Shortest Paths in Digraphs of Small Treewidth.
Part II: Optimal Parallel Algorithms

Shiva Chaudhuri Christos D. Zaroliagis

MPI–I–95–1-021 August 1995

Shortest Paths in Digraphs of Small Treewidth. Part II: Optimal Parallel Algorithms *

SHIVA CHAUDHURI CHRISTOS D. ZAROLIAGIS

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

August 9, 1995

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 parallel algorithms for the EREW PRAM model of computation 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 using a single processor, after a preprocessing of $O(\log^2 n)$ time and $O(n)$ work, where $\alpha(n)$ is the inverse of Ackermann's function. The class of constant treewidth graphs contains outerplanar graphs and series-parallel graphs, among others. To the best of our knowledge, these are the first parallel algorithms which achieve these bounds for any class of graphs except trees. We also give a dynamic algorithm which, after a change in an edge weight, updates our data structures in $O(\log n)$ time using $O(n^\beta)$ work, for any constant $0 < \beta < 1$. Moreover, we give an algorithm of independent interest: computing a shortest path tree, or finding a negative cycle in $O(\log^2 n)$ time using $O(n)$ work.

Keywords: shortest path, graph theory, treewidth, parallel computing, PRAM, dynamic algorithm.

*This work was partially supported by the EU ESPRIT Basic Research Action No. 7141 (ALCOM II).

1 Introduction

Finding shortest paths in digraphs is a fundamental problem in network optimization [2]. Given an n-vertex, m-edge digraph G with real edge weights, the shortest paths problem asks for paths of minimum weight between vertices in G. 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 [2].

For general digraphs the best parallel algorithm for the apsp problem takes $O(\log^2 n)$ time using $O(n^3)$ work[1] on an EREW PRAM [11]. In the case of planar digraphs there is an $O(\log^4 n)$-time, $O(n^2)$-work EREW PRAM algorithm [9]. An apsp algorithm must output paths between $\Omega(n^2)$ vertex pairs and thus requires this much work and space. For sparse digraphs (i.e. $m = O(n)$) 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. For example, for *outerplanar* digraphs, it was shown in [10] that after preprocessing requiring $O(\log n)$ time and $O(n \log n)$ work on a CREW PRAM, a distance query is answered in $O(\log n)$ time using a single processor and a shortest path query in $O(\log n)$ time using $O(L + \log n)$ work (where L is the number of edges of the reported path). In [10] it is also shown how distance queries in planar digraphs can be answered in $O(\log n + \log^2 q)$ time using $O(\log n + q)$ work, after polylog-time and $O(n \log n \log^* n + q^{1.5})$-work preprocessing on a CREW PRAM. These latter bounds are given in terms of a minimum number of faces q that collectively cover all vertices of the planar digraph. Note that q varies from 1 (outerplanar digraph) up to $\Theta(n)$.

The study of graphs using the *treewidth* as a parameter was pioneered by Robertson and Seymour [15, 16] and continued by many others (see e.g. [4, 6]). Informally, the treewidth is a measure of how close the structure of the graph is to a tree (see Section 2 for a formal definition). Graphs of treewidth t are also known as partial t-trees. These graphs have at most tn edges. 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 outerplanar graphs, series-parallel graphs, graphs with bounded bandwidth and cutwidth and many other classes [4, 6]. Thus, giving efficient algorithms parameterized by treewidth is an important step in the development of better algorithms for many natural classes of sparse graphs.

In this paper we consider the problem of preprocessing a digraph of small treewidth in parallel, so that afterwards, queries can be efficiently answered using a single processor. We also consider the dynamic version of the problem, where edge weights may change. In [8] sequential

[1] work = time $\times$ number of processors, or alternatively the total number of operations.

algorithms are given that, for digraphs of constant treewidth, after $O(n)$ time preprocessing answer a distance (resp. shortest path) query in $O(\alpha(n))$ (resp. $O(L\alpha(n))$) time[2]. After a change in an edge weight, the algorithm updates the data structure in $O(n^\beta)$ time, for any constant $0 < \beta < 1$.

The main contribution of this paper is an algorithm that achieves optimal parallelization, on the EREW PRAM, of the above results. For digraphs of constant treewidth, after $O(\log^2 n)$ time and $O(n)$ work preprocessing, our algorithm answers a distance query in $O(\alpha(n))$ time using a single processor and a shortest path query in $O(\alpha(n)\log n)$ time using $O(L + \alpha(n)\log n)$ work. Updates can be performed in $O(\log n)$ time using $O(n^\beta)$ work for any constant $0 < \beta < 1$. This improves all previous parallel results for this class of graphs. Moreover, it improves the results in [10] for outerplanar digraphs in many ways: it improves the preprocessing and distance query bounds, it runs on the weakest PRAM model and it applies to a larger class of graphs. We note that the time bottleneck in preprocessing is the computation of the *tree-decomposition* (see Section 2) of the input graph. If an explicit tree-decomposition of the graph is also provided with the input, then the preprocessing time is $O(\log n)$.

As in [8], we give a tradeoff between the preprocessing work and the query bounds. For bounded treewidth digraphs, after $O(nI_k(n))$ preprocessing, we can answer distance (resp. shortest path) queries in $O(k)$ (resp. $O(k\log n)$) time using a single processor (resp. using $O(L + k\log n)$ work), for an integer $1 \leq k \leq \alpha(n)$. $I_k(n)$ is a function that decreases rapidly with k (see Section 3). In particular $I_1(n) = \lceil \log n \rceil$ and $I_2(n) = \log^* n$.

A solution to the single-source problem consists of a *shortest path tree* rooted at a given vertex. A shortest path tree exists iff there is no negative weight cycle in the digraph. In parallel computation, the best algorithm for constructing a shortest path tree (or finding a negative cycle) in a general digraph G takes as much time as computing apsp in G [11]. Some improvements have been made for outerplanar [10] and planar digraphs [9] with no negative cycles. In those papers, a shortest path tree can be computed in $O(\log^2 n)$ time, after a preprocessing of the input digraph. The preprocessing work of [9] is $O(n^{1.5})$ on an EREW PRAM, while the preprocessing work in [10] is $O(n\log n)$ on a CREW PRAM. Even with randomization allowed, and the weights restricted to being positive integers, for planar digraphs, the best polylog-time algorithm uses n processors (and hence $\Omega(n\log n)$ work) on an EREW PRAM. Although, on a CREW PRAM, a negative cycle in an outerplanar digraph can be found in $O(\log n \log^* n)$ time and $O(n)$ work, this algorithm does not construct the shortest path tree [13]. Hence, the work for finding a shortest path tree in polylog-time is $\Omega(n\log n)$, even for the case of outerplanar digraphs.

