Shortest Path Queries in Digraphs of Small Treewidth ⋆

Shiva Chaudhuri and Christos D. Zaroliagis

Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany
E-mail: {shiva, zaro}@mpi-sb.mpg.de

Abstract. We consider the problem of preprocessing an n-vertex digraph with real edge weights so that subsequent queries for the shortest path or distance between any two vertices can be efficiently answered. We give algorithms that depend on the *treewidth* of the input graph. When the treewidth is a constant, our algorithms can answer distance queries in $O(\alpha(n))$ time after $O(n)$ preprocessing. This improves upon previously known results for the same problem. We also give a dynamic algorithm which, after a change in an edge weight, updates the data structure in time $O(n^{\beta})$, for any constant $0 < \beta < 1$. The above two algorithms are based on an algorithm of independent interest: computing a shortest path tree, or finding a negative cycle in linear time.

1 Introduction

Finding shortest paths in digraphs is a well-studied and important problem with many applications, especially in network optimization (see e.g. [1]). The problem is to find paths of minimum weight between vertices in an n-vertex, m-edge digraph with real edge weights (Section 2). In the single-source problem we seek such paths from a specific vertex to all other vertices and in the all-pairs shortest paths (apsp) problem we seek such paths between every pair [1].

For general digraphs the best algorithm for the apsp problem takes $O(nm + n^2 \log n)$ time [14]. An apsp algorithm must output paths between $\Omega(n^2)$ vertex pairs and thus requires this much time and space. A more efficient approach is to preprocess the digraph so that subsequently, *queries* can be efficiently answered. A query specifies two vertices and a *shortest path query* asks for a minimum weight path between them, while a *distance query* only asks for the weight of such a path. This approach is particularly promising when the digraph is sparse i.e. $m = O(n)$. An interesting subclass of sparse digraphs, namely *outerplanar* digraphs, has been intensively studied. In [12] it was shown that after $O(n)$ preprocessing, a shortest path or distance query is answered in $O(L+\log n)$ time (where L is the number of edges of the reported path). In [8], a different approach reduces the distance query time to $O(\log n)$ (with the same preprocessing time). Recently, in [13], the distance query time is improved to $O(\alpha(n))$, where $\alpha(n)$

⋆ To appear in *Proc. 22nd ICALP'95.* This work was partially supported by the EU ESPRIT Basic Research Action No. 7141 (ALCOM II).

is the inverse of (the well-known) Ackermann's function and is a very slowly growing function.

Another important subclass of sparse graphs is the class of graphs with bounded treewidth. The study of graphs using the *treewidth* as a parameter was pioneered by Robertson and Seymour [17] and continued by many others (see e.g. [3–5]). Roughly speaking, the treewidth of a graph G is a parameter which measures how close is the structure of G to a tree. (A formal definition is given in Section 2.) Graphs of treewidth t are also known as partial t-trees and have at most tn edges. In [6], the same bounds as in [8] are achieved for the above problem on digraphs with treewidth at most 2. Classifying graphs based on treewidth is useful because diverse properties of graphs can be captured by a single parameter. For instance, the class of graphs of bounded treewidth includes series-parallel graphs, outerplanar graphs, graphs with bounded bandwidth and graphs with bounded cutwidth [3, 5]. Thus giving efficient algorithms parameterized by treewidth is an important step in the development of better algorithms for sparse graphs.

In this paper we consider the above problem for digraphs of small treewidth. Our main result is an algorithm that, for digraphs of constant treewidth, after $O(n)$ preprocessing answers a distance query in $O(\alpha(n))$ time and a shortest path query in $O(L\alpha(n))$ time. This improves the results in [6, 8, 12, 13] in two ways: it improves the distance query time and applies to a larger class of graphs. The data structures in [12, 13] are not dynamic, while those in [6, 8] are dynamic. After a change in the weight of an edge, these data structures can be updated in $O(\log n)$ time. We also give a dynamic data structure that does not achieve this update bound, but does achieve a sublinear one. In particular, we can perform updates in $O(n^\beta)$ time, for any constant $0 < \beta < 1$, maintaining the previous query times.

We actually show a trade-off between the preprocessing and query times which is parameterized by the treewidth of the graph and an integer $1 \leq k \leq \alpha(n)$. Specifically, for a digraph of treewidth t and any integer $1 \leq k \leq \alpha(n)$, we give an algorithm that achieves distance (resp. shortest path) query time $O(t^4k)$ (resp. $O(t^4kL)$). The preprocessing bound required is $O(t^4n \log n)$, when $k = 1$, $O(t^4n \log^* n)$, when $k = 2$, and decreases rapidly to $O(t^4n)$ when $k = \alpha(n)$ (Section 4). We note that graphs of treewidth t may have $\Omega(tn)$ edges.

