Optimal Reachability and a Space-Time Tradeoff for Distance Queries in Constant-Treewidth Graphs

Krishnendu Chatterjee[†] Rasmus Ibsen-Jensen[†] Andreas Pavlogiannis[†]

[†] IST Austria

{kchatterjee, ribsen, pavlogiannis}@ist.ac.at

Abstract

We consider data-structures for answering reachability and distance queries on constant-treewidth graphs with n nodes, on the standard RAM computational model with wordsize $W = \Theta(\log n)$. Our first contribution is a data-structure that after $O(n)$ preprocessing time, allows (1) pair reachability queries in $O(1)$ time; and (2) single-source reachability queries in $O(\frac{n}{\log n})$ time. This is (asymptotically) *optimal* and is *faster than DFS/BFS* when answering more than a constant number of single-source queries. The data-structure uses at all times $O(n)$ space. Our second contribution is a space-time tradeoff data-structure for distance queries. For any $\epsilon \in (0, 1]$, we provide a data-structure with $O(n)$ preprocessing time that allows pair queries in $O(n^{1-\epsilon} \cdot \alpha^2(n))$ time, where α is the inverse of the Ackermann function, and at all times uses $O(n^\epsilon)$ working space, where the input graph G only belongs to the input space and does not contribute to the working space.

Keywords: *Graph algorithms; Constant-treewidth graphs; Reachability queries; Distance queries*

1 Introduction

In this work we consider two of the most classic graph algorithmic problems, namely the reachability and distance problems, on low-treewidth graphs. We consider the case where the input is a graph G with n nodes and a tree-decomposition $\mathrm{Tree}(G)$ of G with $b = O(n)$ bags and width t. The computational model is the standard RAM with wordsize $W = \Theta(\log n)$.

Low-treewidth graphs. A very well-known concept in graph theory is the notion of *treewidth* of a graph, which is a measure of how similar a graph is to a tree (a graph has treewidth 1 precisely if it is a tree) [34]. The treewidth of a graph is defined based on a *tree decomposition* of the graph [27], see Section 2 for a formal definition. Beyond the mathematical elegance of the treewidth property for graphs, there are many classes of graphs which arise in practice and have low (even constant) treewidth. An important example is that the control flow graph for goto-free programs for many programming languages are of constant treewidth [37]. Also many chemical compounds have treewidth 3 [39]. For many other applications see the surveys [11, 14]. Given a tree decomposition of a graph with low treewidth t, many problems on the graph become complexity-wise easier (i.e., many NP-complete problems for arbitrary graphs can be solved in time polynomial in the size of the graph, but exponential in t, given a tree decomposition [4, 8, 10]). Even for problems that can be solved in polynomial time, faster algorithms can be obtained for low-treewidth graphs, for example, for the distance (or the shortest path) problem [18]. The constant treewidth of control flow graphs has also been shown to lead to faster algorithms for interprocedural analysis [16], quantitative verification [17], and analysis of concurrent programs [15].

Reachability/distance problems. The *pair* reachability (resp., distance) problem is one of the most classic graph algorithmic problems that, given a pair of nodes u, v, asks to compute if there is a path from u to v (resp., the weight of the shortest path from u to v). The *single-source* variant problem given a node u asks to solve the pair problem u, v

for every node v. Finally, the *all pairs* variant asks to solve the pair problem for each pair u, v. While there exist many classic algorithms for the distance problem, such as A^*-algorithm (pair) [29], Dijkstra's algorithm (single-source) [21], Bellman-Ford algorithm (single-source) [6, 25, 32], Floyd-Warshall algorithm (all pairs) [24, 38, 35], and Johnson's algorithm (all pairs) [30] and others for various special cases, there exist in essence only two different algorithmic ideas for reachability: Fast matrix multiplication (all pairs) [23] and DFS/BFS (single-source) [20].

Previous results. The algorithmic question of the distance (pair, single-source, all pairs) problem for low-treewidth graphs has been considered extensively in the literature, and many data-structures have been presented [2, 18, 33, 1, 5, 19]. The previous results are incomparable, in the sense that the best data-structure depends on the treewidth and the number of queries. The pair query reachability for low-treewidth graphs has been considered in [40]. Despite many results for constant (or low) treewidth graphs, none of them improves the complexity for the basic single-source reachability problem, i.e., the bound for DFS/BFS has not been improved in any of the previous works.

