_G(u, v)$, or the shortest u-v path leaves G through separation vertex s_1 and enters G through separation vertex s_2. In the latter case, $d(u, v) = d_G(u, s_1) + d(s_1, s_2) + d_G(s_2, v)$. Observe that $d_G(u, s_1)$ and $d_G(s_2, v)$ are available from $SR(G)$ and $d(s_1, s_2)$ is available from $D(M(G))$. Since there are a constant number of separation pairs (s_1, s_2) to consider in $D(M(G))$, we can determine $d(u, v)$ in $O(1)$ time. Hence, in either case the whole computation takes $O(1)$ time.

We now return to the computation of $D(M(G))$. This is done by the following algorithm, which is based on the above idea for computing correct shortest path information between vertices in $SR(G)$.

ALGORITHM Attached_Pairs(G)
BEGIN

1. Let G' be the parent of G in $ST(G_{op})$. If $G' = G_{op}$, then $D(M(G')) := \emptyset$; otherwise, compute recursively $D(M(G'))$ by running Attached_Pairs(G').

2. Let (s'_1, s'_2) be the separation pair associated with G'. Find $d(s'_1, s'_2)$ and $d(s'_2, s'_1)$ in G_{op} by using $SR(G')$ and the information in $D(M(G'))$, as explained above. Set $D(M(G)) := D(M(G') \cap M(G)) \cup \{d(s'_1, s'_2), d(s'_2, s'_1)\}$.

END.

Lemma 3.2 *Algorithm Attached_Pairs(G) computes correctly the set $D(M(G))$ in $O(\log n)$ time.*

Proof: We proceed by induction on the depth of the recursion. The basis case (G being a child of G_{op}) is trivially satisfied. For the induction hypothesis, assume that we have correctly computed $D(M(G'))$, where G' is the parent of G in $ST(G_{op})$ and $G' \neq G_{op}$. By the preprocessing algorithm, $M(G)$ contains (s'_1, s'_2) (the separation pair associated with G') and a subset $M^*(G')$ of $M(G')$. $M^*(G')$ contains precisely those separation pairs from $M(G')$ whose endpoints belong to G. In other words, $M^*(G') = M(G') \cap M(G)$. Hence, $D(M(G)) = D(M(G') \cap M(G)) \cup \{d(s'_1, s'_2), d(s'_2, s'_1)\}$. The distances in $D(M(G') \cap M(G))$ are known by the induction hypothesis. From $SR(G')$ and $D(M(G'))$, the distances $d(s'_1, s'_2)$ and $d(s'_2, s'_1)$ can be computed in $O(1)$ time, as explained above. Hence, the correctness has been established. Concerning the time complexity, every recursion step takes $O(1)$ time and the depth of the recursion is at most the depth of $ST(G_{op})$. Consequently, algorithm Attached_Pairs runs in $O(\log n)$ time. ∎

We now turn to the first problem. Once we know that distances between separation pairs attached to G are correct, we can concentrate on G itself to find the four distances required by Eq. (1). However, these distances may not be directly available, since v and z may not belong to $SR(G)$. We will find these distances in a successive way. We shall use a recursive procedure that finds distances between pairs of vertices in G_{op}. The vertex pairs considered are separation pairs associated with descendant subgraphs of G.

For any two pairs p' and p'' of vertices, let $D(p', p'')$ denote the set of all four distances from a vertex in p' to a vertex in p''. Let v' (resp. z') be a vertex that belongs to the same descendant subgraph of G_{op} that is a leaf of $ST(G_{op})$ and that contains v (resp. z). Let p^v (resp. p^z) denote the pair of vertices (v, v') (resp. (z, z')). Then, Eq. (1) shows that any distance in $D(p^v, p^z)$ (and hence the requested $d(v, z)$) can be found in constant time, given $D(p^v, p)$ and $D(p, p^z)$, where p is the separation pair associated with the descendant subgraph G that separates v from z. We are now ready to give the entire query algorithm.

ALGORITHM Dist_Query_Outerplanar(G_{op}, v, z)
BEGIN

1. Using table A, find pairs of vertices p^v and p^z as defined above.

2. Find a descendant subgraph G of G_{op} such that the separation pair p associated with

G separates p^v and p^z in G. Note that G is the lowest common ancestor of the two leaf subgraphs containing p^v and p^z respectively.

3. Run algorithm Attached_Pairs(G).

4. Run Find_D($G, D(p^v, p)$).

5. Run Find_D($G, D(p, p^z)$).

6. Determine $D(p^v, p^z)$ by computing each distance using Eq. (1) and the distances computed in $D(p^v, p)$ and $D(p, p^z)$.

END.

Steps 4 and 5 of the above algorithm are based on the procedure Find_D($G, D(s, r)$) given below, which finds distances between pairs of vertices. The procedure takes as input a descendant subgraph G of G_{op}, a pair of vertices s belonging to G, and the separation pair r associated with G. It computes the set $D(s, r)$ of distances in G.

PROCEDURE Find_D($G, D(s, r)$)
BEGIN

1. Traverse $ST(G_{op})$ top-down from G to the leaf subgraph containing s to find a descendant subgraph G' of G such that the separation pair r' associated with G' separates s and r in G'.