We give also in this paper an algorithm to construct a shortest path tree (or find a negative cycle) in digraphs of constant treewidth that runs on an EREW PRAM in $O(\log^2 n)$ time using $O(n)$ work (Section 3). If a tree-decomposition is also provided with the input, then the

[2] $\alpha(n)$ is the inverse of Ackermann's function [1] and is a very slowly growing function.

algorithm runs in $O(\log n)$ time. To the best of our knowledge, this is the first deterministic parallel algorithm for the shortest path tree problem that achieves $O(n)$ work.

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 $O(\log^2 n)$ time using $O(n)$ work on an EREW PRAM [7]. Our approach in this paper follows the one in [8]: a certain value is defined for each node of the tree-decomposition, along with an associative operator on these values, and then it is shown that the shortest path problem reduces to computing products of these values along paths in the tree-decomposition. However, our parallel algorithms presented here, that implement the above approach, require different techniques and thus constitute a non-trivial parallelization of the methods in [8]. Our preprocessing vs. query trade-off arises from a similar trade-off in [3], where parallel algorithms are given to compute the product of node values along paths in a tree. The dynamization of our data structures is partially based on a graph equipartitioning result which is of independent interest.

The rest of the paper is organized as follows. Section 2 contains preliminary results and basic definitions. In Section 3 we give our static data structures, as well as the algorithm for computing a shortest path tree or finding a negative cycle. Finally, in Section 4 we give our dynamic data structures. For the sake of completeness, we repeat, throughout the paper, the necessary definitions and results from [8].

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 $wt : E(G) \longrightarrow \mathbb{R}$. We call $wt(u, v)$ the *weight* of the 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 [2] 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 spanning 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 by 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.

Often, we will want to focus on a subgraph induced by a subset of the vertices of a graph, however, we would like the distances between vertices in this subgraph to be the same as in the original graph. 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.) The following lemma, proved in [8], shows that absorbing a subgraph into another preserves distances.

Lemma 2.1 *Let H_1 and H_2 be subgraphs of H and let H_1' be obtained by absorbing H_2 into H_1. Then, for all $x, y \in V(H_1')$, $\delta_{H_1'}(x, y) = \delta_H(x, y)$.*

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 2.1 *[7] Given a constant $t \in \mathbb{N}$ and an n-vertex graph G, there exists an EREW PRAM algorithm, running in $O(\log^2 n)$ time using $O(n)$ work, which tests whether G has treewidth at most t and if so, outputs a tree-decomposition (X, T) of G with treewidth at most t.*

Fact 2.2 *[5, 7] Given a constant $t \in \mathbb{N}$ and a tree-decomposition of treewidth t of an n-vertex graph G, we can compute a rooted, binary tree-decomposition of G with depth $O(\log n)$ and treewidth at most $3t + 2$, in $O(\log n)$ time using $O(n)$ work on an EREW PRAM.*

We shall call the tree-decomposition found in Fact 2.2 *balanced*. Given a tree-decomposition of G, we can easily find separators in G, as the following proposition shows.

Proposition 2.1 *[16] Let G be a graph, (X, T), its tree-decomposition, $e = (i, j) \in E(T)$ and T_1 and T_2 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 The Static Data Structures

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, in particular, I_1 behaves like $\log n$ and I_2 like $\log^* n$. Define $\alpha(n) = \min\{j \mid I_j(n) \leq 1\}$. The following was proved in [3].

Fact 3.1 *[3] Let $\bullet$ be an associative operator defined on a set S, such that for $x, y \in S$, $x \bullet y$ can be computed in $O(m)$ time and $O(w)$ work. 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 $O(m \log n)$-time and $O(wnI_k(n))$-work preprocessing on an EREW PRAM, the composition of labels along any path in the tree can be computed in $O(wk)$ time by a single processor; and (ii) after $O(m \log n)$-time and $O(wn)$-work preprocessing on an EREW PRAM, the composition of labels along any path in the tree can be computed in $O(w\alpha(n))$ time by a single processor.*

The main idea of our algorithm is, as in [8], to reduce shortest path computations to the above problem. This reduction is done by first defining a certain value for each node of the tree-decomposition of G, as well as an associative operator on these values, and then showing that shortest path computation reduces to computing products of those values along paths in the tree-decomposition. Then, the rest follows by Fact 3.1. A brief description of the reduction follows.

A tuple (a, b, c) is called a *distance tuple* if a, b are arbitrary symbols and $c \in \mathbb{R}$. Let (a_1, b_1, c_1) and (a_2, b_2, c_2) be two distance tuples. Then, their product is defined as $(a_1, b_1, c_1) \otimes (a_2, b_2, c_2) = (a_1, b_2, c_1 + c_2)$ if $b_1 = a_2$ and as nonexistent otherwise. Let M be a set of distance tuples and define minmap$(M) = \{(a, b, c) : \ (a, b, c) \in M$ and $\forall (a', b', c') \in M$ if $a' = a, b' = b$, then $c \leq c'\}$, In other words, minmap retains, among all tuples with the same first and second components, the one with the smallest third component.

For two sets M_1 and M_2 of distance tuples define the operator $\circ$ by $M_1 \circ M_2 = \text{minmap}(M)$, where $M = \{x \otimes y : \ x \in M_1, \ y \in M_2\}$. It can be easily verified that $\circ$ is an associative operator.

Consider now a digraph G with real edge weights. The above definition actually says that, 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. Finally, define $P(X, Y) = \{(a, b, \delta_G(a, b)) : \ a \in X, b \in Y\}$, where $X, Y \subseteq V(G)$ and X, Y are not necessarily distinct. (By definition, $P(X, X)$ contains tuples $(x, x, 0)$, $\forall x \in X$.)

The following lemma, proved in [8], establishes the desired connection between computing shortest paths and products along tree paths of the operator $\circ$ defined above.

Lemma 3.1 *Let G be a weighted digraph and (X,T) its tree decomposition. For $i \in V(T)$, define $\gamma(i) = P(X_i, X_i)$. Let $v_1, \ldots, v_p$ be a path in T. Then $\gamma(v_1) \circ \ldots \circ \gamma(v_p) = P(X_{v_1}, X_{v_p})$.*

Therefore, it only remains to show how the γ values can be efficiently computed in parallel for each node of a tree-decomposition. This is shown in the next lemma. The following algorithm first converts the given tree-decomposition into a balanced one, and then repeatedly shrinks the tree. The shrinking is accomplished by processing the tree bottom-up and absorbing, in every stage, the subgraphs corresponding to leaves. When the tree is reduced to a single node, the algorithm computes γ using any known method, for this node. Since distances are preserved during absorption, the distances computed for this single node are the distances in the original graph. Finally, the shrinking process is reversed and the tree is expanded. The γ values of the newly expanded nodes can be computed using the γ values of the nodes computed so far.