Concerning the single-source problem, most algorithms either construct a *shortest path tree* rooted at a given vertex, or find a negative weight cycle. Constructing a shortest path tree is often easier when the digraph has non-negative edge weights. For general digraphs with non-negative real edge weights the best algorithm takes $O(m + n \log n)$ time [14] to construct the shortest path tree. If the digraph contains negative real edge weights, then one needs $O(nm)$ time to either construct a shortest path tree, or find a negative weight cycle [1]. For outerplanar digraphs, in $O(n)$ time, a shortest path tree can be constructed [8, 11], or a negative cycle can be found [15]. For *planar* digraphs with positive real edge weights, an $O(n)$ time algorithm is given in [16]. With negative but integer weights, the same paper gives an $O(n^{4/3} \log n)$ time algorithm which constructs a shortest path tree, or finds a negative cycle. In the case of negative real edge weights, the results for planar digraphs in [9, 10], imply an algorithm that in $O(n\sqrt{\log\log n})$ time either computes a shortest path tree, or decides that the graph contains a negative cycle. (We note that this algorithm does not find the cycle.) The best algorithm to construct a shortest path tree, or find a negative cycle in a planar digraph takes (in the worst case) $O(n^{1.5}\log n)$ time [15].

We also give here an $O(n)$ time algorithm that, for digraphs of constant treewidth, either constructs a shortest path tree or finds a negative cycle (Section 3). This generalizes the results in [15] for outerplanar digraphs. To the best of our knowledge, this is the most general class of graphs for which the complexity of computing a shortest path tree matches that of finding a negative cycle.

All of our algorithms start by computing a *tree-decomposition* of the input digraph G. The tree decomposition of a graph with constant treewidth can be computed in $O(n)$ time [4]. The main idea behind our algorithms is the following. We define a certain value for each node of the tree-decomposition of G, and an associative operator on these values. We then show that the shortest path problem reduces to computing the product of these values along paths in the tree-decomposition. Algorithms to compute the product of node values along paths in a tree are given in [2, 7]. Our preprocessing vs. query time trade-off arises from a similar trade-off in [2, 7]. The dynamization of our data structures is based on the above ideas and on a graph partitioning result which is of independent interest. Due to space limitations some proofs have been shortened or omitted.

2 Preliminaries

In this paper, we will be concerned with finding shortest paths or distances between vertices of a directed graph. Thus, we assume that we are given an *n-vertex weighted digraph* G, i.e. a digraph $G = (V(G), E(G))$ and a weight function $wt : E(G) \longrightarrow I\!R$. We call $wt(u, v)$ the *weight* of edge $\langle u, v\rangle$. The weight of a path in G is the sum of the weights of the edges on the path. For $u, v \in V(G)$, a *shortest path* in G from u to v is a path whose weight is minimum among all paths from u to v. The *distance* from u to v, written as $\delta(u, v)$ or $\delta_G(u, v)$, is the weight of a shortest path from u to v in G. A cycle in G is a (simple) path starting and ending at the same vertex. If the weight of a cycle in G is less than zero, then we will say that G contains a *negative cycle*. It is well-known [1] that shortest paths exist in G, iff G does not contain a negative cycle.

For a subgraph H of G, and vertices $x, y \in V(H)$, we shall denote by $\delta_H(x, y)$ the distance of a shortest path from x to y in H. A *shortest path tree* rooted at $v \in V(G)$, is a tree such that $\forall w \in V(G)$, the tree path from v to w is a shortest path in G from v to w.

Let G be a (directed or undirected) graph and let $W \subseteq V(G)$. Then by $G[W]$ we shall denote the subgraph of G induced on W. Let V_1, V_2 and S be disjoint subsets of $V(G)$. We say that S is a *separator for V_1 and V_2*, or that S *separates V_1 from V_2*, iff every path from a vertex in V_1 (resp. V_2) to a vertex in V_2 (resp. V_1) passes through a vertex in S. Let H be a subgraph of G. A *cut-set* *for* H is a set of vertices $C(H) \subseteq V(H)$, whose removal separates H from the rest of the graph.

Definition 1. *Let H be a digraph, with V_1, V_2 and U a partition of $V(H)$ such that U is a separator for V_1 and V_2. Let H_1 and H_2 be subgraphs of H such that $V(H_1) = V_1 \cup U$, $V(H_2) = V_2 \cup U$ and $E(H_1) \cup E(H_2) = E(H)$. We say that H_1' is a graph obtained by* absorbing H_2 into H_1, *if H_1' is obtained from H_1 by adding edges $\langle u, v\rangle$, with weight $\delta_{H_2}(u, v)$ or $\delta_H(u, v)$, for each pair $u, v \in U$. (In case of multiple edges, retain the one with minimum weight.)*

Absorbing preserves distances in a digraph, as the following lemma shows. This allows us to absorb the subgraph on one side of the separator and restrict our attention to the remaining subgraph, which maybe is smaller.

Lemma 1. *Let H, H_1, H_2 and H_1' be as in Definition 1. Then, for all $x, y \in V(H_1')$, $\delta_{H_1'}(x, y) = \delta_H(x, y)$.*