2. If G' is a leaf of $ST(G_{op})$, then determine $D(s, r)$ directly in constant time.

3. If G' is not a leaf, then run Find_D($G', D(s, r')$). Compute each distance in $D(s, r)$ using Eq. (1) and the distances in $D(s, r')$ and $D(r', r)$. The latter set is available from $SR(G')$.

END.

Lemma 3.3 *Algorithm Dist_Query_Outerplanar(G_{op}, v, z) finds the distance between any two vertices v and z of an n-vertex outerplanar digraph G_{op} in $O(\log n)$ time.*

Proof: To establish the correctness of the algorithm it suffices, in view of Lemma 3.2 and our preceding discussion, to establish the correctness of the procedure Find_D. The latter follows easily by an induction argument similar to that used in the proof of Lemma 3.2, and the fact that, due to Step 3, the distances between vertices of the separation pairs attached to G are the correct ones (i.e., the distances in G_{op}).

Concerning the time complexity, it is clear that Steps 1 and 2 take $O(1)$ time (by the preprocessing algorithm). Step 3 takes $O(\log n)$ time by Lemma 3.2. Step 6 takes $O(1)$ time (as explained above). The time complexity of Steps 4 and 5 is that of procedure Find_D, which can be evaluated as follows.

Let $Q(\ell)$ be the maximum time necessary to compute $D(s,r)$, where ℓ is the level of G in $ST(G_{op})$. Let also ℓ_{max} be the depth of $ST(G_{op})$ and let $\ell < \ell' \leq \ell_{max}$ be the level of G'. Then, from the description of the algorithm

$$Q(\ell) \leq Q(\ell') + O(\ell' - \ell) \quad \text{for } \ell < \ell' \leq \ell_{max},$$

which gives $Q(\ell) = O(\log n)$. Thus, the total time needed by the algorithm is $O(\log n)$. ∎

Algorithm Dist_Query_Outerplanar can be modified to answer shortest path queries as well. 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 $ST(G_{op})$. To do this, we maintain some additional information during the preprocessing phase of our algorithm. More precisely, during the construction of $SR(G)$ (for some descendant subgraph G of G_{op}), we compute, for every e in $SR(G)$, a pointer $trail(e)$ pointing to the list of edges in $SR(G_1) \cup SR(G_2)$ that e represents, where G_1 and G_2 are the children of G in $ST(G_{op})$. If G is a leaf, then $trail(e)$ is e itself. Moreover, for the special case where G is not a leaf and e represents only a single edge e', $trail(e)$ is defined to be the same as $trail(e')$. This condition ensures that, for all non-leaf G, $trail(e)$ points to a list containing at least two elements. Since every $SR(G)$ is of $O(1)$ size, the computation of the $trail(\cdot)$ values can be done easily and within the resource bounds of Lemma 3.1.

Lemma 3.4 *The shortest path between any two vertices v and z of an n-vertex outerplanar digraph G_{op} can be found in $O(L + \log n)$ time, where L is the number of edges of the path.*

Proof: Since every $SR(G)$ is of constant size, it is trivial to extend procedure Find_D (and consequently algorithm Dist_Query_Outerplanar) to compute, except for distances in each $SR(G)$, the corresponding shortest paths. As a consequence, every shortest path returned by Dist_Query_Outerplanar consists of edges e belonging to sparse representatives of descendant subgraphs of G_{op}. Now, every such edge e can be regarded as a root of a tree defined by the $trail(\cdot)$ values, by considering the elements of $trail(e)$ as the children of e. The leaves of this tree, which we call the *trail tree* of e, are the edges of G_{op}. Clearly, the trail tree has no more than L leaves, where L is the number of the edges of the requested shortest path. Then, the time to answer a shortest path query is the time taken by Dist_Query_Outerplanar, which is $O(\log n)$, plus the time required to output the path.

The latter is proportional to the time required to traverse the trail tree, which is linear in L, since each non-leaf node of the trail tree has at least two children. ∎

3.3 The update algorithm

We now show how our data structures for answering shortest path and distance queries in G_{op}, can be updated in the case where an edge cost is modified. Note that updating after an edge deletion is equivalent to the updating of the cost of the particular edge with a very large value, such that this edge will not be used by any shortest path.

The update algorithm is based on the following idea: the edge will belong to at most $O(\log n)$ subgraphs of G_{op}, as they are determined by the Sep_Tree algorithm. Therefore, it suffices to update (in a bottom-up fashion) the sparse representatives of those subgraphs that are on the path from the subgraph G containing e (where G is a leaf of $ST(G_{op})$) to the root of $ST(G_{op})$. Recall that e can belong to at most two leaf subgraphs of $ST(G_{op})$.

Let $parent(G)$ denote the parent of a node G in $ST(G_{op})$, and $\hat{G}$ denote the sibling of a node G in a $ST(G_{op})$. Note that $G \cup \hat{G} = parent(G)$ and $SR(G) \cup SR(\hat{G}) \supset SR(parent(G))$. The algorithm for the update operation is the following.