Lemma 3.2 *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 $Dist(u,v) = \delta(u,v)$. Then, in $O(\log n \log^2 t)$ time using $O(t^3 n)$ work on an EREW PRAM, we can either find a negative cycle in G, or compute the values $Dist(u,v)$ for each such pair u, v.*

Proof. Initially $Dist(u,v) = wt(u,v)$, if $\langle u,v \rangle \in E(G)$, and $Dist(u,v) = \infty$, otherwise. We give an inductive algorithm. First, convert (X,T) into a balanced tree-decomposition of G using Fact 2.2. Then, for each vertex of T, we compute its *level number*, which is one more than the level of its parent, with the root having level number 1. This computation can be done in $O(\log n)$ time and $O(n)$ work ([12], Theorem 3.4).

We use induction on the number of levels of T. Let d be the depth of T and N_d be the set of tree nodes at level d. For all nodes $z \in N_d$, run the algorithm of [11] to solve the apsp problem in $G[X_z]$. This will take $O(\log^2 t)$ time and $O(|N_d| t^3)$ work. If there is a negative cycle in some $G[X_z]$, it will be found by the algorithm of [11]. If this is the case, then stop and report the cycle. Otherwise, assume henceforth that there is no $G[X_z]$ containing a negative cycle. For all $u, v \in X_z$ and $\forall z \in N_d$, update the values $Dist(u,v)$ 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.

If $d = 1$ (which implies that $|V(T)| = 1$), we are done. Otherwise, remove all nodes $z \in N_d$ from T and call the resulting tree T'. Let $V' = \cup_{i \in V(T')} X_i$ and construct G' by absorbing every $G[X_z]$ into $G[V']$, where the weight of each added edge $\langle u,v \rangle$ is $\delta_{G[X_z]}(u,v)$. (The absorption is done in two steps: first all $G[X_z]$ are absorbed, where z is a left child, and then all $G[X_z]$ for which z is a right child.) After the absorptions, we have, by Lemma 2.1, that for any vertices $u, v \in V'$, $\delta_{G'}(u,v) = \delta_G(u,v)$. Moreover, if G contains a negative cycle, so does G'. Let $Y = \cup_{z \in N_d} X_z$. Then, note that $(X - Y, T')$ 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 the subgraphs $G[X_z]$, $z \in N_d$, and the algorithm stops. Otherwise, we assume that G' does not contain a negative cycle and for $a, b \in V'$, $Dist(a,b) = \delta_{G'}(a,b) = \delta_G(a,b)$, as desired.

Construct a digraph G'' by absorbing $G[V']$ into every $G[X_z]$, with each added edge $\langle u, v \rangle$ having weight $\delta_G(u,v)$. By Lemma 2.1, $\delta_{G''}(x,y) = \delta_G(x,y)$, $\forall x, y \in X_z$. Run the algorithm of [11] on G'' to recompute all pairs shortest paths. Update the values $Dist(a,b)$ for $a, b \in X_z$ as before. Now for each $z \in N_d$ and $\forall a, b \in X_z$, $Dist(a,b) = \delta_{G''}(a,b) = \delta_G(a,b)$ as desired. Thus, the values computed are correct for all pairs a, b which completes the induction and the description of the algorithm.

Concerning the resource bounds, it suffices to notice that the algorithm performs a bottom-up and a top-down traversal of T by processing the tree level-by-level and visiting every tree node at most twice. (Either of the traversals can be done in $O(\log |T|)$ time with $O(|T|)$ work on an EREW PRAM, using standard techniques [12].) At each level, the algorithm takes $O(\log^2 t)$ time using $O(t^3)$ work per node. Hence, in total it takes $O(\log n \log^2 t)$ time and $O(t^3 n)$ work on an EREW PRAM. ∎

We are now ready to give our static algorithms. The preprocessing algorithm consists of three steps. First, compute a tree-decomposition (X, T) of the input weighted digraph G, using Fact 2.1. Second, use Lemma 3.2 to either find a negative cycle (and in such a case stop), or compute values $Dist(u,v)$ for u, v such that $u, v \in X_i$ for some $i \in V(T)$. Having these values, compute $\gamma(i)$, $\forall i \in V(T)$. Third, use Fact 3.1 to preprocess T so that product queries on γ can be answered. The distance query algorithm is as follows. Let $u, v \in V(G)$ be the query vertices and let $u \in X_i$ and $v \in X_j$. Then, simply ask for the product of the γ values on the path in T between i and j. By Lemma 3.1, the answer to this product contains the information about $\delta(u,v)$. The next theorem follows easily by the description of the algorithms, the bounds in Fact 3.1 and Lemma 3.2, and by the fact that the composition of any two γ values can be computed in $O(\log^2 t)$ time using $O(t^3)$ work.

Theorem 3.1 *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 $T(n,t)$ parallel time using $W(n,t)$ work on an EREW PRAM. Then, the following hold on an EREW PRAM: (i) After $O(T(n,t) + \log n \log^2 t)$ time and $O(W(n,t) + t^3 n I_k(n))$ work and space preprocessing, distance queries in G can be answered in $O(t^3 k)$ time using a single processor. (ii) After $O(T(n,t) + \log n \log^2 t)$ time and $O(W(n,t) + t^3 n)$ work and space preprocessing, distance queries in G can be answered in $O(t^3 \alpha(n))$ time using a single processor.*

In [8] it is shown how a distance query of time Q yields a shortest path query of time $O(LQ)$, where L is the number of edges of the reported path. That approach, while simple, is not parallelizable. For this reason, a different approach is followed here which is described in the next theorem.

Theorem 3.2 *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 $T(n,t)$ parallel time using $W(n,t)$ work on an EREW PRAM. Then, the following hold on an EREW PRAM: (i) After $O(T(n,t)+\log n \log^2 t)$ time and $O(W(n,t)+t^3 n I_k(n))$ work and space preprocessing, shortest path queries in G can be answered in $O(t^4 k \log n)$ time using $O(t^4(L+k\log n))$ work, where L is the number of edges of the reported path. (ii) After $O(T(n,t)+\log n \log^2 t)$ time and $O(W(n,t)+t^3 n)$ work and space preprocessing, shortest path queries in G can be answered in in $O(t^4 \alpha(n)\log n)$ time using $O(t^4(L+\alpha(n)\log n))$ work, where L is the number of edges of the reported path.*

Proof. Let (X,T) be the tree-decomposition of G. Make T balanced using Fact 2.2. The preprocessing phase consists of the following steps. Use Lemma 3.2 to compute the values $Dist(u,v)$ for all pairs $u,v \in X_i$, for some $i \in V(T)$ and consequently the $\gamma(i)$ values, for all $i \in V(T)$. Use Theorem 3.1 to compute a parallel data structure so that distance queries between any two vertices in G can be answered in $O(t^3\alpha(n))$ (or $O(t^3 k)$) time by a single processor. Use the algorithm of [17] to preprocess T so that lowest common ancestor (LCA) queries can be answered in $O(1)$ time. For each $v \in V(G)$ define $h(v)$ to be the tree node i such that $v \in X_i$ and i is the closest such node to the root. The values $h(v)$, $\forall v \in V(G)$, can be found by a top-down, level-by-level traversal of T, where the processor associated with a node $i \in V(T)$ forks two other processors and associates them with the children of i. It is easy to see that the resource bounds for the preprocessing are dominated by those of Theorem 3.1.