Proof. It is enough to show that $\delta_H(x, y) \leq \delta_{H_1'}(x, y)$ and $\delta_{H_1'}(x, y) \leq \delta_H(x, y)$. Call an edge $\langle u, v\rangle$ in H_1' an H_2-edge if it has weight $\delta_{H_2}(u, v)$ and an H-edge if its weight is $\delta_H(u, v)$.
Case 1: $\delta_H(x, y) \leq \delta_{H_1'}(x, y)$. Consider a shortest path from x to y in H_1'. Construct a walk from x to y in H by replacing, in the above path from H_1', all H_2-edges by a path in H_2 of the same weight and all H-edges by a path in H of the same weight (both of which exist, by construction). Now this walk has weight $\delta_{H_1'}(x, y)$ and a shortest path in H_1' from x to y cannot weight more.
Case 2: $\delta_{H_1'}(x, y) \leq \delta_H(x, y)$. Consider a shortest path from x to y in H. Find all maximal (w.r.t. the number of edges) subpaths that are contained in H_2. These paths must start and end in vertices in U. Let W be the weight of one such path (in H_2) from u to v, $u, v \in U$. Then H_1' has an edge $\langle u, v\rangle$ with weight either $\delta_{H_2}(u, v)$ or $\delta_H(u, v)$, both of which are at most W. Construct a path from x to y in H_1' by replacing each such subpath by the corresponding H_2-edge or H-edge in H_1'. The resulting path has weight at most $\delta_H(x, y)$. □

A *tree-decomposition* of a (directed or undirected) graph G is a pair (X, T) where $T = (V(T), E(T))$ is a tree and X is a family $\{X_i | i \in V(T)\}$ of subsets of $V(G)$, such that $\cup_{i \in V(T)} X_i = V(G)$ and also the following conditions hold:

- (*edge mapping*) $\forall (v, w) \in E(G)$, there exists an $i \in V(T)$ with $v \in X_i$ and $w \in X_i$.
- (*continuity*) $\forall i, j, k \in V(T)$, if j lies on the path from i to k in T, then $X_i \cap X_k \subseteq X_j$, or equivalently: $\forall v \in V(G)$, the nodes $\{i \in V(T) | v \in X_i\}$ induce a connected subtree of T.

The *treewidth* of a tree-decomposition is $\max_{i \in V(T)} |X_i| - 1$. The treewidth of G is the minimum treewidth over all possible tree-decompositions of G.

Fact 1 *[4] (a) For all constant $t \in I\!N$, there exists an $O(n)$ time algorithm which tests whether a given n-vertex graph G has treewidth $\leq t$ and if so, outputs a tree-decomposition (X, T) of G with treewidth $\leq t$, where $|V(T)| = n - t$.*

(b) We can, in $O(n)$ time, convert (X,T) into another tree-decomposition (X_b,T_b) of G with treewidth t, where T_b is a binary tree and $|V(T_b)| \leq 2(n-t)$.

Part (b) of the above fact follows by the usual binarization of an arbitrary tree. We will use this in Section 5. Given a tree-decomposition of G, we can quickly find separators in G, as the following proposition shows.

Proposition 1. *[17] Let G be a graph and let (X,T) be its tree-decomposition. Also let $e=(i,j) \in E(T)$ and let T_1 and T_2 be the two subtrees obtained by removing e from T. Then $X_i \cap X_j$ separates $\cup_{m \in V(T_1)} X_m$ from $\cup_{m \in V(T_2)} X_m$.*

3 Constructing a shortest path tree

Call a tuple (a,b,c) a *distance tuple* if a,b are arbitrary symbols and $c \in \mathbb{R}$. For two distance tuples, $(a_1,b_1,c_1),(a_2,b_2,c_2)$, define their product $(a_1,b_1,c_1) \otimes (a_2,b_2,c_2) = (a_1,b_2,c_1+c_2)$ if $b_1 = a_2$ and as nonexistent otherwise.

For a set of distance tuples, M, define minmap(M) to be the set $\{(a,b,c) : (a,b,c) \in M$ and $\forall (a',b',c') \in M$ if $a'=a, b'=b$, then $c \leq c'\}$, i.e. among all tuples with the same first and second components, minmap retains only the tuples with the smallest third component.

Let M_1 and M_2 be sets of distance tuples. Define the operator $\circ$ by $M_1 \circ M_2 =$ minmap(M), where $M = \{x \otimes y : \ x \in M_1, \ y \in M_2\}$. It is not difficult to show that $\circ$ is an associative operator.

Let G be a digraph with real edge weights. Note that in the above definition, if M_1 and M_2 have tuples of the form (a,b,x) where $a,b \in V(G)$ and x is the weight of a path from a to b, then $M_1 \circ M_2$ computes tuples (a,b,y) where y is the (shortest) distance from a to b using only the paths represented in M_1 and M_2.

For $X,Y \subseteq V(G)$, not necessarily distinct, define $P(X,Y) = \{(a,b,\delta_G(a,b)) : a \in X, b \in Y\}$. We will write $S(X)$ for $P(X,X)$. (By definition, $S(X)$ contains tuples $(x,x,0), \ \forall x \in X$.)

Definition 2. *Let G be an n-vertex weighted digraph without negative cycles and let (X,T) be a tree decomposition of G, with treewidth t. Then, for $i \in V(T)$, we define $\gamma(i) = S(X_i)$.*

The following lemma shows that we can compute $\delta(a,b)$ by computing the product of the γ values on the path in T between nodes i and j such that $a \in X_i$ and $b \in X_j$.