Our results. Our algorithms take as input a graph G with n nodes, and a tree decomposition of width t with $O(n)$ bags. Our main contributions are as follows (summarized in Table 1 and Table 2):

1. Our first contribution is a data-structure that supports reachability queries in G. The computational complexity we achieve is as follows: (i) $O(n \cdot t^2)$ preprocessing (construction) time; (ii) $O(n \cdot t)$ space; (iii) $O(\lceil t/\log n \rceil)$ pair-query time; and (iv) $O(n \cdot t/\log n)$ time for single-source queries. Note that for constant-treewidth graphs, the data-structure is *optimal* in the sense that it only uses linear preprocessing time, and supports answering queries in the size of the output (the output for single-source queries requires one bit per node, and thus has size $\Theta(n/W) = \Theta(n/\log n)$). Moreover, also for constant-treewidth graphs, the data-structure answers single-source queries *faster* than DFS/BFS, after linear preprocessing time (which is asymptotically the same as for DFS/BFS). Thus there exists a constant c_0 such that the total of the preprocessing and querying time of the data-structure is smaller than that of DFS/BFS for answering at least c_0 single-source queries. To the best of our knowledge, this is the first data-structure which is *faster than DFS/BFS* for solving single-source reachability on an important class of sparse graphs. While our data-structure achieves this using so-called word-tricks, DFS/BFS have not been made faster using word-tricks.
2. Second, we present a space-time tradeoff data-structure that supports distance queries in G and given a number ϵ in $(0, 1]$. The weights of G come from the set of integers $\mathbb{Z}$, but we do not allow negative cycles. The computational complexity we achieve is as follows: (i) $O(n \cdot t^2)$ preprocessing (construction) time; (ii) $O(n^\epsilon \cdot t^2)$ working space; and (iii) $O(n^{1-\epsilon} \cdot t^2 \cdot \alpha^2(n))$ time for pair queries, where α is the inverse Ackermann function. The above data-structure considers that a tree decomposition of G is part of the input, and does not contribute to the space complexity. Instead, if only the graph G is given as input, we present a data-structure for constant-treewidth graphs that for any $\epsilon \in [1/2, 1]$ operates in (i) polynomial preprocessing time; (ii) $O(n^\epsilon)$ working space; and (iii) $O(n^{1-\epsilon} \cdot \alpha^2(n))$ time for pair queries.

Table 1: Data-structures for pair and single-source reachability queries, on a directed graph G with n nodes, and a tree decomposition of width t. The model of computation is the standard RAM model with wordsize $W = \Theta(\log n)$. Rows 1 and 2 are previous results, and row i is the result of this paper.

Row	Preprocessing time	Space usage	Pair query time	Single-source query time	From
1	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(\log n)$	$O(n \cdot \log n)$ [a]	[40] [b]
2	–	$O(\lceil n/\log n \rceil)$	–	$O(n \cdot t) = O(m)$	DFS/BFS [20]
i	$O(n \cdot t^2)$	$O(n \cdot t)$	$O(\lceil \frac{t}{\log n} \rceil)$	$O(\frac{n \cdot t}{\log n})$	Theorem 2

[a] Obtained by multiplying the time for a pair query by n.
[b] The result is only stated for constant treewidth.

Technical contributions. Our results rely on three key technical contributions:

1. A core component in both of our results is computing local distances (or reachability) in the bags of a tree-decomposition, namely, for every pair of nodes u, v that appear in a bag, compute the distance from u to v, (or whether v is reachable from u). This computation has been considered before, e.g. in [2, 18, 31]. The existing

Table 2: Data-structures for pair and single-source distance queries, on a weighted directed graph G with n nodes, m edges, and a tree decomposition of width t, height h and b bags (and $b \geq n/t$). The number ϵ can be any fixed number in $(0, 1]$. We use $\widetilde{O}$ to hide polynomial factors of $\alpha(n)$, which is the inverse Ackermann function. When measuring space complexity, we do not count the input size. Rows 1-6 are previous results, and row i is the result of this paper.

Row	Preprocessing time	Space usage	Pair query time	Single-source query time	From
1	$O(n^2 \cdot t)$	$O(n^2)$	$O(1)$	$O(n)$	[33] [a]
2	$O(n \cdot t^3)$	$O(n \cdot t^3)$	$\widetilde{O}(t^3)$	$O(n \cdot t^3)$	[18]
3	$O(n \cdot t^3 \cdot \log h)$	$O(b \cdot t^2)$	$O(t^2 \cdot \log\log n)$	$O(n \cdot t^2 \cdot \log\log n)$ [b]	[2]
4	$O(n \cdot t^2 \cdot \log^2 n)$	$O(n \cdot t \cdot \log n)$	$O(t \cdot \log n)$	$O(n \cdot t \cdot \log n)$[b]	[1]
5	$O(n \cdot t \cdot \log n)$	$O(n \cdot t \cdot \log n)$	$O(t^2 \cdot \log^2 n)$	$O(n \cdot t^2 \cdot \log^2 n)$ [b]	[5, 19]
6	Not given	$O(n^\epsilon \cdot t^2 \cdot \log^2 n)$ [c]	$O(n^{1-\epsilon} \cdot t \cdot \log n)$	– [d]	[2] [e]
i	$O(n \cdot t^2)$	$O(n^\epsilon \cdot t^2)$	$\widetilde{O}(n^{1-\epsilon} \cdot t^2)$	– [d]	Theorem 5