ALGORITHM Update_Outerplanar$(G_{op}, e, w(e))$
BEGIN
1. Find a leaf G of $ST(G_{op})$ 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 $\hat{G}$ then update the cost of e in $\hat{G}$.
4. **While** $G \neq G_{op}$ **do**
    (a) Update $SR(parent(G))$ using the new versions of $SR(G)$ and $SR(\hat{G})$.
    (b) $G := parent(G)$.

END.

Lemma 3.5 *Algorithm Update_Outerplanar updates, after an edge cost modification or edge deletion, the data structures created by the preprocessing algorithm in $O(\log n)$ time.*

Proof: Using table B (computed during preprocessing), we can implement Step 1 in $O(1)$ time. Steps 2 and 3 also require $O(1)$ time. Each iteration of Step 4 takes $O(1)$ time, since it involves the computation of a sparse representative from the (updated) sparse representatives of its children (recall the discussion following Definition 2.1). By Lemma 2.1, the number of iterations in Step 4 is $O(\log n)$. Hence, Step 4 requires a total of $O(\log n)$ time. ∎

Note that if e belongs to a leaf G' which is not a sibling of G, then simply run the algorithm once more for this leaf subgraph. Clearly, this does not change the time bound of Lemma 3.5.

3.4 Handling of negative cycles and summary of the results

The initial digraph G_{op} can be tested for the existence of a negative cycle in $O(n)$ time by [29]. Assume now that G_{op} does not contain a negative cycle and that the cost $c(v, w)$ of an edge $\langle v, w \rangle$ in G_{op} has to be changed to $c'(v, w)$. We must check if this change does not create a negative cycle. We modify our algorithms in the following way. Before running algorithm Update_Outerplanar, run algorithm Dist_Query_Outerplanar to find the distance $d(w, v)$. If $d(w, v) + c'(v, w) < 0$, then stop and announce non-acceptance of this edge cost modification (because it obviously creates a negative cycle). Otherwise, continue with algorithm Update_Outerplanar. Clearly, the above procedures for testing the initial digraph and testing the acceptance of the edge cost modification do not affect the resource bounds of our preprocessing or of our update algorithm, respectively. Hence, we can now summarize the main result of Section 3.

Theorem 1 *Given an n-vertex outerplanar digraph G_{op} with real-valued edge costs but no negative cycles, there exists an algorithm for the dynamic shortest path problem in G_{op} with the following performance characteristics: (i) preprocessing time and space $O(n)$; (ii) distance query time $O(\log n)$; (iii) shortest path query time $O(L + \log n)$ (where L is the number of edges of the reported path); (iv) update time, after an edge cost modification or edge deletion, $O(\log n)$.*

Proof: Immediate from Lemmata 3.1, 3.3, 3.4, 3.5, and the preceding discussion. ∎

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 technique (recall Section 2) and on the algorithms of the previous section. Let q be the minimum cardinality of a hammock decomposition of G. Each update or query operation will be carried out in two levels: on the hammock level, where we consider the one or two hammocks containing the updated edge or query vertices, and on the level of the compressed planar digraph G_q (obtained as a result of compression of all hammocks). We propose two types of algorithms for planar digraphs depending on which algorithm we use for computing the shortest path information in G_q: the algorithm of [31] which leads to a dynamic algorithm with smaller update time, or the algorithms of

[4, 12] which leads to a class of dynamic algorithms with improved query times. Throughout this section let G be an n-vertex planar digraph with nonnegative real edge costs.

4.1 Update-efficient algorithm

In this version we do not construct (in the preprocessing phase) or maintain (in the update algorithm) special data structures associated with the compressed digraph G_q. In the query algorithm we just run on G_q the single-source shortest path algorithm from [31]. Details are given below.

The preprocessing algorithm for G is the following.

ALGORITHM Pre_Planar(G)
BEGIN

1. Find a hammock decomposition of G into q hammocks.
2. Run the algorithm Pre_Outerplanar(H) in each hammock H.
3. Compress each hammock H with respect to its attachment vertices as described in [19]. This results into a planar digraph G_q, which is of size $O(q)$.

END.

Lemma 4.1 *Algorithm Pre_Planar runs in $O(n)$ time and uses $O(n)$ space.*

Proof: Step 1 runs in $O(n)$ time [19]. Step 2 runs, overall hammocks, in $O(n)$ time by Theorem 1.

From the discussion in Section 2 and the fact that in Step 2 we compute $SR(H)$ for every hammock H, it is easy to see that Step 3 takes $O(1)$ time per hammock H, or $O(q)$ time in total. The bounds follow. ∎

The update algorithm is straightforward. Let e be the edge whose cost has been modified. There are two data structures that should be updated: the hammock H to which e belongs, and the graph G_q. The first data structure can be updated by the algorithm Update_Outerplanar. Note that this algorithm provides G_q with a new updated sparse representative of H, from which the compressed version of H (with respect to its attachment vertices) can be constructed using the method in [19]. As a result, we obtain also the updated digraph G_q. Therefore, we have the following lemma.