Lemma 2. *Let G, (X,T) and $\gamma(i)$, for $i \in V(T)$, be as in Definition 2. Let $v_1,\ldots,v_p$ be a path in T. Then $\gamma(v_1) \circ \ldots \circ \gamma(v_p) = P(X_{v_1}, X_{v_p})$.*

Proof. It is not hard to show, from the definitions of $P(X,Y)$ and of $\circ$, that $P(X,Y) \circ P(Z,W) = \{(x,w,d) : \ x \in X, w \in W, \ d$ is the weight of the shortest x to w path that includes a vertex in $Y \cap Z$ (this vertex may be x or w)$\}$.

We prove the lemma by induction on p. If $p = 1$, then the lemma holds by the definition of $\gamma(v_1)$. If $p > 1$, then by the inductive hypothesis, $\gamma(v_1) \circ \ldots \circ \gamma(v_{p-1}) = P(X_{v_1}, X_{v_{p-1}})$. By definition, $\gamma(v_p) = S(X_{v_p})$. By Proposition 1, *all* paths from a vertex in X_{v_1} to a vertex in X_{v_p} include a vertex from $X_{v_{p-1}} \cap X_{v_p}$. Hence, by the characterization above, $P(X_{v_1}, X_{v_{p-1}}) \circ S(X_{v_p}) = \{(x, y, \delta_G(x, y)) : x \in X_{v_1}, y \in X_{v_p}\} = P(X_{v_1}, X_{v_p})$. □

The following lemma shows that we can efficiently compute the γ values for each node of a tree-decomposition. The algorithm repeatedly shrinks the tree, by absorbing the subgraphs corresponding to leaves. When the tree is reduced to a single node, the algorithm computes γ using brute force, for this node, since the distances are preserved during absorption. Then, it reverses the shrinking process and expands the tree, using the γ values already computed to compute γ values for the newly expanded nodes.

Lemma 3. *Let G be an n-vertex weighted digraph and let (X, T) be the tree decomposition of G, of treewidth t. For each pair u, v such that $u, v \in X_i$ for some $i \in V(T)$, let $Dist(u, v) = \delta(u, v)$ and $Int(u, v) = x$ where x is some intermediate vertex on a shortest path from u to v. (If $wt(u, v) = \delta(u, v)$, then $Int(u, v) =$ null.) Then in $O(t^4 n)$ time, we can either find a negative cycle in G, or compute the values $Dist(u, v)$ and $Int(u, v)$ for each such pair u, v.*

Proof. Initially all the values $Dist(u, v)$ are set to ∞ and $Int(u, v)$ to null. We give an inductive algorithm.

We use induction on $|V(T)|$. Choose a leaf, l, of T. Run the Bellman-Ford algorithm on $G[X_l]$ in time $O(t^4)$. If $G[X_l]$ contains a negative cycle, it will be found, so henceforth assume that $G[X_l]$ does not contain a negative cycle. Update the values for pairs $u, v \in X_l$ as follows: if the weight of the shortest path found is less than the current value of $Dist(u, v)$, then set $Dist(u, v)$ to the new value and $Int(u, v)$ to any intermediate vertex on the shortest path found. If $wt(u, v)$ is equal to the weight of the shortest path found, then set $Int(u, v) =$ null.

If $|V(T)| = 1$, we are done. Otherwise remove l from T and call the resulting tree T'. Let $V' = \cup_{i \in V(T')} X_i$ and construct G' by absorbing $G[X_l]$ into $G[V']$, where the weight of each added edge $\langle u, v\rangle$ is $\delta_{G[X_l]}(u, v)$. Then, for any vertices $u, v \in V'$, $\delta_{G'}(u, v) = \delta_G(u, v)$, by Lemma 1. In particular, if G contains a negative cycle, so does G'. Note that $(X - X_l, T')$ is a tree-decomposition for G'. Inductively run the algorithm on G'. If a negative cycle is found in G', then a negative cycle in G can be found by replacing any edges added during the absorption by their corresponding paths in $G[X_l]$. Hence, we may assume that G' does not contain a negative cycle.

For $a, b \in V'$, $Dist(a, b) = \delta_{G'}(a, b) = \delta_G(a, b)$, as desired. If $Int(a, b) = x \neq$ null, then x is an intermediate vertex on a shortest a to b path in G' and hence also in G, as desired. If $Int(a, b) =$ null, then $\langle a, b\rangle$ is a shortest path in G'. If $wt(a, b) > Dist(a, b)$, then this edge must have been added during the absorption. Correct the value $Int(a, b)$ by setting it to some intermediate vertex on the corresponding a to b shortest path found in $G[X_l]$. After this, all Int values are correct for $a, b \in V'$.

Construct a digraph G'' by absorbing $G[V']$ into $G[X_l]$, with each added edge $\langle u, v \rangle$ having weight $\delta_G(u, v)$. By Lemma 1, $\delta_{G''}(x, y) = \delta_G(x, y), \forall x, y \in X_l$. Run the Bellman-Ford algorithm on G'' to recompute all pairs shortest paths. Update the values $Dist(a, b)$ and $Int(a, b)$ for $a, b \in X_l$ as before.