[a] This data-structure solves the all pairs problem in the given time and space bounds.
[b] Obtained by multiplying the time for a pair query by n.
[c] This is the space usage after preprocessing.
[d] Not given since the size of the output is larger than the data-structure.
[e] Note that [2] does not explicitly state the tradeoff given (they only state linear space), but it follows from their technique by picking other values for their variable k.

solutions are "bag-centric", as they rely on applying an all-pairs (transitive closure) computation in the bags of the tree-decomposition. We develop a new algorithm for this problem which is "node-centric", and saves a factor t in the time and space complexity compared to the existing methods. The correctness is based on a newly introduced notion of "U-shaped paths", which may be of general interest.

2. For pair reachability queries, the key idea is to store reachability information from each node to $O(\log n)$ other nodes. For single-source queries, for some nodes this reachability information might be of size $\Theta(n)$, but on average remains $O(\log n)$. In addition, our data-structure computes reachability information in such a way that allows for compact representation and fast retrieval using word tricks, which for constant-treewidth graphs leads to asymptotically optimal preprocessing and query (both pair and single-source) bounds. The idea of storing $O(\log n)$ information per node has appeared before ([40, 18]) however those algorithms follow different approaches, where word tricks do not seem to be applicable (at least not without significantly modifying the algorithms).
3. For distance queries, we also introduce the notion of summary trees, which lead to a family of preprocessing vs query time tradeoffs. The latter is a rather technical result, and is presented only in the full version, which is attached in the Appendix. Our space efficient data-structure uses the techniques of U-shaped paths and summary trees together with a space efficient way to partition a tree decomposition into components, which leads to the stated space-time tradeoff.

2 Preliminaries

Graphs. We consider weighted directed graphs $G = (V, E, \mathsf{wt})$ where V is a set of n nodes, $E \subseteq V \times V$ is an edge relation of m edges, and $\mathsf{wt} : E \to \mathbb{Z}$ is a weight function where $\mathbb{Z}$ is the set of integers. In the sequel we write graphs for directed graphs, and explicitly mention if the graph is undirected. Given a set $X \subseteq V$, we denote by $G[X]$ the subgraph $(X, E \cap (X \times X))$ of G induced by the set of nodes X. A path $P : u \rightsquigarrow v$ is a sequence of nodes $(x_1, \ldots, x_k)$ such that $u = x_1$, $v = x_k$, and for all $1 \leq i \leq k-1$ we have $(x_i, x_{i+1}) \in E$. The path P is *simple* if every node appears at most once in P. The length of P is $k-1$, and a single node is by itself a 0-length path. We denote by $E^* \subseteq V \times V$ the transitive closure of E, i.e., $(u, v) \in E^*$ iff there exists a path $P : u \rightsquigarrow v$. Given a path P, a node u, and a set of nodes A, we use the set notation $u \in P$ to denote that u appears in P, and $A \cap P$ to refer to the set of nodes that appear in both P and A. The weight function is extended to paths, and the weight of a path

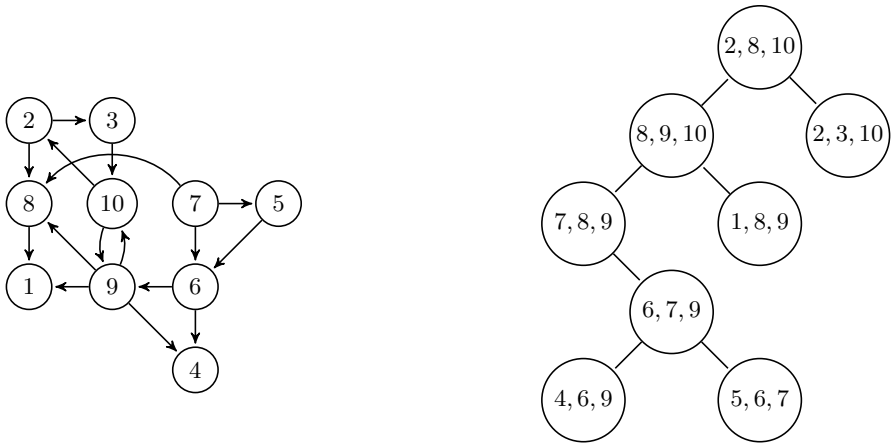


Figure 1: A graph G with treewidth 2 (left) and a corresponding tree-decomposition $\mathrm{Tree}(G)$ (right).