Lemma 4.2 *The data structures created by algorithm Pre_Planar can be updated in the case of an edge cost modification or edge deletion in $O(\log n)$ time.*

Proof: The correctness is obvious. By Lemma 3.5, Algorithm Update_Outerplanar updates H in $O(\log |H|)$ time. The compressed version of H can be computed in $O(1)$ time from $SR(H)$. Hence, the claimed bound follows. ∎

For the query algorithm we will make use of the recent linear-time algorithm for the single source shortest path problem given in [31]. This algorithm solves the problem in a directed digraph G in $O(|V(G)|)$ time, if G has an 2/3-separator of size $O(n^{1-\beta})$, for any $\beta > 0$, and if a corresponding division of G, called $\delta(n)$*-division* [18], into edge disjoint subgraphs of size $\delta(n) = o(n)$ can be constructed in linear time. Both assumptions are satisfied in the case of planar digraphs.

The query algorithm for finding the shortest path or distance between any two vertices v and z of G is similar to the ones given in [13, 20] which originate from ideas developed in [19]. The crucial observation is that if both v and z belong to the same hammock H, then their shortest path does not necessarily have to stay in H. In any case, all we have to do is a constant number of queries between vertices inside a hammock (using algorithm Dist_Query_Outerplanar) and a constant number of queries between attachment vertices in the graph G_q using the algorithm in [31]. Note that distances in G_q are the same as those in G.

ALGORITHM Query_Planar(G, v, z)
BEGIN
(* Let H, H' be hammocks with attachment vertices a_i, $1 \leq i \leq 4$ and a'_i, $1 \leq i \leq 4$, respectively, such that $v \in H$ and $z \in H'$. *)
if $H \equiv H'$ (* i.e., both v, z belong to H *) **then**

1. Compute $d_H(v,z), d_H(v,a_i), d_H(a_i,z)$ using algorithm Dist_Query_Outerplanar and $d_{G_q}(a_i,a_j)$ using the algorithm in [31], $\forall 1 \leq i,j \leq 4$.
2. $d_{ij}(v,z) = \min_{i,j}\{d_H(v,a_i) + d_{G_q}(a_i,a_j) + d_H(a_j,z)\}$.
3. $d(v,z) = \min\{d_H(v,z), d_{ij}(v,z)\}$.

else (* $H \neq H'$ *)

4. Compute $d_H(v,a_i), d_{H'}(a'_j,z)$ using algorithm Dist_Query_Outerplanar and $d_{G_q}(a_i,a'_j)$ using the algorithm in [31], $\forall 1 \leq i,j \leq 4$.
5. $d(v,z) = \min_{i,j}\{d_H(v,a_i) + d_{G_q}(a_i,a'_j) + d_{H'}(a'_j,z)\}$.

END.

It is very easy to extend the above algorithm to report the shortest path between v and z, since all calls to algorithm Dist_Query_Outerplanar and to algorithm in [31] can return the corresponding shortest paths as well.

Lemma 4.3 *Algorithm Query_Planar computes the distance (resp., shortest path) between any two vertices in a planar digraph in $O(\log n + q)$ (resp., $O(L + \log n + q)$) time, where L is the number of the edges of the reported path.*

Proof: The correctness of the algorithm can be easily verified (see also [13, 20]). Let us analyze the time complexity. We need $O(q)$ time for queries in G_q using the single source shortest path algorithm of [31] (for computing a distance or a compressed shortest path) and $O(\log |H|)$ or $O(L_H + \log |H|)$ time respectively for distance and shortest path queries in each hammock H (Theorem 1), where $|H|$ is the size of H and L_H is the portion (in number of edges) of the shortest path contained in H. This results in a total of $O(q + \log n)$ or $O(L + q + \log n)$ over all hammocks, where $L = \sum_H L_H$. ∎

The results in this subsection can be summarized in the following theorem.

Theorem 2 *Let G be an n-vertex planar digraph with nonnegative edge costs and let q be the minimum cardinality of a hammock decomposition of G. There exists an algorithm for the dynamic shortest path problem on G with the following performance characteristics: (i) preprocessing time and space $O(n)$; (ii) distance query time $O(\log n + q)$; (iii) shortest path query time $O(L + \log n + q)$ (where L is the number of edges of the reported path); (iv) update time, after an edge cost modification or edge deletion, $O(\log n)$.*

Proof: Follows immediately by Lemmata 4.1, 4.2, and 4.3. ∎

4.2 Query-efficient algorithms

We present now an algorithm whose preprocessing and query time may not be as good as those of Theorem 2, but whose query time is sublinear on q. The algorithm is based on the following result proved independently in [4, 12].

Theorem 3 *[4, 12] Any n-vertex planar digraph G can be preprocessed using $O(n^{3/2})$ time and space so that any single-pair distance query can be answered in $O(n^{1/2})$ time.*

We make the following simple modifications to the algorithms from the previous subsection. In algorithm Pre_Planar, we run as a (new) Step 4 the preprocessing phase of the algorithm from Theorem 3 in G_q. In the update algorithm, we recompute the data structure associated with G_q using the preprocessing part of the algorithm from Theorem 3 (in addition to updating the data structures associated with the hammocks). Finally, the query algorithm is the same, only we use Theorem 3 instead of the algorithm from [31] for computing distances between vertices of G_q. Hence, we have the following result.