Let $p \in V(T)$. Denote by T_p the subtree of T rooted at p and by $G[T_p]$ the subgraph of G induced on $\bigcup_{i \in V(T_p)} X_i$. Consider a set X_p after the above preprocessing. Each edge $\langle a,b\rangle \in X_p$ is either a real edge (i.e. $\langle a,b\rangle \in E(G)$), or it is an egde added during some absorption (Lemma 3.2). For each node $p \in V(T)$ and $\forall \langle a,b\rangle \in X_p$, define $R_p(a,b)$ as follows:

$$R_p(a,b) = \begin{cases} p & \text{if } \delta_G(a,b) = \delta_{G[X_p]}(a,b) \\ r & \text{if } \delta_G(a,b) = \delta_{G[T_p]}(a,b) \text{ and } r \text{ is a child of } p \\ q & \text{if } \delta_G(a,b) = \delta_{G[T-T_p]}(a,b) \text{ and } q \text{ is the parent of } p \end{cases}$$

The value of all $R_p(a,b)$ can be easily computed during the preprocessing phase. In particular, during the execution of the algorithm implied by Lemma 3.2: when we retain, during some absorption, among multiple edges the one with minimum weight (or similarly when we add a new edge), it is easy to keep a pointer denoting where this minimum weight edge comes from. Hence, the computation of the $R_p(a,b)$ values can be done within the resource bounds of Lemma 3.2.

Let the query be for the shortest path from u to v in G, denoted as $SP(u,v)$. (W.l.o.g. we assume that there is a path from u to v in G.) As in [8], it suffices to consider the case where $h(u)$ is a descendant of $h(v)$, or vice versa. If $h(u)$ and $h(v)$ are not descendants of each other, then by Proposition 2.1 $SP(u,v)$ passes through some vertex $z \neq u,v$ in X_j, where $j = \text{LCA}(h(u),h(v))$,

and $\delta(u,v) = \delta(u,z) + \delta(z,v)$. This vertex z can be found by $O(t)$ distance queries. Hence, to find $SP(u,v)$ it suffices to find $SP(u,z)$ and $SP(z,v)$, and both $h(u)$ and $h(v)$ are descendants of $h(z)$.

We will consider the case where $h(u)$ is a descendant of $h(v)$. (The other case is similar.) Let $path(i,j)$ denote the path in T between nodes i and j and $X_{u,v} = \bigcup_{i \in path(h(u),h(v))} X_i$. Define $EP(u,v)$ to be the shortest path from u to v in $G[X_{u,v}]$. (Note that $G[X_{u,v}]$ is the digraph resulted by absorbing $G[X - X_{u,v}]$ into it.)

The rest of the proof is based on the following three claims.

Claim 1 *$EP(u,v)$ is an encoded version of $SP(u,v)$ and can be found in $O(t^4\alpha(n)\log n)$ (or $O(t^4 k \log n)$) time by a single processor.*

Proof of Claim. The existence of $EP(u,v)$ is guaranteed by Proposition 2.1. The vertices of $EP(u,v)$ can be found as follows. Associate one processor with $h(u)$. If $h(u)$ and $h(v)$ coincide, then the required vertices belong to $X_{h(u)}$ and $EP(u,v)$ is available by the preprocessing phase. Otherwise ($h(u) \neq h(v)$), let p be the parent of $h(u)$ in T. Then, by $O(t)$ distance queries we can find the vertices $x \in X_p$ that belong to $EP(u,v)$. To find the rest of the vertices in $EP(u,v)$, repeat the process with every node in $path(p,h(v))$. The claimed time bound follows easily.

It remains to show that $EP(u,v)$ is a sequence of real edges and/or edges added during some absorption. Let $EP(u,v) = (u = a_1, a_2, \ldots, a_k = v)$. Assume that (a_i, a_{i+1}), for some $1 \leq i \leq k$, is neither a real edge, nor an edge added during an absorption. If a_i and a_{i+1} do not appear together in some X_j, then by Proposition 2.1 there exists a vertex $b \neq a_i, a_{i+1}$ in the shortest path from a_i to a_{i+1}. But this implies that a_i and a_{i+1} are not consecutive in $EP(u,v)$, a contradiction. Hence, a_i and a_{i+1} must appear together in some X_j. Since $(a_i, a_{i+1}) \notin E(G)$ and (a_i, a_{i+1}) was not added during an absorption, we have that: either (i) there is no path in G from a_i to a_{i+1}, or (ii) there is an intermediate vertex $b \neq a_i, a_{i+1}$ in the shortest path from a_i to a_{i+1}. But (i) implies that there is no u to v path in G, a contradiction, and (ii) implies that a_i and a_{i+1} are not consecutive in $EP(u,v)$, again a contradiction. ∎

Let $\langle x,y \rangle$ be an edge of $EP(u,v)$ added during some absorption and let also $\langle x,y \rangle \in X_{i_k}$, $1 \leq k \leq \ell$, where i_1 is a descendant of i_ℓ in T and $path(i_1, i_\ell)$ is a subpath of $path(h(u), h(v))$. Define $g(x,y)$ as follows: (i) if $R_{i_{k-1}}(x,y) = i_k$ and $R_{i_{k+1}}(x,y) = i_k$, $1 \leq k \leq \ell$, then $g(x,y)$ is the subtree of T rooted at $R_{i_k}(x,y)$; (ii) if $R_{i_2}(x,y) = i_1$, then $g(x,y) = T_{i_1}$; (iii) if $R_{i_{\ell-1}}(x,y) = i_\ell$, then $g(x,y) = T - T_{i_{\ell-1}}$. Define the *attachment node* of $g(x,y)$ to be the root of the corresponding subtree in cases (i) and (ii), or the leaf i_ℓ of subtree $T - T_{i_{\ell-1}}$ in case (iii).