$P = (x_1, \ldots, x_k)$ is $\mathsf{wt}(P) = \sum_{i=1}^{k-1} \mathsf{wt}(x_i, x_{i+1})$ if $k > 1$, else $\mathsf{wt}(P) = 0$. For $u, v \in V$, the distance from u to v is defined as $d(u, v) = \min_{P: u \rightsquigarrow v} \mathsf{wt}(P)$, where P ranges over simple paths in G (and $d(u, v) = \infty$ if no such path exists). We consider that G does not have negative cycles.

Trees. A (rooted) tree $T = (V_T, E_T)$ is an undirected graph with a distinguished node r which is the root such that there is a unique simple path $P_u^v : u \rightsquigarrow v$ for each pair of nodes u, v. The *size* of T is $|V_T|$. Given a tree T with root r, the *level* $\mathsf{Lv}(u)$ of a node u is the length of the simple path P_u^r from u to the root r, and every node in P_u^r is an *ancestor* of u. If v is an ancestor of u, then u is a *descendant* of v. Note that a node u is both an ancestor and descendant of itself. For a pair of nodes $u, v \in V_T$, the *lowest common ancestor (LCA)* of u and v is the common ancestor of u and v with the largest level. The *parent* u of v is the unique ancestor of v in level $\mathsf{Lv}(v) - 1$, and v is a *child* of u. A *leaf* of T is a node with no children. For a node $u \in V_T$, we denote by $T(u)$ the subtree of T rooted in u (i.e., the tree consisting of all descendants of u). The tree T is *binary* if every node has at most two children. The *height* of T is $\max_u \mathsf{Lv}(u)$ (i.e., it is the length of the longest path P_u^r), and T is *balanced* if its height is $O(\log |V_T|)$. Given a tree T, a *connected component* $\mathcal{C} \subseteq V_T$ of T is a set of nodes of T such that for every pair of nodes $u, v \in \mathcal{C}$, the unique simple path P_u^v in T visits only nodes in $\mathcal{C}$.

Tree decompositions. Given a graph G, a tree-decomposition $\mathrm{Tree}(G) = (V_T, E_T)$ is a tree with the following properties.

T1: $V_T = \{B_1, \ldots, B_b : \text{ for all } 1 \leq i \leq b.\ B_i \subseteq V\}$ and $\bigcup_{B_i \in V_T} B_i = V$.
T2: For all $(u, v) \in E$ there exists $B_i \in V_T$ such that $u, v \in B_i$.
T3: For all B_i, B_j and any bag B_k that appears in the simple path $B_i \rightsquigarrow B_j$ in $\mathrm{Tree}(G)$, we have $B_i \cap B_j \subseteq B_k$.

The sets B_i which are nodes in V_T are called *bags*. The *width* of a tree-decomposition $\mathrm{Tree}(G)$ is the size of the largest bag minus 1, and the *treewidth* of G is the width of a minimum-width tree decomposition of G. Let G be a graph, $T = \mathrm{Tree}(G)$, and B_0 be the root of T. For $u \in V$, we say that a bag B is the *root bag* of u if B is the bag with the smallest level among all bags that contain u. By definition, for every node u there exists a unique bag which is the root of u. We often write B_u for the root bag of u, i.e., $B_u = \arg\min_{B_i \in V_T:\ u \in B_i} \mathsf{Lv}(B_i)$, and denote by $\mathsf{Lv}(u) = \mathsf{Lv}(B_u)$. A bag B is said to *introduce* a node $u \in B$ if either B is a leaf, or u does not appear in any child of B. In this work we consider only *binary* tree decompositions (if not, a tree decomposition can be made binary by a standard process that increases its size by a constant factor while keeping the width the same).

See Figure 1 for an example of a graph and a tree-decomposition of it. The following lemma states a well-known "separator property" of tree decompositions.