Theorem 4 *Let G be an n-vertex planar digraph with nonnegative edge costs and let q be the minimum cardinality of a hammock decomposition of G. There exists an algorithm for the dynamic shortest path problem in G with the following performance characteristics: (i) preprocessing time and space $O(n + q^{3/2})$; (ii) distance query time $O(\log n + q^{1/2})$; (iii) shortest path query time $O(L + \log n + q^{1/2})$ (where L is the number of edges of the reported path); (iv) update time, after an edge cost modification or edge deletion, $O(\log n + q^{3/2})$.*

Proof: By Theorem 3, the new Step 4 in the preprocessing algorithm contributes an additive factor of $O(q^{3/2})$ to the preprocessing time and space given by Lemma 4.1. The same holds for the update algorithm. Substituting the calls to the algorithm in [31] by the query part of the algorithm in Theorem 3, contributes an additive factor of $O(q^{1/2})$ to the distance and shortest path query bounds given by Lemma 4.3. The claimed bounds follow. ∎

5 Further Results

In this section we give some further results following from our approach described in the previous sections. First, we show that with our data structures for outerplanar digraphs we can solve in $O(n)$ time the single-source shortest path problem in an n-vertex outerplanar digraph with real edge costs but no negative cycles, thus matching the results given in [19, 21] for the same problem. Note that none of these results is superseded by the linear-time algorithm for planar digraphs in [31], since that algorithm does not handle negative real edge costs. Then, we present an efficient parallel implementation of our algorithms on the CREW PRAM model of computation. Next, we discuss extensions of our results to digraphs of genus $\gamma = O(n^{1-\epsilon})$ for any $\epsilon > 0$. Finally, we discuss how the edge insertion and edge re-insertion operations can be supported.

5.1 Single-source shortest paths in outerplanar digraphs

We show here that our data structures of Section 3 can solve optimally the single-source shortest path problem in the case of outerplanar digraphs.

Let G_{op} be an n-vertex outerplanar digraph with real edge costs. Since G_{op} can be tested for a negative cycle in $O(n)$ time [30], we can assume w.l.o.g. that G_{op} does not have such a cycle. Preprocess G_{op} using algorithm Pre_Outerplanar (Section 3.1). Let $U \subset V$ be a subset of $O(1)$ vertices of G_{op} with a weight $d_0(u)$ on any $u \in U$. For any vertex v of G_{op}, define the weighted distance $d(U,v)$ as $d(U,v) = \min\{d_0(u) + d(u,v) | u \in U\}$. We assume that $d(U,v) = d_0(v)$ for every $v \in U$. The following algorithm computes $d(U,v)$, $\forall v \in G_{op}$.

ALGORITHM Single_Source_Query_Outerplanar(G_{op}, U)
BEGIN

1. Let S be the 2/3-separator associated with the root G_{op} of $ST(G_{op})$. Compute $d(u,s)$ for all vertices $u \in U$ and $s \in S$ by using algorithm Dist_Query_Outerplanar.

2. For any $s \in S$ define $d_0(s) = \min\{d(u,s) | u \in U\} = d(U,s)$.

3. Run recursively Single_Source_Query_Outerplanar($G, (S \cup U) \cap G$), on each child subgraph G of G_{op} which is not a leaf of $ST(G_{op})$. If G is a leaf, then distances are computed easily since the associated subgraph is of $O(1)$ size.

END.

The correctness of the algorithm follows from its description. Let $D(n)$ be the running time of the algorithm. Then, $D(n) \leq \max\{D(n_1) + D(n_2) | n_1 + n_2 = n + 2,\ n_1, n_2 \leq 2n/3\} + O(|S| \cdot |U| \cdot \log n) = \max\{D(n_1) + D(n_2) | n_1 + n_2 = n,\ n_1, n_2 \leq 2n/3\} + O(\log n)$, which gives $D(n) = O(n)$.

Let v be any vertex of G_{op}. To compute the distances $d(v,u)$ from v to every other vertex u in G_{op}, run Single_Source_Query_Outerplanar($G_{op}, \{v\}$) with $d_0(v) = 0$. To compute the shortest paths from v to every other vertex (which, as it is well-known, form a tree rooted at v) do the following: Perform a breadth-first search starting at v that is based on the computed distances. The children of a vertex u which already belongs to T are those adjacent vertices z of u that satisfy $d(v,z) = d(v,u) + c(u,z)$, where $c(u,z)$ is the cost of the edge $\langle u, z \rangle$.

Hence, the single-source shortest path problem in G_{op} can be solved in $O(n)$ time.

5.2 Parallel Implementation

We start with the case of outerplanar digraphs. All we have to show is how the methods from Section 3 can be efficiently parallelized. This is shown in the following theorem.