Claim 2 *Let $\langle x,y \rangle$ be as above. Then, $g(x,y)$ is the subtree of T containing $SP(x,y)$.*

Proof of Claim. Assume that $SP(x,y)$ is not totally contained in $g(x,y)$. Then, there must be at least one vertex $b \in X_{i_k}$ (or in X_{i_1}, X_{i_ℓ}, resp.) such that $SP(x,y)$ passes through b. (Such

a vertex exists by the continuity condition.) But this implies that $\langle x, y\rangle$ is not a shortest path itself in X_{i_k} (or in X_{i_1}, X_{i_ℓ}, resp.), a contradiction since $\langle x, y\rangle$ is an edge of $EP(u, v)$. ∎

The values $g(x, y)$, $\forall \langle x, y\rangle \in EP(u, v)$ added during some absorption, can be found in $O(t \log n)$ time by a single processor performing a bottom-up traversal of $path(h(u), h(v))$: at each node of the path, the processor checks which case of the definition of $g(x, y)$ applies and assigns a value to $g(x, y)$ accordingly.

Claim 3 *Let $\langle x, y\rangle \in EP(u, v)$ and $\langle x, y\rangle$ has been added during some absorption. Let also $L(x, y)$ be the number of edges of $SP(x, y)$. Then, in $O(t \log n)$ time and $O(tL(x, y))$ work on an EREW PRAM, we can output $SP(x, y)$.*

Proof of Claim. By Claim 2, it suffices to consider only the subtree $g(x, y)$. Let r be the attachment node of $g(x, y)$. If $SP(x, y)$ consists of a single edge, then we visit the neighbor z of r, such that $R_r(x, y) = z$. Otherwise, we may have to visit both neighbors of r, depending on the $R_r()$ values of the edges in the shortest path from x to y in $G[X_r]$. In this case, the processor associated with r forks two other processors and associates them with the neighbors of r. Repeat the above process inductively at the neighbors of r. Since we have to output a path of $L(x, y)$ edges, we have to visit (in the worst case) $L(x, y)$ nodes of $g(x, y)$ and hence the total work is $O(tL(x, y))$. At each node j of $g(x, y)$ its associated processor takes $O(t)$ time. (This is needed to avoid concurrent accesses.) Since the depth of $g(x, y)$ can be $O(\log n)$ in the worst-case, the total time complexity is $O(t \log n)$. ∎

Hence, to output $SP(u, v)$, it suffices (by Claim 1) to find $EP(u, v)$ and then to output the real shortest paths in G which correspond to the edges of $EP(u, v)$ added during some absorptions. Claims 2 and 3 imply that we can do this in work proportional to the size of the real shortest paths. Therefore, $SP(u, v)$ can be output in $O(t^4 \alpha(n) \log n)$ (or $O(t^4 k \log n)$) time using $O(t^4(L + \alpha(n) \log n))$ (or $O(t^4(L + k \log n))$) work on an EREW PRAM, where L is the number of the edges in $SP(u, v)$. This ends the proof of the theorem. ∎

Corollary 3.1 *Let G be an n-vertex weighted digraph of constant treewidth and let $k \geq 1$ be any constant integer. Then, the following hold on an EREW PRAM: (i) After $O(\log^2 n)$ time and $O(nI_k(n))$ work and space preprocessing, distance queries in G can be answered in $O(k)$ time using a single processor and shortest path queries in $O(k \log n)$ time using $O(L + k \log n)$ work, where L is the number of edges of the reported path. (ii) After $O(\log^2 n)$ time and $O(n)$ work and space preprocessing, distance queries in G can be answered in $O(\alpha(n))$ time using a single processor and shortest path queries in $O(\alpha(n) \log n)$ time using $O(L + \alpha(n) \log n)$ work, where L is the number of edges of the reported path.*

In [8] it is shown how the values provided by Lemma 3.2, can be used in the computation of a shortest path tree rooted at a given vertex $s \in V(G)$. But the approach in [8] cannot be

efficiently parallelized in a trivial way, because it is based on a depth-first search of T followed by a (kind of) breadth-first search of G starting at s. Hence, a different method has to be followed which is given in the next theorem.

Theorem 3.3 *For any integer t, let G be an n-vertex weighted digraph of treewidth t, whose tree-decomposition can be found in $T(n,t)$ parallel time using $W(n,t)$ work on an EREW PRAM. Let also $s \in V(G)$. Then, in $O(t \log n + T(n,t))$ time using $O(t^3 n + W(n,t))$ work on an EREW PRAM, we can either compute a shortest path tree rooted at s, or find a negative cycle in G (if exists).*

Proof. Let (X,T) be the tree-decomposition of G. Using Lemma 3.2, we either compute $Dist(u,v)$, for u,v such that $u,v \in X_i$, for some $i \in V(T)$, or find a negative cycle in G. If there is no negative cycle, we can easily compute $\gamma(i)$, $\forall i \in V(T)$. Let $i \in V(T)$ such that $s \in X_i$. Root T at node i and make it balanced, using Fact 2.2. Starting at i, perform a top-down traversal of T by visiting all nodes of T level-by-level. (This can be done by letting each processor associated with some node z of T to fork two other processors and to associate them with the children of z.) At each node $j \in V(T)$ visited, store the product of the γ values on the path from i to j. Since the composition of two γ values can be computed in $O(\log^2 t)$ time using $O(t^3)$ work on an EREW PRAM and each node of T is visited exactly once, the whole process takes $O(\log n \log^2 t)$ time using $O(t^3 n)$ work.

For an edge $\langle v,u \rangle$ of G, define $h(v,u)$ to be the node z of T such that $v,u \in X_z$ and z is the closest such node to the root. (By the continuity condition, $h(v,u)$ is unique.) It is easy to see that during the above top-down traversal of T, we can found such nodes $h(v,u)$ for each edge $\langle v,u \rangle$ in G. We also assume that for each $u \in V(G)$, we have the value $\delta(s,u)$. This is true, since by Lemma 3.1 the value stored at node $j \in V(T)$, $j \neq i$ and $u \in X_j$, during the above mentioned top-down traversal, is $P(X_i, X_j)$ which contains the tuple $(s,u,\delta(s,u))$.

To construct the shortest path tree $\mathcal{T}$, we do the following. Starting at the root node i, we perform a second, level-by-level, top-down traversal of T. For a node $j \in V(T)$ at level $\ell \geq 1$, we check (sequentially) edges $\langle v,u \rangle$, where $v,u \in X_{h(v,u)}$ and v belongs to the shortest path tree $\mathcal{T}^*$ constructed so far, while $u \notin \mathcal{T}^*$. (Initially, $j = i$ and $v = s$.) If $\delta(s,u) = \delta(s,v) + wt(v,u)$, then make v the parent of u in $\mathcal{T}$. If v,u belong also to any child of $X_{h(v,u)}$, then mark the edge $\langle v,u \rangle$ as being "examined" in the local memory of the processor associated with this child. Note that this last operation is needed in order to avoid concurrent access conflicts in the shared memory, in the case where there is another node $k \in V(T)$ at the same level with j for which $v,u \in X_k$.