Lemma 1. *Consider a graph $G = (V, E)$, a binary tree-decomposition $T = \mathrm{Tree}(G)$, and a bag B of T. Let $(\mathcal{C}_i)_{1 \leq i \leq 3}$ be the components of T created by removing B from T, and let V_i be the set of nodes that appear in bags*

of component $\mathcal{C}_i$. For every $i \neq j$, nodes $u \in V_i$, $v \in V_j$ and path $P : u \rightsquigarrow v$, we have that $P \cap B \neq \emptyset$ (i.e., all paths between u and v go through some node in B).

Proof. Examine any such path P. If there exists an $x \in P \cap V_i \cap V_j$, then there exists bags $B_i \in \mathcal{C}_i$ and $B_j \in \mathcal{C}_j$ with $x \in B_i \cap B_j$. Since $B \in B_i \rightsquigarrow B_j$, by property T3 we have $x \in B$. Otherwise there exists a last node y_i of P with $y_i \in V_i$, and if y_j is the node of P following y_i, we have $(y_i, y_j) \in E$. By property T2, $(y_i, y_j) \in B'$ for some bag B', and by the choice of y_i, y_j, it can only be $B' = B$, hence $y_i, y_j \in B$. □

Using Lemma 1, we prove the following stronger version of the separator property, which will be useful throughout the paper.

Lemma 2. *Consider a graph $G = (V, E)$ and a tree-decomposition $\mathrm{Tree}(G)$. Let $u, v \in V$, and consider two distinct bags B_1 and B_j such that $u \in B_1$ and $v \in B_j$. Let $P' : B_1, B_2, \ldots, B_j$ be the unique simple path in T from B_1 to B_j. For each $i \in \{2, \ldots, j\}$ and for each path $P : u \rightsquigarrow v$, there exists a node $x_i \in (B_{i-1} \cap B_i \cap P)$.*

Proof. Let $T = \mathrm{Tree}(G)$. Fix a number $i \in \{2, \ldots, j\}$. We argue that for each path $P : u \rightsquigarrow v$, there exists a node $x_i \in (B_{i-1} \cap B_i \cap P)$. We construct a tree T', which is similar to T except that instead of having an edge between bag B_{i-1} and bag B_i, there is a new bag B, that contains the nodes in $B_{i-1} \cap B_i$, and there is an edge between B_{i-1} and B and one between B and B_i. It is easy to see that T' satisfies the properties T1-T3 of a tree-decomposition of G. By Lemma 1, each bag B' in the unique path $P'' : B_1, \ldots, B_{i-1}, B, B_i, \ldots, B_j$ in T' separates u from v in G. Hence, each path $u \rightsquigarrow v$ must go through some node in B, and the result follows. □

The following lemma states that for nodes that appear in bags B, B' of the tree-decomposition $T = \mathrm{Tree}(G)$, their distance can be written as a sum of distances $d(x_i, x_{i+1})$ between pairs of nodes (x_i, x_{i+1}) that appear in bags B_i that constitute the unique $B \rightsquigarrow B'$ path in T.

Lemma 3. *Consider a weighted graph $G = (V, E, \mathsf{wt})$ and a tree-decomposition $\mathrm{Tree}(G)$. Let $u, v \in V$, and $P' : B_1, B_2, \ldots, B_j$ be a simple path in T such that $u \in B_1$ and $v \in B_j$. Let $A = \{u\} \times \left(\prod_{1<i\leq j} (B_{i-1} \cap B_i)\right) \times \{v\}$. Then $d(u, v) = \min_{(x_1,\ldots,x_{j+1}) \in A} \sum_{i=1}^{j} d(x_i, x_{i+1})$.*

Proof. Consider a witness path $P : u \rightsquigarrow v$ such that $\mathsf{wt}(P) = d(u, v)$. By Lemma 2, there exists some node $x_i \in (B_{i-1} \cap B_i \cap P)$, for each $i \in \{1, \ldots, j\}$. It easily follows that $d(u, v) = \sum_{i=1}^{j} d(x_i, x_{i+1})$ with $x_1, \ldots x_{j+1} \in A$. □

Nice tree decompositions. A tree-decomposition $T = \mathrm{Tree}(G)$ is called *nice* if every bag B is one of the following types:

Leaf. $|B| = 1$.
Forget. B has exactly one child B', and $B \subset B'$ and $|B| = |B'| - 1$.
Introduce. B has exactly one child B', and $B' \subset B$ and $|B'| = |B| - 1$.
Join. B has exactly two children B_1, B_2, and $B = B_1 = B_2$.

For technical convenience, we also require the root of a nice tree decomposition to have size 1. Thus in a nice tree decomposition every bag is the root bag of at most one node.