Theorem 5 *Given an n-vertex outerplanar digraph G_{op} with real edge costs but no negative cycles, there exists a CREW PRAM algorithm for the dynamic shortest path problem in G_{op} with the following performance characteristics: (i) preprocessing time $O(\log n)$ with $O(n \log n)$ work and space $O(n)$; (ii) distance query time $O(\log n)$ using a single processor; (iii) shortest path query time $O(\log n)$ using $O(L + \log n)$ work (where L is the number of edges of the path); (iv) update time $O(\log n)$ using a single processor, after an edge cost modification or edge deletion.*

Proof: We will first show how the preprocessing algorithm from Section 3.1 can be implemented in parallel. Step 1 can be implemented in $O(\log n)$ time and $O(n \log n)$ work as follows. Triangulate each face of G_{op} and construct the dual graph T of the resulting triangulation G_T, excluding the outer face of G_{op}. Since G_{op} is outerplanar, then T is a tree. Assign each vertex x of G_{op} to a unique triangle of G_T incident on x and determine for each triangle t the number of vertices of G_{op} assigned to t. Call this number the *weight of* t and also the *weight of the vertex of* T that corresponds to t. Then, compute for each node v of T the number $w(v)$ equal to the sum of the weights of all descendants of v (including v itself). This can be easily done in $O(\log n)$ time and $O(n)$ work (see e.g., [28, Chapter 3]). Using the numbers $w(v)$, find in constant time and $O(n)$ work an edge e of T whose removal divides T into two subtrees T_1 and T_2 each of total weight on its vertices at most 2/3 of the total weight of T. Then, the pair of the endpoints of the dual edge of e in G_T will be a 2/3-separator of G_{op}. Moreover, updating the numbers $w(\cdot)$ for T_1 and T_2 requires $O(1)$ time and $O(n)$ work. Thus Step 1 requires $O(\log n)$ time and $O(n \log n)$ work. The total work required by Step 2 is described by the recurrence for $P(n)$ in the proof of Lemma 3.1. The parallel time of Step 2 satisfies the recurrence $T_p(n) = T_p(n/2) + O(1)$, whose solution is $T_p(n) = O(\log n)$. Step 3 can be easily implemented in $O(\log n)$ time and $O(n)$ work. The same bounds hold for Step 4 by [37]. Hence, the data structures of the preprocessing phase can be constructed in $O(\log n)$ time and $O(n \log n)$ work.

The sequential distance query and the update algorithms for outerplanar digraphs take $O(\log n)$ time using a single processor, and hence need no special treatment.

The shortest path sequential query algorithm, however, requires $O(L + \log n)$ time, where L is the number of edges of the path and in the worst-case L may be $\Omega(n)$. We can find an optimal logarithmic-time parallel implementation of the shortest path query algorithm using the trail information (recall the discussion at the end of Section 3.2). All that is required is to perform a breadth-first traversal of the appropriate trail tree which has at most L leaves and size $O(L)$. Using standard parallel techniques (see e.g., [28]), we can perform this traversal (and thus output the path) in $O(\log n)$ time and $O(L)$ work. This completes the proof of the theorem. ∎

In the case of planar digraphs we need a parallel algorithm to build the data structures in G_q (recall Section 4). We will make use of the following result [9]: In any n-vertex planar digraph the single-source shortest path problem can be solved in $O(\log^4 n)$ time with $O(n^{3/2})$ work on a CREW PRAM. Furthermore, finding a hammock decomposition (Step 1 of algorithm Pre_Planar) takes $O(\log n \log^* n)$ time and $O(n \log n \log^* n)$ work by [34]. Combining these results with Theorem 5 and using the construction from Section 4 we have the following.

Theorem 6 *Given an n-vertex planar digraph G with real edge costs but no negative cycles, there exists a CREW PRAM algorithm for the dynamic shortest path problem in G with the following performance characteristics: (i) preprocessing time $O(\log n \log^* n)$ with $O(n \log n \log^* n)$ work and $O(n)$ space; (ii) distance query time $O(\log n + \log^4 q)$ with $O(\log n + q^{3/2})$ work; (iii) shortest path query time $O(\log n + \log^4 q)$ with $O(L + \log n + q^{3/2})$ work (where L is the number of edges of the reported path); and (iv) update time $O(\log n)$ using a single processor, after an edge cost modification or edge deletion.*

Proof: We start with the preprocessing algorithm. Step 1 takes $O(\log n \log^* n)$ time and $O(n \log n \log^* n)$ work [34]. Step 2 takes $O(\log n)$ time and $O(n \log n)$ work by Theorem 5. The compression, in parallel, of each hammock with respect to its attachment vertices is also described in [34] and can be done within the resource bounds required by Step 1. The space is $O(n)$ by Theorem 5 and the results in [34]. Hence, the preprocessing algorithm takes $O(\log n \log^* n)$ time with $O(n \log n \log^* n)$ work, and requires $O(n)$ space.

The distance or shortest path query algorithm involves a constant number of queries in at most 2 hammocks and a constant number of queries in G_q. Each of the former queries takes $O(\log n)$ time with $O(\log n)$ or $O(L + \log n)$ work, respectively, by Theorem 5. Each of the latter queries needs $O(\log^4 q)$ time with $O(q^{3/2})$ work by [9]. Hence, a distance (resp. shortest path) query takes $O(\log n + \log^4 q)$ time with $O(\log n + q^{3/2})$ (resp. $O(L + \log n + q^{3/2})$) work.