For $a, b \in X_l$, $Dist(a, b) = \delta_{G''}(a, b) = \delta_G(a, b)$ as desired. For $a, b \in V' \cap X_l$, $Int(a, b)$ is not changed since $Dist(a, b)$ is already $\delta_G(a, b)$. If either a or b does not belong to $V' \cap X_l$, $Int(a, b) =$ an intermediate vertex on a shortest path in G'' and hence in G, or $Int(a, b) =$ null in which case $wt(a, b) = \delta_G(a, b)$. Thus, the values computed are correct for all pairs a, b which completes the induction. The time analysis follows easily. □

Therefore, we can assume in the following that G does not contain a negative cycle. We will now briefly describe how a shortest path tree, rooted at a given vertex s, is computed. Perform a DFS of T starting at vertex i, where $s \in X_i$, storing at each vertex $j \in V(T)$ the product of the γ values on the path from i to j. Let $y \in V(G)$ and let $j \in V(T)$ such that $y \in X_j$. By Lemma 2, the value stored at vertex j during the DFS, is $P(X_i, X_j)$ which contains the tuple $(s, y, \delta(s, y))$. This implies that for each $y \in V(G)$, we have the distance $\delta(s, y)$. Having the distances, we construct the actual tree by performing a kind of BFS (starting at s) in G based on these distances. Hence, we conclude:

Theorem 1. *Let G and (X, T) be as in Definition 2. Let $s \in V(G)$. In $O(t^4 n)$ time we can compute a shortest path tree rooted at s.*

4 Shortest path and distance queries

For a function f let $f^{(1)}(n) = f(n)$; $f^{(i)}(n) = f(f^{(i-1)}(n))$, $i > 1$. Define $I_0(n) = \lceil \frac{n}{2} \rceil$ and $I_k(n) = \min\{j \mid I_{k-1}^{(j)}(n) \leq 1\}$, $k \geq 1$. The functions $I_k(n)$ decrease rapidly as k increases; note, for example, that $I_1(n) = \lceil \log n \rceil$ and $I_2(n) = \log^* n$. Finally, define $\alpha(n) = \min\{j \mid I_j(n) \leq j\}$. The following theorem was proved in [2, 7].

Theorem 2. *Let $\bullet$ be an associative operator defined on a set S, such that for $q, r \in S$, $q \bullet r$ can be computed in $O(m)$ time. Let T be a tree with n nodes such that each node is labelled with an element from S. Then: (i) for each $k \geq 1$, after $O(mnI_k(n))$ preprocessing, the composition of labels along any path in the tree can be computed in $O(mk)$ time; and (ii) after $O(mn)$ preprocessing, the composition of labels along any path in the tree can be computed in $O(m\alpha(n))$ time.*

We use this in the proof of the following.

Theorem 3. *For any integer t and any $k \geq 1$, let G be an n-vertex weighted digraph of treewidth at most t, whose tree-decomposition can be found in $T(n, t)$*

time. Then, the following hold: (i) After $O(t^4 nI_k(n) + T(n,t))$ time and space preprocessing, distance queries in G can be answered in time $O(t^4 k)$. (ii) After $O(t^4 n + T(n,t))$ time and space preprocessing, distance queries in G can be answered in time $O(t^4 \alpha(n))$.

Proof. First, we compute the tree-decomposition (X,T) of G. By Lemma 3, we compute values $Dist(u,v)$ for u,v such that $u,v \in X_i$ for some $i \in V(T)$. From these values, we can easily compute $\gamma(i)$, $\forall i \in V(T)$. By Theorem 2 we preprocess T so that product queries on γ can be answered. Given a query, $u,v \in V(G)$, let i,j be vertices of T such that $u \in X_i$ and $v \in X_j$. We ask for the product of the γ values on the path between i and j. By Lemma 2, the answer to this query contains the information about $\delta(u,v)$. The bounds follow easily by the ones given in Theorem 2 and by the fact that the composition of any two γ values can be computed in $O(t^4)$ time. □

Theorem 4. *For any integer t and any $k \geq 1$, let G be an n-vertex weighted digraph of treewidth at most t, whose tree-decomposition can be found in $T(n,t)$ time. Then, the following hold: (i) After $O(t^4 nI_k(n) + T(n,t))$ preprocessing, we can answer shortest path queries in G in time $O(t^5 kL)$, where L is the length of the reported path. (ii) After $O(t^4 n + T(n,t))$ preprocessing, we can answer shortest path queries in G in time $O(t^5 \alpha(n) L)$, where L is the length of the reported path.*

Proof. We first compute a tree decomposition (X,T) of G. In the preprocessing phase, we compute the following data structures. Using Lemma 3, we compute the values $Dist(u,v)$ and $Int(u,v)$, for all pairs $u,v \in X_i$, for some $i \in V(T)$. From the $Dist$ values, we compute $\gamma(i)$, $\forall i \in V(T)$. We use Theorem 3 to compute a data structure in $O(t^4 nI_k(n))$ (or in $O(t^4 n)$) time so that distance queries can be answered in time $O(t^4 k)$ (or $O(t^4 \alpha(n))$). Root the tree T arbitrarily. Define, for each vertex $v \in V(G)$, $h(v)$ to be the tree node i such that $v \in X_i$ and i is the closest such node to the root of the tree. Preprocess T so that $h(v)$ can be found in constant time. Such a preprocessing can easily be done with, say, a DFS of T. Further, preprocess T so that lowest common ancestor (LCA) queries can be answered in constant time. Clearly, the time for the preprocessing is dominated by the time required by Theorem 3.