Model and word tricks. We consider the standard RAM model with word size $W = \Theta(\log n)$, where $\mathsf{poly}(n)$ is the size of the input. Our reachability algorithm (in Section 4) uses so called "word tricks" heavily. We use constant-time lowest common ancestor queries which also require word tricks [7].

Iterated logarithms.

For $\lambda \in \mathbb{N}$, we use the λ-iterated logarithm, defined as:

$$\log^{(\lambda)*} n = \begin{cases} 0 & \text{if } x \leq 1 \\ 1 + \log^{(\lambda)*}\left(\log^{(\lambda-1)*} n\right) & \text{if } x > 1 \end{cases}$$

where $\log^{(0)*} n = \log n$ and $\log^{(1)*} n = \log^* n$. Furthermore, we use the inverse of the Ackermann function [3]. The Ackermann function is:

$$A(i,j) = \begin{cases} 2j & \text{if } i = 0 \text{ and } j \geq 0 \\ 1 & \text{if } i \geq 1 \text{ and } j = 0 \\ A(i-1, A(i, j-1)) & \text{if } i, j \geq 1 \end{cases}$$

The inverse Ackermann function is $\alpha(n) = \mathsf{argmin}_j(A(j,j) \geq n)$. We have that $\log^{(\lambda)*} n = \mathsf{argmin}_j(A(\lambda+1, j) \geq n)$, and hence $\log^{(\alpha(n)-1)*} n = \mathsf{argmin}_j(A(\alpha(n), j) \geq n) \leq \alpha(n)$.

Graph queries. In this work we consider the following queries on the graph G.

1. Given nodes $u, v \in V$, the *pair reachability query* returns true iff $(u, v) \in E^*$.
2. Given a node $u \in V$, the *single-source reachability query* returns the set $\{v : (u, v) \in E^*\}$ of nodes reachable from u.
3. Given nodes $u, v \in V$, the *pair distance query* returns the distance $d(u, v)$.
4. Given a node $u \in V$, the *single-source distance query* returns the distance $d(u, v)$ from u to v, for all v such that $(u, v) \in E^*$.

3 Local Distance Computation

Consider a graph $G = (V, E)$, with a tree-decomposition $\mathrm{Tree}(G) = (V_T, E_T)$ of $|V_T| = O(n)$ bags and width t. In this section we present an algorithm for computing "local distances" in each bag, namely, the distance $d(u, v)$ between every pair of nodes u, v that appear in some bag B. The result of this section is used in later sections for answering reachability (which is a special case of distance) and distance queries on G.

Local distances. Given a tree-decomposition $T = \mathrm{Tree}(G)$ of a graph G and a node $u \in V$, we define the *local forward* and *backward maps* $\textsc{Fwd}_u, \textsc{Bwd}_u : B_u \to \mathbb{Z} \cup \{\infty\}$ of u as

$$\textsc{Fwd}_u(v) = d(u, v); \quad \textsc{Bwd}_u(v) = d(v, u)$$

i.e., $\textsc{Fwd}_u$ (resp., $\textsc{Bwd}_u$) stores the forward (resp., backward) distances to and from nodes that appear in the root bag of u. For any bag $B \in V_T$, we define the *local distance map* of B as

$$\mathsf{LD}_B : B \times B \to \mathbb{Z} \cup \{\infty\} \text{ such that for all } u, v \in B, \text{ we have } \mathsf{LD}_B(u, v) = d(u, v)$$

Note that actually storing all LD_B explicitly requires $\Omega(n \cdot t^2)$ space. The following lemma implies that the distance between any two nodes that appear in some bag is captured in the local forward and backward maps. Hence the local distance maps LD_B are stored implicitly in the forward and backward maps.

Lemma 4. *For every $B \in V_T$ and $u, v \in B$, we have $u \in B_v$ iff $\mathsf{Lv}(v) \geq \mathsf{Lv}(u)$.*

Proof. Consider some $(u, v) \in E$, such that $\mathsf{Lv}(v) \geq \mathsf{Lv}(u)$. By the definition of tree decomposition, there exists some $B_i \in V_T$ such that $u, v \in B_i$. Then u appears in all bags B_j in the simple path $P : B_i \rightsquigarrow B_u$, and since $\mathsf{Lv}(v) \geq \mathsf{Lv}(u)$, the bag B_v appears in P. Hence $u \in B_v$. □

The task of this section is to compute the local forward and backward maps of each node fast. We establish the following theorem.