It can be easily verified (by induction) that the above procedure creates a shortest path tree rooted at s. It is also easy to see that each tree node is visited exactly once and that we need $O(t)$ time (using a single processor) in such a node. Hence, in total, $\mathcal{T}$ can be constructed in $O(t \log n)$ time using $O(t^3 n)$ work. ∎

Corollary 3.2 *Let G be an n-vertex weighted digraph of constant treewidth and let $s \in V(G)$. Then, in $O(\log^2 n)$ time using $O(n)$ work on an EREW PRAM, we can either compute a shortest path tree rooted at s, or find a negative cycle in G (if exists). If the tree-decomposition of G is also provided with the input, then the computation takes $O(\log n)$ time.*

4 The Dynamic Algorithm

In this section we shall give our dynamic data structures and algorithms. The approach follows the one in [8], but the parallel implementation is rather different. The main idea is as follows. We divide the digraph into subgraphs with disjoint edge sets and small cut-sets, and construct 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 have size $O(\sqrt{n})$. We then construct 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 we update the data structure of this subgraph. 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 constant number of updates in subgraphs of size $O(\sqrt{n})$, which yields $O(\sqrt{n})$ update work. By recursively applying this idea, we get an update work of $O(n^\beta)$, for any constant $0 < \beta < 1$.

The parallel algorithms which implement the above approach differ from their sequential counterparts [8] at two points: (a) in the graph equipartitioning results (Section 4.1) which require a different method; and (b) in the analysis of our parallel bounds (Section 4.2) due to the fact that we have to work with balanced tree-decompositions which increase the treewidth of the subgraphs and the reduced digraph. In the following, we first give the graph partitioning results and then give the details of our algorithms.

4.1 Graph Equipartitions

Lemma 4.1 *Let T be a rooted binary tree on n nodes and let $1 \leq m \leq n$. Then, in $O(\log n)$ time and $O(n)$ work on an EREW PRAM, we can partition the nodes of T into at least n/m and at most $8n/m$ groups such that each group: (i) is a connected subtree; (ii) is connected to the rest of the tree through at most 3 edges; and (iii) has at most m nodes.*

Proof. We give an algorithm which is a variant of the well-known parallel tree contraction algorithm (see e.g. [12]). Assign a weight of 1 to each node in the tree. By adding a leaf (with weight 0) as a child to each node that has one child, we obtain a tree in which each node is a

leaf or has two children. Number the leaves of the tree from left to right using the Euler tour technique [12]. From now on assume that we have a tree with weights on the nodes adding up to n, in which each internal node has two children, and in which some of the leaves are numbered from left to right. Our algorithm for obtaining the desired partition performs a number of *rounds*. Each round (consisting of three steps) forms groups of nodes which, at the end, will give the components. The algorithm is as follows:

Repeat the following steps (round) $\lceil \log_{4/3} n \rceil$ times.

1. In parallel, for each odd numbered leaf that is a left child, if the sum of the weights of the leaf, its parent and its sibling is at most m, then shrink the edges connecting the leaf and its sibling to their parent. Assign the parent a weight equal to the sum of the weights of the three nodes. If the sibling is a leaf, it is even numbered. Assign this number to the parent (which is now a leaf in the modified tree). If the sum of the weights exceeds m, then delete the numbers (if they exist) from the leaf and the sibling.

2. Repeat step 1 for each odd numbered leaf that is a right child.

3. After these two steps, all the numbered leaves in the tree have an even number. Divide each of these numbers by 2.

It is not hard to see that after the i-th iteration, at most $l/2^i$ leaves have numbers, where l is the initial number of leaves. Thus, at the end, there are no numbered leaves. Throughout, the following invariant is maintained: if a leaf does not have a number, then the weights of the leaf, its parent and sibling add up to more than m. (Note that such a leaf will not participate in any subsequent iteration.) Call such a triple of leaf, parent and sibling an *overweight* group.

Each non-numbered leaf is contained in some overweight group, and no node can belong to more than two overweight groups. Thus, the sum of the weights of all the overweight groups is at most $2n$, hence the number of overweight groups is at most $2n/m$. Since each overweight group contains at most two non-numbered nodes, the total number of non-numbered leaves at the end is $4n/m$. Since each internal node has two children, the total number of nodes remaining in the tree is at most $8n/m$.

Each node v in the remaining tree is associated with the connected subtree induced by the nodes that were shrunk into v in the above process. These are the required groups. It is easy to see that v has a weight equal to the number of nodes in the associated subtree. Since this weight is at most m, there are at least n/m such connected subtrees. Also, as shown above, there are no more than $8n/m$ connected subtrees. It follows from the construction that each subtree is connected to the rest of the tree through at most 3 edges which are incident on at most 2 nodes of the subtree. ∎

In order to implement the above algorithm – as well as the subsequent ones – on an EREW PRAM, we make the following conventions for the input-output representation.

Input-Output Conventions: We assume that the above algorithm has its input tree specified as a linked structure in n contiguous memory cells. The output it produces is in $O(n)$ contiguous memory cells, divided into contiguous blocks, each block containing one of the connected components in the same linked format, and one final block containing the compressed tree (i.e. the tree at the end of the shrinking process) in a linked format. This can be accomplished using standard EREW PRAM methods in $O(\log n)$ time and $O(n)$ work (see e.g. [12]), which we now describe briefly.

By assigning the preorder number to each node in the compressed tree, we can assign a unique number between 1 and q (where q is the number of nodes in the compressed tree) to each connected subtree. Then, by solving a prefix summation problem on q elements, where the ith element is the number of nodes in subtree i, we can allocate contiguous memory blocks for the various subtrees. It remains to copy the subtrees into the appropriate blocks.

Since each node in the compressed tree knows the memory addresses allocated for its subtree, reversing the shrinking process, we can assign a unique memory address in the appropriate block to each node in a subtree. Now it is a simple matter for each node to copy itself into this address, and duplicate its link structure.

Definition 4.1 *Let λ, δ be positive integer constants and let $1 \leq m \leq n$. Then, given an n-vertex digraph G as well as its balanced tree-decomposition of treewidth t, we define an* (λ, δ, m)-equipartition of G *to be a partition of G into q subgraphs $H_1, \ldots, H_q$, where $n/m \leq q \leq \delta n/m$, along with the construction of another subgraph H' such that: (i) H_i has at most tm vertices and a cut-set $C(H_i)$ of size at most λt; (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 $1 \leq i \leq q$; and (iii) we have a tree-decomposition of treewidth t for each H_i and a tree-decomposition for H' of treewidth $3t$.*

The following lemma shows that an $(3, 8, m)$-equipartition can be efficiently computed.