Let the query be for the shortest path between u and v. We first show that it is sufficient to consider the case when $h(u)$ is a descendant of $h(v)$ in T, or vice versa. Suppose $h(u)$ and $h(v)$ are not descendants of each other. Then let i be the LCA of the two. By Proposition 1, a shortest path from u to v passes through some vertex $z \neq u,v$ in X_i, and $\delta(u,v) = \delta(u,z) + \delta(z,v)$. By $O(t)$ queries, we can find this vertex z and then find the shortest paths from u to z and from z to v, and $h(u)$ and $h(v)$ are both descendants of $h(z)$.

Therefore, assume $h(u)$ is a descendant of $h(v)$. (A similar argument holds when $h(v)$ is a descendant of $h(u)$.) The query algorithm first checks if u and v both belong to X_i, for some $i \in V(T)$. In particular, if there exists such an X_i, then u and v appear together in $X_{h(u)}$. If they do, then, if $Int(u,v) =$ null, the algorithm returns the edge $\langle u,v \rangle$. If $Int(u,v) = x \neq$ null, the algorithm recursively queries for the shortest paths from u to x and from x to v, and returns the concatenation of these two paths. Therefore, assume that u and v do not appear together in any X_i. Let p be the parent of $h(u)$ in T. Then, by Proposition 1, there exists a vertex $z \in X_p$ such that a shortest path from u to v passes through z, hence, $\delta(u, v) = \delta(u, z) + \delta(z, v)$. (Note that z may be v.) This vertex can be found with $O(t)$ distance queries. The algorithm recursively queries for the shortest paths from u to z and from z to v, and returns the concatenation of these two paths.

A simple induction shows that the query algorithm returns a path in $O(t^5 kL)$ (or $O(t^5 \alpha(n) L)$) time, where L is the number of edges of the reported path. □

Hence, the results claimed in the Introduction, for digraphs of constant treewidth, are now immediate from Fact 1 and Theorems 3 and 4.

5 Dynamization

In this section we shall give our dynamic data structures and algorithms. The following lemma about graph partitions plays a key role. (The proof is omitted for lack of space.)

Lemma 4. *Given an n-vertex digraph G, a binary tree-decomposition of G of treewidth t and a positive integer $1 \le m \le n$, we can, in $O(t^2 n)$ time, divide G into $q \le 16n/m$ subgraphs $H_1, \ldots, H_q$, and construct another subgraph H' such that: (i) H_i has at most tm vertices and a cut-set $C(H_i)$ of size at most $3t$; (ii) H' is the induced subgraph on vertices $\cup_{i=1}^{q} C(H_i)$, augmented with edges $\langle x, y \rangle$, $x, y \in C(H_i)$ for each $1 \le i \le q$; and (iii) we have a binary tree decomposition of treewidth t for each H_i and a binary tree decomposition for H' of treewidth $3t$.*

Our dynamic algorithm works as follows. Using the above Lemma, it divides the digraph into subgraphs with disjoint edge sets and small cut-sets, and constructs another (smaller) digraph – the reduced digraph – by absorbing each subgraph. The sizes of the subgraphs are chosen so that the subgraphs and the reduced digraph both have size roughly $\sqrt{n}$. The algorithm then constructs a query data structure for each subgraph and for the reduced digraph. Queries can be efficiently answered by querying these data structures. Since the edge sets are disjoint, a change in the weight of an edge affects the data structure for only one subgraph. Then the data structure of this subgraph is updated. This may result in new distances between vertices in its cut-set, which appear in the reduced digraph as changes in the weights of edges between these cut-set vertices. Since the cut-set is small, the weights of only a few edges in the reduced digraph change. The data structure for the reduced digraph is updated to reflect these changes. Thus an update in the original digraph is accomplished by a small number of updates in subgraphs of size $\sqrt{n}$. This idea is recursively applied below to further reduce the update time.

Let $Dyn(G, P, U, Q)$ be a dynamic data structure for a digraph G, where $O(P)$ is the preprocessing time and space to be set up, $O(Q)$ is the time to answer a distance query and $O(U)$ is the time to update it after the modification of an edge-weight.

Theorem 5. *For all positive integers t, r, given an n-vertex weighted digraph G, and a binary tree-decomposition of G of treewidth t, we can construct the following dynamic data structures: (i) $Dyn(G, c^r t^3 n, c^{2r} t^2 n^{(1/2)^{r-1}}, c^{2r} t^2 \alpha(n))$; and (ii) $Dyn(G, c^r t^3 n I_k(n), c^{2r} t^2 n^{(1/2)^{r-1}}, c^{2r} t^2 k)$, where $c = 3^{r+2} t$.*

Proof. We shall prove part (i). Part (ii) can be proved similarly. We use induction on r. For $r = 1$, the basis is given by the static data structure of Theorem 3, with updates implemented by simply recomputing the data structure.

We use the notation $D(G, n, r, t)$ for $Dyn(G, c^r t^3 n, c^{2r} t^2 n^{(1/2)^{r-1}}, c^{2r} t^2 \alpha(n))$. Assume that the theorem holds for any $r' < r$. We show how to construct $D(G, n, r, t)$.