Theorem 1. *Given a weighted graph $G = (V, E, \mathsf{wt})$ and a tree-decomposition $\mathrm{Tree}(G)$ of G of width t and $O(n)$ bags, the local forward and backward maps of all nodes can be computed in total $O(n \cdot t^2)$ time and $O(n \cdot t)$ space.*

Ordered set and map data-structures. We fix a total order on the vertices V of G. A set $A \subseteq V$ is represented as an *ordered set* by enumerating the elements of A in increasing order. For $A \subseteq V$, a map $M : A \to \mathbb{Z} \cup \{\infty\}$ is represented as an *ordered map* by a pair of lists (L_1, L_2), where L_1 is an ordered set representation of A, and L_2 such that in position i it contains the image under M of the element of L_1 in position i. Intersecting two ordered sets requires linear time in the size of the sets. Given two maps $M_1 : A_1 \to \mathbb{Z} \cup \{\infty\}$, $M_2 : A_2 \to \mathbb{Z} \cup \{\infty\}$ represented as ordered maps, computing the map $M : A_1 \cap A_2 \to \mathbb{Z} \cup \{\infty\}$ with $M(u) = \min(M_1(u), M_2(u))$ also requires linear time in the size of M_1, M_2.

We assume that the bags of $\mathrm{Tree}(G)$ are stored using the ordered set data-structure (if not, this can be done in $O(n \cdot t \cdot \log t)$ time). Clearly all forward and backward maps can be stored as ordered maps in $O(n \cdot t)$ space (using $O(t)$ space per map). Given the forward and backward maps as ordered maps, the local distance map LD_B for any bag B can be constructed in $O(t^2)$ time, by examining all $O(t)$ maps $\textsc{Fwd}_u$, $\textsc{Bwd}_u$ for each $u \in B$. In the following we present algorithm LocDis for computing the maps $\textsc{Fwd}_u$ and $\textsc{Bwd}_u$ of each node u.

Subsuming tree-decomposition. Given a nice tree-decomposition $T' = (V'_T, E'_T)$ and a tree-decomposition $T = (V_T, E_T)$ of a graph G, we say that T' *subsumes* T if the following conditions hold.

1. For every $B' \in V'_T$ there exists $B \in V_T$ such that $B' \subseteq B$.
2. For every $B \in V_T$, there exists $B' \in V'_T$ with $B = B'$.

We present our algorithm LocDis for the local distance computation on a nice tree decomposition. In order to apply LocDis on any tree-decomposition T, we first construct a nice tree-decomposition T' that subsumes T, and then execute LocDis on T'. It is straightforward to verify that for any bag B of T, we have $\mathsf{LD}_B = \mathsf{LD}_{B'}$ where B' is a bag of T' such that $B = B'$. Here we describe a slightly technical construction of such a T' such that T' uses asymptotically the same space as T.

Lemma 5. *For every tree-decomposition $T = (V_T, E_T)$ with b bags and width t there exists a nice tree-decomposition $T' = (V'_T, E'_T)$ of $O(b \cdot t)$ bags and width t that subsumes T. Moreover, T' uses $O(b \cdot t)$ space and can be constructed in $O(b \cdot t \cdot \log t)$ time.*

Proof. Let $B_1, \ldots B_b$ be the bags of V_T. We present an informal outline of the construction. Along the construction, we build a map $f : V'_T \to \{1, \ldots, b\}$. First, create T' identical to T, and for each $B_i \in V'_T$, sort the nodes of B_i in some order, and let $f(B) = i$. Then, as long as one of the following cases holds, proceed accordingly.

1. If there exists a $B \in V'_T$ with two children B^1, B^2 such that $B \neq B^1$ or $B \neq B^2$, insert bags $\overline{B}^1$ and $\overline{B}^2$ in V'_T such that $\overline{B}^1 = \overline{B}^2 = B$. Make each $\overline{B}^i$ a child of B, and parent of B^i. Set $f(\overline{B}^i) = f(B)$.
2. If there exists a $B \in V'_T$ which is the root bag of $k > 1$ nodes $u_1, \ldots, u_k$, insert a line of $k-1$ bags $B^1, \ldots, B^{k-1}$, where B^i is the parent of B^{i+1}, and $B^i = B \setminus \{u_{i+1}, \ldots, u_k\}$. Make B^{k-1} the parent of B and B^1 a child of the parent of B in T, and set $f(B^i) = f(B)$ for all i.
3. If there exists a $B \in V'_T$ which introduces $k > 1$ nodes $u_1, \ldots, u_k$, insert a line of $k-1$ bags $B^1, \ldots, B^{k-1}$, where B^i is the child of B^{i+1}, and $B^i = B \setminus \{u_{i+1}, \ldots, u_k\}$. Make B^{k-1} the unique child of B, and make B^1 the parent of all children of B in T, and set $f(B^i) = f(B)$ for all i.