Lemma 4.2 *Given an n-vertex digraph G along with its balanced tree-decomposition of treewidth t, we can compute an $(3, 8, m)$-equipartition of G in $O(\log n)$ time using $O(t^2 n)$ work on an EREW PRAM, where $1 \leq m \leq n$.*

Proof. Let (X, T) be the balanced tree decomposition of G. By Fact 2.2, T has at most $2n$ nodes. Partition the nodes of T into $n/m \leq q \leq 8n/m$ connected components using Lemma 4.1. For each component T_i, $1 \leq i \leq q$, create a subgraph H_i which is the induced subgraph of G on the vertices in $\cup_{z \in V(T_i)} X_z$. Note that the number of vertices in H_i is at most $t|V(T_i)| = tm$ and T_i is a tree decomposition of H_i. Let z_1, z_2 and z_3 be the nodes through which T_i is connected to the other components. Then, $C(H_i) = X_{z_1} \cup X_{z_2} \cup X_{z_3}$, and $C(H_i)$ has at most $3t$ vertices.

Construct H' by creating a clique on $C(H_i)$, for each $1 \leq i \leq q$. The tree decomposition for H' is constructed by shrinking each component T_i into a single node z and assigning $X_z = C(H_i)$. It is easy to verify that this is a tree decomposition of H' of width $3t$. Also, it is not hard to see that the work required for the above constructions is bounded by $O(t^2 n)$ and the time by $O(\log(tn))$. The EREW PRAM implementation can be easily accomplished using the data structures described after the proof of Lemma 4.1. ∎

4.2 Data Structures and Algorithms

Let $PD(G, \{P_W, P_T\}, \{U_W, U_T\}, Q)$ be a parallel dynamic data structure for a digraph G, where $O(P_W)$ (resp. $O(P_T)$) is the preprocessing work and space (resp. time) to be set up, $O(Q)$ is the time to answer a distance query using a single processor and $O(U_W)$ (resp. $O(U_T)$) is the work (resp. time) to update it after the modification of an edge-weight.

Theorem 4.1 *Assume that we are given an n-vertex weighted digraph G and its balanced tree decomposition of treewidth t. Then, for $r > 0$, we can construct, on an EREW PRAM, the following (with $A = 5t^3 11^{3r}$): (i) $PD(G, \{A^r n, 2^r \log n\}, \{A^r n^{(1/2)^{r-1}}, A^r \log n\}, A^r \alpha(n))$; and (ii) $PD(G, \{A^r n I_k(n), 2^r \log n\}, \{A^r n^{(1/2)^{r-1}}, A^r \log n\}, A^r k)$, for $k \geq 1$.*

Proof. We shall prove part (i). Part (ii) can be proved similarly. We use induction on r. If $r = 1$, then, the work and time allowed for updates exceeds the preprocessing, and the static data structure of Theorem 3.1 suffices, with updates implemented by simply recomputing the whole data structure.

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

First construct an $(3, 8, \sqrt{n})$-equipartition of G using Lemma 4.2, yielding H' and $H_1, \ldots, H_q$, $\sqrt{n} \leq q \leq 8\sqrt{n}$. 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 2.1, it follows that $\delta_{G'}(x, y) = \delta_G(x, y)$, $\forall x, y \in V(G')$. It is easy to verify that the constructions of G_i's and G' can be accomplished within the resource bounds of Lemma 4.2.

Let u, v be the two query vertices. Assume first that $u \in V(G_i)$ and $v \in V(G_j) - V(G_i)$. Then, any path from u to v must pass through a vertex in each of the cut-sets of G_i and G_j. This implies that $\delta_G(u, v) = \min\{\delta_{G_i}(u, x) + \delta_{G'}(x, y) + \delta_{G_j}(y, v) : \ x \in C(G_i), \ y \in C(G_j)\}$. Similarly, if $u, v \in V(G_i)$, then $\delta_G(u, v) = \min\{\delta_{G_i}(u, v), \min\{\delta_{G_i}(u, x) + \delta_{G'}(x, y) + \delta_{G_i}(y, v) : x, y \in C(G_i)\}\}$. It is therefore clear that in order to obtain a query algorithm for any pair $u, v \in V(G)$, it suffices to be able to answer queries of the form $\delta_{G_i}(a, b)$ and $\delta_{G'}(a, b)$, for any pair of vertices a, b.

In the following, let $n_i = |V(G_i)|$ and $n' = |V(G')|$. The $(3, 8, \sqrt{n})$-equipartition of G gives us a tree-decomposition of treewidth t for each subgraph G_i, and a tree-decomposition of treewidth $3t$ for G'. We balance these tree-decompositions, yielding tree-decompositions for each G_i with treewidth at most $3t + 2 \leq 5t$ and for G' with treewidth at most $9t + 2 \leq 11t$. Inductively, we construct in parallel $D(G_i, n_i, r-1, 5t)$, for each $1 \leq i \leq q$, which enables us to answer queries of the form $\delta_{G_i}(a, b)$, and $D(G', n', r-1, 11t)$ which enables us to answer queries of the form $\delta_{G'}(a, b)$.

The update algorithm is as follows. First observe that (by construction) $E(G_i) \cap E(G_j) = \emptyset,\ i \neq j$, 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'. There are two cases to consider.

(i) The weight of an edge belonging to G_i is changed. Then, update the data structure for G_i. This may result in new values for $\delta_{G_i}(x, y),\ x, y \in C(G_i)$. 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 weight of an edge in G_i does not change $\delta_{G_j}(x, y),\ x, y \in C(G_j)$, for $j \neq i$. Thus, after we change, in G', the weight 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 2.1. After the last update, the data structure for G' yields correct distances in G, between vertices in $V(G')$.

(ii) The weight of an edge belonging to G' is changed. Then the distances $\delta_{G_i}(x, y)$ do not change. Thus, in this case, simply update the data structure for G'.

This completes the description of the preprocessing, query and update algorithms.

The time and work required to set up this data structure is the time and work required to construct (1) the equipartitions of G_i's and G', and (2) the data structures of G_i's and G' inductively. By Lemma 4.2, (1) requires $O(\log n)$ time and $O(t^2 n)$ work. Then, writing $PW(r, t)n$ and $PT(r, t)\log n$ for the preprocessing work and time respectively, we have

$$PW(r, t)n \leq t^2 n + \sum_{i=1}^{q} PW(r-1, 5t)n_i + PW(r-1, 11t)n'$$

$$PT(r, t)\log n \leq \log n + \max\{PT(r-1, 11t)\log n', PT(r-1, 5t)\log N\}$$

where $N = \max\{n_1, \ldots, n_q\}$.