Use Lemma 4 (with parameter $m = 8\sqrt{n}$) to divide G into subgraphs $H_1, \ldots, H_q$, $q \leq 2\sqrt{n}$, each with at most $8t\sqrt{n}$ vertices and construct H' which has at most $(3t)2\sqrt{n}$ vertices. Define G_i to be H_i with all edges joining pairs of vertices in its cut-set deleted. Define G' to be H' with edges $\langle x, y \rangle$ weighted $\delta_{G_i}(x, y)$ for each pair $x, y \in C(G_i)$, $1 \leq i \leq q$. Replace multiple edges by the edge of minimum weight. Note that G' is exactly the graph obtained by absorbing $G_1, G_2, \ldots, G_q$ into the rest of the graph. By Lemma 1, it follows that $\delta_{G'}(x, y) = \delta_G(x, y),\ \forall x, y \in V(G')$.

Let $u \in V(G_i), v \in V(G_j) - V(G_i)$. Then, any path from u to v must pass through a vertex in each of the cut-sets of G_i and G_j. Then we have $\delta_G(u, v) = \min\{\delta_{G_i}(u, x) + \delta_{G'}(x, y) + \delta_{G_j}(y, v) :\ x \in C(G_i),\ y \in C(G_j)\}$. Similarly, for $u, v \in V(G_i)$, we have $\delta_G(u, v) = \min\{\delta_{G_i}(u, v), \min\{\delta_{G_i}(u, x) + \delta_{G'}(x, y) + \delta_{G_i}(y, v) :\ x, y \in C(G_i)\}\}$. If we are able to make queries of the form $\delta_{G_i}(x, y)$ and $\delta_{G'}(x, y)$, the above directly yields a query algorithm for any pair of vertices x, y.

Write n_i for $|V(G_i)|$ and n' for $|V(G')|$. Note that Lemma 4 also gives us a tree-decomposition of treewidth t for each subgraph G_i, and a tree-decomposition of treewidth $3t$ for G'. Thus we can inductively construct $D(G_i, n_i, r-1, t)$ for each $1 \leq i \leq q$, which enables us to answer queries of the form $\delta_{G_i}(x, y)$, and $D(G', n', r-1, 3t)$ which enables us to answer queries of the form $\delta_{G'}(x, y)$. The data structure $D(G, n, r, t)$ is the union of the above data structures.

The update procedure is the following: note that $E(G_i) \cap E(G_j) = \emptyset,\ i \neq j$ and $E(G_i) \cap E(G') = \emptyset$, i.e. each edge of G belongs to exactly one of the G_i's or to G'. Suppose the cost of an edge belonging to G_i is changed. Then, we update the data structure for G_i. This may result in new values for $\delta_{G_i}(x, y),\ x, y \in C(G_i)$. We query the updated data structure for $\delta_{G_i}(x, y),\ x, y \in C(G_i)$ and change the weights of the corresponding edges of G', updating the data structure for G' after each change. That the procedure is correct follows from the fact that changing the cost of an edge in G_i does not change $\delta_{G_j}(x, y),\ x, y \in C(G_j)$ when $j \neq i$. Thus, after we change, in G', the cost of edges $\langle x, y \rangle,\ x, y \in C(G_i)$, we have $\delta_{G'}(u,v) = \delta_G(u,v),\ u,v \in V(G')$, again, by repeated applications of Lemma 1. After the last update, the data structure for G' yields correct distances in G, between vertices in $V(G')$. Now suppose we change the cost of an edge belonging to G'. Then the distances $\delta_{G_i}(x,y)$ do not change. Thus, in this case, we simply update the data structure for G'. This completes the description of the preprocessing and update algorithms.

Let the time taken for preprocessing, querying and updating $D(G,n,r,t)$ be $P(r,t)n$, $Q(r,t)\alpha(n)$ and $U(r,t)n^{(1/2)^{r-1}}$, respectively. Writing $N = \max\{n_i : 1 \leq i \leq q\}$, we have the following recurrences:

$$\begin{aligned} P(r,t)n &\leq t^4 n + \sum_{i=1}^{q} P(r-1,t)N + P(r-1,3t)n' \\ Q(r,t)\alpha(n) &\leq (3t)^2[2Q(r-1,t)\alpha(N) + Q(r-1,3t)\alpha(n')] \\ U(r,t)n^{(1/2)^{r-1}} &\leq U(r-1,t)N^{(1/2)^{r-2}} + \\ &\quad (3t)^2[Q(r-1,t)\alpha(N) + U(r-1,3t)(n')^{(1/2)^{r-2}}] \end{aligned}$$

The terms in the recurrence for $P(r,t)n$ are for constructing the G_i's and G using Lemma 4, for constructing $D(G_i,n_i,r-1,t)$ for each G_i and for constructing $D(G',n',r-1,3t)$. The terms in the recurrence for $Q(r,t)\alpha(n)$ are for the two queries in G_i and G_j and for the query in G', which have to be made for each pair of vertices, one in the cut-set of G_i and one of G_j. The terms in the update recurrence are for updating G_i, and then updating the edges in G' between vertices in the cut-set of G_i.