The update algorithm takes $O(\log n)$ time using a single processor and hence needs no special treatment. This completes the proof of the theorem. ∎

5.3 Extensions to digraphs of small genus

The hammock decomposition technique can be extended to n-vertex digraphs G of genus $\gamma = O(n^{1-\varepsilon})$ for any $\varepsilon > 0$. We make use of the fact [20] that, in this case, the minimum number q of hammocks is at most a constant factor times $\gamma + q'$, where q' is the minimum number of faces among all cellular embeddings of G on a surface of genus γ that cover all vertices of G. Note that the method in [20] does not require such an embedding to be provided by the input in order to produce the hammock decomposition. The decomposition can be found in linear time [20]. The only other property of planar graphs that is relevant to our shortest path algorithms (as well as to the algorithm from [31]) is the existence of a 2/3-separator of size $O(n^{1-\beta})$ for any n-vertex graph G, where $\beta > 0$, and that a $\delta(n)$-division for G can be constructed in linear time for any $\delta(n) = o(n)$.

For any n-vertex digraph G of genus $\gamma > 0$, a 2/3-separator of size $O(\sqrt{\gamma n})$ exists. Such a separator can be found in linear time and an embedding of the graph does not need to be provided by the input [10, 11]. If such an embedding *is* provided as an input, then one can find also an $o(n)$-division for G in $O(n)$ time [3]; otherwise, this division is computed in $O(n \log n)$ time. Thus, we have the following two types of results for the class of digraphs of genus $\gamma = O(n^{1-\varepsilon})$, where $\varepsilon > 0$.

If an embedding of G on a surface of genus γ is given as an input, then the statement of Theorem 2 as well as its extensions discussed in this section, hold for G. If an embedding of G is not provided by the input, then we have an additive $O(q \log q)$ factor in the preprocessing bounds of Theorem 2 and its extensions.

5.4 Handling edge insertions and re-insertions

Our algorithms do not directly support the operation edge insertion (i.e., they are partially dynamic). However, edge insertions can be handled in the following way. If a new edge e has to be inserted in an outerplanar digraph G_{op}, then one can just find an outerplanar embedding of the graph $G_{op} + e$ and run the preprocessing algorithm Pre_Outerplanar on it. This takes $O(n)$ time in total. In a similar way one can handle edge insertions in planar digraphs. Thus, all algorithms for outerplanar and planar digraphs discussed in this paper can be used as fully dynamic algorithms with preprocessing, query, edge deletion, and edge cost modification times the same as of the original algorithms, and edge insertion time equal to the preprocessing time. Hence, a fully dynamic algorithm derived from the above idea and Theorem 2 compares favorably with the fully dynamic solution given in [31].

The algorithms from this paper can support the operation edge *re-insertion*, which requires the insertion of an edge that has been deleted by a previous *delete* operation, within the bounds required for an update operation. This follows from the observation that deletion of an edge e is equivalent to changing the cost of e to a very large value (e.g., twice the sum of the absolute values of all edge costs) such that e will not be used by any shortest path. Now, re-insertion of (a previously deleted edge) e with new cost C is equivalent to decreasing the cost of e to C.

Acknowledgement. The authors are indebted to an anonymous referee for several helpful comments and suggestions that improved considerably the presentation of the paper.

References

[1] R. Agrawal, A. Borgida, and H.V. Jagadish, Efficient management of transitive relationships in large data and knowledge bases, in *Proc. ACM-SIGMOD Int'l Conf. on Management of Data*, 1989.

[2] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[3] L. Aleksandrov and H. Djidjev, Linear Algorithms for Partitioning Embedded Graphs of Bounded Genus, *SIAM Journal on Discrete Mathematics*, **9** (1996), 129–150.

[4] S. Arikati, D.Z. Chen, L.P. Chew, G. Das, M. Smid, and C. Zaroliagis, Planar Spanners and Approximate Shortest Path Queries among Obstacles in the Plane, in *Algorithms* (Proc. ESA'96), *Lecture Notes in Computer Science* **1136** (Springer-Verlag, 1996), pp. 514–528.

[5] G. Ausiello, G.F. Italiano, A.M. Spaccamela, U. Nanni, Incremental algorithms for minimal length paths, *Journal of Algorithms*, **12** (1991), 615-638.

[6] M. Carroll and B. Ryder, Incremental Data Flow Analysis via Dominator and Attribute Grammars, in *Proc. 15th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages*, 1988.

[7] S. Chaudhuri and C. Zaroliagis, Shortest Paths in Digraphs of Small Treewidth. Part I: Sequential Algorithms, *Algorithmica*, in print.

[8] B. Chazelle, A Theorem on Polygon Cutting with Applications, in *Proc. 23rd IEEE Symposium on Foundations of Computer Science*, 1982, pp.339-349.

[9] E. Cohen, Efficient Parallel Shortest-paths in Digraphs with a Separator Decomposition, *Journal of Algorithms*, **21** (1996), 331–357.