Querying involves taking the minimum of the results of the sub-queries specified in the query algorithm previously. Writing $Q(r, t)\alpha(n)$ for the query time, we have

$$Q(r, t)\alpha(n) \leq (5t)^2[2Q(r-1, 5t)\alpha(N) + Q(r-1, 11t)\alpha(n')]$$

During updates, in the worst case there is one update in a graph G_i and then, at most $(5t)^2$ queries in G_i and updates in graph G'. Thus, with $UW(r, t)n^{(1/2)^{r-1}}$ and $UT(r, t)\log n$

representing the work and time respectively, we have

$$\begin{aligned} UW(r,t)n^{(1/2)^{r-1}} &\leq UW(r-1,5t)N^{(1/2)^{r-2}} \\ &\quad +(5t)^2[Q(r-1,5t)\alpha(N)+UW(r-1,11t)(n')^{(1/2)^{r-2}}] \end{aligned}$$

$$UT(r,t)\log n \leq UT(r-1,5t)\log N + (5t)^2[Q(r-1,5t)\alpha(N)+UT(r-1,11t)\log n']$$

It is easy to show that $n' \leq 88tn^{1/2}$, $\sum_{i=1}^{q} n_i \leq 5tn$ and $N = 5tn^{1/2}$. Using these facts and easy estimates, we obtain the following recurrences.

$$\begin{aligned} PW(r,t)n &\leq 2t^2 PW(r-1,11t) \\ PT(r,t) &\leq 2PT(r-1,11t) \\ Q(r,t) &\leq (5t)^3 Q(r-1,11t) \\ UW(r,t) &\leq (5t)^3 UW(r-1,11t) \\ UT(r,t) &\leq (5t)^3 UW(r-1,11t) \end{aligned}$$

from which the claimed bounds follow.

Thus we can construct $D(G,n,r,t)$, completing the induction. ∎

The following theorem shows how to obtain an update work of $O(n^\beta)$, for any constant $0 < \beta < 1$, in a digraph of constant treewidth.

Theorem 4.2 *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 on an EREW PRAM: (i) $PD(G, \{n, \log^2 n\}, \{n^\beta, \log n\}, \alpha(n))$; and (ii) $PD(G, \{nI_k(n), \log^2 n\}, \{n^\beta, \log n\}, k)$.*

Proof. Using Facts 2.1 and 2.2, we can compute a balanced tree-decomposition of G in $O(\log^2 n)$ time and $O(n)$ work on an EREW PRAM. The rest of the proof follows now by Theorem 4.1, if we set $r = 1 - \log\beta$. ∎

The algorithms described above give answers to distance queries only. They can be modified to answer path queries as well, in a way similar to that described in [8]. (The shortest path can be output in the same resource bounds as those stated in Corollary 3.1.) 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 the input digraph G. This can be easily tested in time proportional to that of finding a distance query (see [8]).

References

[1] W. Ackermann, *Zum Hilbertschen Aufbau der reellen Zahlen*, Math. Ann., 99 (1928), pp. 118-133.

[2] R. Ahuja, T. Magnanti and J. Orlin, *Network Flows*, Prentice-Hall, 1993.

[3] N. Alon and B. Schieber, *Optimal Preprocessing for Answering On-line Product Queries*, Tech. Rep. No. 71/87, Tel-Aviv University, 1987.

[4] S. Arnborg, *Efficient Algorithms for Combinatorial Problems on Graphs with Bounded Decomposability - A Survey*, BIT, 25 (1985), pp. 2-23.

[5] H. Bodlaender, *NC-algorithms for Graphs with Small Treewidth*, in Proc. 14th Workshop on Graph-Theoretic Concepts in Comp. Sc. (WG'88), LNCS 344, Springer-Verlag, 1989, pp. 1-10.

[6] H. Bodlaender, *A Tourist Guide through Treewidth*, Acta Cybernetica, 11:1-2 (1993), pp. 1-21, 1993.

[7] H. Bodlaender and T. Hagerup, *Parallel Algorithms with Optimal Speedup for Bounded Treewidth*, in Proc. 22nd Int'l Coll. on Automata, Languages and Programming (ICALP'95), LNCS 944, Springer-Verlag, 1995, pp. 268-279.

[8] S. Chaudhuri and C. Zaroliagis, *Shortest Path Queries in Digraphs of Small Treewidth*, in Proc. 22nd Int'l Coll. on Automata, Languages and Programming (ICALP'95), LNCS 944, Springer-Verlag, 1995, pp. 244-255; full version submitted as *Shortest Paths in Digraphs of Small Treewidth. Part I: Sequential Algorithms.*

[9] E. Cohen, *Efficient Parallel Shortest-paths in Digraphs with a Separator Decomposition*, in Proc. 5th ACM Symp. on Parallel Algorithms and Architectures (SPAA'93), 1993, pp. 57-67.

[10] H. Djidjev, G. Pantziou and C. Zaroliagis, *On-line and Dynamic Algorithms for Shortest Path Problems*, in Proc. 12th Symp. on Theor. Aspects of Comp. Sc. (STACS'95), LNCS 900, Springer-Verlag, 1995, pp. 193-204.

[11] Y. Han, V. Pan and J. Reif, *Efficient Parallel Algorithms for Computing All Pair Shortest Paths in Directed Graphs*, in Proc. 4th ACM Symp. on Parallel Algorithms and Architectures (SPAA'92), 1992, pp. 353-362.

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

[13] D. Kavvadias, G. Pantziou, P. Spirakis and C. Zaroliagis, *Efficient Sequential and Parallel Algorithms for the Negative Cycle Problem*, in Proc. 5th Int'l Symp. on Algorithms and Computation (ISAAC'94), LNCS 834, Springer-Verlag, 1994, pp. 270-278.

[14] P. Klein and S. Subramanian, *A linear-processor polylog-time algorithm for shortest paths in planar graphs*, in Proc. 34th IEEE Symp. on Foundations of Comp. Sc. (FOCS'93), 1993, pp. 259-270.

[15] N. Robertson and P. Seymour, *Graph Minors I: Excluding a Forest*, J. Combinatorial Theory Series B, 35 (1983), pp. 39-61.

[16] N. Robertson and P. Seymour, *Graph Minors II: Algorithmic Aspects of Treewidth*, J. of Algorithms, 7 (1986), pp. 309-322.

[17] B. Schieber and U. Vishkin, *On Finding Lowest Common Ancestors: Simplification and Parallelization*, SIAM J. Computing, 17:6 (1988), pp. 1253-1262.