By construction, $n', N \leq 8t\sqrt{n}$. The sum of the number of vertices in each G_i cannot exceed the number of vertices in the initial tree decomposition, so $\sum_{i=1}^{q} n_i \leq 2tn$. Making these substitutions in the above recurrences and estimating gives:

$$\begin{aligned} P(r,t)n &\leq t^4 n + 2tnP(r-1,t) + 8t\sqrt{n}P(r-1,3t) \leq 9tP(r-1,3t)n \\ Q(r,t)\alpha(n) &\leq (3t)^2[2Q(r-1,t)\alpha(8t\sqrt{n}) + Q(r-1,3t)\alpha(8t\sqrt{n})] \\ &\leq 3(3t)^2 Q(r-1,3t)\alpha(n) \\ U(r,t)n^{(1/2)^{r-1}} &\leq U(r-1,t)(8t\sqrt{n})^{(1/2)^{r-2}} + (3t)^2[Q(r-1,t)\alpha(8t\sqrt{n}) \\ &\quad + U(r-1,3t)(8t\sqrt{n})^{(1/2)^{r-2}}] \leq (3t)^2 16tU(r-1,3t)n^{(1/2)^{r-1}} \end{aligned}$$

It is easily verified that the claimed bounds satisfy the recurrences above. Thus we can construct $D(G,n,r,t)$, completing the induction. □

The next theorem follows directly from Fact 1 and Theorem 5 with $r = 1 - \log\beta$.

Theorem 6. *Let $k \geq 1$ be any constant integer and let $0 < \beta < 1$ be any constant. Given an n-vertex weighted digraph G of constant treewidth, we can construct: (i) $Dyn(G,n,n^\beta,\alpha(n))$; and (ii) $Dyn(G,nI_k(n),n^\beta,k)$.*

The algorithms described above give answers to distance queries only. They can be modified to answer path queries as well, in time $O(kL)$ (or $O(L\alpha(n))$). Also, before running our update procedure after a change in the weight of an edge, we have to ensure that this change does not create a negative cycle in G. This can be easily tested as follows. Let $\langle u, v\rangle$ be an edge with weight $wt(u, v)$ and let $wt'(u, v)$ be its new weight. Clearly, the new weight $wt'(u, v)$ creates a negative cycle in G iff $\delta_G(v, u) + wt'(u, v) < 0$. This test takes time proportional to that of finding $\delta_G(v, u)$ and hence does not affect our update bound.

Acknowledgement. We would like to thank Hans Bodlaender for many interesting discussions concerning the treewidth of graphs.

References

1. R. Ahuja, T. Magnanti and J. Orlin, "Network Flows", Prentice-Hall, 1993.
2. N. Alon and B. Schieber, "Optimal Preprocessing for Answering On-line Product Queries", Tech. Rep. No. 71/87, Tel-Aviv University, 1987.
3. S. Arnborg, "Efficient Algorithms for Combinatorial Problems on Graphs with Bounded Decomposability - A Survey", *BIT*, 25, pp.2-23, 1985.
4. H. Bodlaender, "A Linear Time Algorithm for Finding Tree-decompositions of Small Treewidth", *Proc. 25th ACM STOC*, pp.226-234, 1993.
5. H. Bodlaender, "A Tourist Guide through Treewidth", *Acta Cybernetica*, Vol.11, No.1-2, pp.1-21, 1993.
6. H. Bodlaender, "Dynamic Algorithms for Graphs with Treewidth 2", *Proc. 19th WG'93*, LNCS 790, pp.112-124, Springer-Verlag, 1994.
7. B. Chazelle, "Computing on a Free Tree via Complexity-Preserving Mappings", *Algorithmica*, 2, pp.337-361, 1987.
8. H. Djidjev, G. Pantziou and C. Zaroliagis, "On-line and Dynamic Algorithms for Shortest Path Problems", *Proc. 12th STACS*, 1995, LNCS 900, pp.193-204, Springer-Verlag.
9. E. Feuerstein and A.M. Spaccamela, "Dynamic Algorithms for Shortest Paths in Planar Graphs", *Theor. Computer Science*, 116 (1993), pp.359-371.
10. G.N. Frederickson, "Fast algorithms for shortest paths in planar graphs, with applications", *SIAM J. on Computing*, 16 (1987), pp.1004-1022.
11. G.N. Frederickson, "Planar Graph Decomposition and All Pairs Shortest Paths", *J. ACM*, 38(1991), pp.162-204.
12. G.N. Frederickson, "Searching among Intervals and Compact Routing Tables", *Proc. 20th ICALP*, 1993, LNCS 700, pp.28-39, Springer-Verlag.
13. G.N. Frederickson, "Using Cellular Graph Embeddings in Solving All Pairs Shortest Path Problems", accepted in *J. of Algorithms*, 1994.
14. M. Fredman and R. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms", *J. ACM*, 34(1987), pp. 596-615.
15. D. Kavvadias, G. Pantziou, P. Spirakis and C. Zaroliagis, "Efficient Sequential and Parallel Algorithms for the Negative Cycle Problem", *Proc. 5th ISAAC*, 1994, LNCS 834, pp.270-278, Springer-Verlag.
16. P. Klein, S. Rao, M. Rauch and S. Subramanian, "Faster shortest-path algorithms for planar graphs", *Proc. 26th ACM STOC*, 1994, pp.27-37.
17. N. Robertson and P. Seymour, "Graph Minors II: Algorithmic Aspects of Treewidth", *J. Algorithms*, 7(1986), pp.309-322.