[10] H. Djidjev, A Separator Theorem for Graphs of Fixed Genus, *SERDICA*, **11** (1985), 319-329.

[11] H. Djidjev, A Linear Algorithm for Partitioning Graphs of Fixed Genus, *SERDICA*, **11** (1985), 369-387.

[12] H. Djidjev, On-Line Algorithms for Shortest Path Problems on Planar Digraphs, in *Graph-Theoretic Concepts in Computer Science* (Proc. WG'96), *Lecture Notes in Computer Science*, **1335** (Springer-Verlag, 1997), pp. 151–165.

[13] H. Djidjev, G. Pantziou and C. Zaroliagis, Computing Shortest Paths and Distances in Planar Graphs, in *Automata, Languages and Programming* (Proc. ICALP'91), *Lecture Notes in Computer Science*, **510** (Springer-Verlag, 1991), pp. 327–339.

[14] H. Djidjev, G. Pantziou and C. Zaroliagis, On-line and Dynamic Algorithms for Shortest Path Problems, in *Theoretical Aspects of Computer Science* (Proc. STACS'95), *Lecture Notes in Computer Science*, **900** (Springer-Verlag, 1995), pp. 193–204.

[15] S. Even and H. Gazit, Updating distances in dynamic graphs, *Methods of Operations Research*, **49** (1985), 371-387.

[16] P. Erdős and J. Spencer, *Probabilistic Methods in Combinatorics*, Academic Press, 1974.

[17] E. Feuerstein and A. Marchetti-Spaccamela, Dynamic Algorithms for Shortest Paths in Planar Graphs, *Theor. Computer Science*, **116** (1993), 359-371.

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

[19] G.N. Frederickson, Planar Graph Decomposition and All Pairs Shortest Paths, *Journal of the ACM*, **38:1** (1991), 162-204.

[20] G.N. Frederickson, Using Cellular Graph Embeddings in Solving All Pairs Shortest Path Problems, *Journal of Algorithms*, **19** (1995), 45-85.

[21] G.N. Frederickson, Searching among Intervals and Compact Routing Tables, *Algorithmica*, **15** (1996), 448–466.

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

[23] M. Fredman and R. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, *Journal of the ACM*, **34** (1987), 596-615.

[24] M.T. Goodrich and R. Tamassia, Dynamic Trees and Dynamic Point Location, in *Proc. 23rd ACM Symp. on Theory of Computing*, 1991, pp. 523–533.

[25] F. Harary, *Graph Theory*, Addison-Wesley, 1969.

[26] R. Hassin, Maximum flow in (s,t)-planar networks, *Inform. Process. Lett.*, **13** (1981), 107.

[27] N. Horspool, Incremental Generation of LR Parsers, Tech. Report, Dept of Computer Science, University of Victoria, 1988.

[28] J. JáJá, *An Introduction to Parallel Algorithms*, Addison-Wesley, 1992.

[29] D. Kavvadias, G. Pantziou, P. Spirakis and C. Zaroliagis, Hammock-on-Ears Decomposition: A Technique for the Efficient Parallel Solution of Shortest Paths and Other Problems, *Theoretical Computer Science*, **168:1** (1996), 121–154.

[30] D. Kavvadias, G. Pantziou, P. Spirakis and C. Zaroliagis, "Efficient Sequential and Parallel Algorithms for the Negative Cycle Problem", in *Algorithms and Computation* (Proc. ISAAC'94), *Lecture Notes in Computer Science*, **834** (Springer-Verlag, 1994), pp. 270–278.

[31] P. Klein, S. Rao, M. Rauch and S. Subramanian, Faster shortest-path algorithms for planar graphs, *Journal of Computer and System Sciences*, **5(1)** (1997), 3–23.

[32] A. Lingas, Efficient Parallel Algorithms for Path Problems in Planar Directed Graphs, in Proc. SIGAL'90, *Lecture Notes in Computer Science*, **450** (Springer-Verlag, 1990), pp. 447–457.

[33] G. Miller and J. Naor, Flows in planar graphs with multiple sources and sinks, in *Proc. 30th IEEE Symp. on Foundations of Computer Science*, 1989, pp.112-117.

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

[35] G. Ramalingan and T. Reps, On the Computational Complexity of Incremental Algorithms, Technical Report, University of Wisconsin-Madison, 1991.

[36] H. Rohnert, A dynamization of the all pairs least cost path problem, in *Theoretical Aspects of Computer Science* (Proc. STACS'85), *Lecture Notes in Computer Science*, **182** (Springer-Verlag, 1985), pp. 279–286.

[37] B. Schieber and U. Vishkin, On Finding Lowest Common Ancestors: Simplification and Parallelization, *SIAM Journal on Computing*, **17:6** (1988), 1253–1262.

[38] D. Sleator and R. Tarjan, A Data Structure for Dynamic Trees, *Journal of Computer and System Sciences*, **26** (1983), 362–391.

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

[40] D. Yellin and R. Strom, "INC: A language for incremental computations", *ACM Trans. Prog. Lang. Systems*, **13:2** (1991), 211–236.