Finally, in the above construction each $B \in V'_T$ is not stored explicitly as a set, but implicitly as a pointer $f(B)$ to a bag $B_{f(B)}$ of T, and (optionally) two integers i_B, j_B. A node $u \in B_{f(B)}$ is considered to belong to B if one of the following holds.

1. $B_{f(B)}$ is not the root bag of u, and u is not introduced in $B_{f(B)}$.
2. $B_{f(B)}$ is the root bag of u and u is the i-th node with root bag $B_{f(B)}$ and $i \leq i_B$.
3. u is the jth node introduced in $B_{f(B)}$ and $j \leq j_B$.

It follows from the definition of tree-decompositions that if none of the above three cases holds, T' is a nice tree-decomposition that subsumes T. The construction requires $O(b \cdot t \cdot \log t)$ time to sort the nodes in each bag of T, and $O(b \cdot t)$ time to construct the $O(b \cdot t)$ bags of T'. The space used is $O(b \cdot t)$ for storing the original T, plus $O(b \cdot t)$ for storing a pointer and index in each bag of T'. □

Remark 1. It is known that every graph with treewidth t has a nice tree decomposition of $O(n \cdot t)$ bags and width t [13, Lemma 7] . We note that Lemma 5 is different, in that T' is made to subsume some other tree-decomposition T. This has the advantage that the local distance maps of T' carry over to T, while T has some other desired properties (in later sections we require that T is balanced).

Algorithm LocDis**.** Given a graph G and a tree-decomposition $T = \mathrm{Tree}(G)$ with $O(n)$ bags and width t, we present an algorithm LocDis for computing the local forward and backward maps. First, construct a nice tree-decomposition $T' = (V'_T, E'_T)$ of $O(n \cdot t)$ bags which subsumes T, using the construction of Lemma 5. The rest of the computation is then performed as a two-way pass on T', which has the property that every bag is the root bag of at most one node x. For each node $u \in V$ maintain two ordered maps $\textsc{Fwd}'_u, \textsc{Bwd}'_u : B_u \to \mathbb{Z} \cup \{\infty\}$. For notational convenience, we think of the maps $\textsc{Fwd}'_u$, $\textsc{Bwd}'_u$ as variables. Initially set $\textsc{Fwd}'_u(v) = \mathsf{wt}(u, v)$ and $\textsc{Bwd}'_u(v) = \mathsf{wt}(v, u)$ for all $u \in V$ (if $(u, v) \notin E$ or $(v, u) \notin E$, then the corresponding entry is ∞). Given any bag B and nodes $u, v \in B$, we write $\mathsf{LD}'_B(u, v) = w$ if either $\textsc{Fwd}'_u(v) = w$ or $\textsc{Bwd}'_v(u) = w$. Since T' is nice, $\mathsf{LD}'_B(u, v)$ is well-defined. Note that as the maps $\textsc{Fwd}'_u$ and $\textsc{Bwd}'_u$ are modified by the algorithm, the maps LD'_B are modified accordingly.

1. *First pass.* Traverse T' level by level starting from the leaves (bottom-up), and for each encountered bag B_x that is the root bag of node x, execute the following steps: For every pair of nodes $u, v \in B_x$, let $z = \min(\mathsf{LD}'_{B_x}(u, v), \mathsf{LD}'_{B_x}(u, x) + \mathsf{LD}'_{B_x}(x, v))$. If $\mathsf{Lv}(u) \geq \mathsf{Lv}(v)$, then assign $\textsc{Fwd}'_u(v) = z$, otherwise assign $\textsc{Bwd}'_v(u) = z$.
2. *Second pass.* Traverse T' level by level starting from the root (top-down), and for each encountered bag B_x that is the root bag of node x, execute the following steps: For every pair of nodes $u, v \in B_x$, let $z_1 = \min(\mathsf{LD}'_{B_x}(x, v), \mathsf{LD}'_{B_x}(x, u) + \mathsf{LD}'_{B_x}(u, v))$, and $z_2 = \min(\mathsf{LD}'_{B_x}(v, x), \mathsf{LD}'_{B_x}(u, x) + \mathsf{LD}'_{B_x}(v, u))$. Assign $\textsc{Fwd}'_x(v) = z_1$ and $\textsc{Bwd}'_x(v) = z_2$.

In the following we establish that at the end of the second pass it holds that $\textsc{Fwd}'_u = \textsc{Fwd}_u$ and $\textsc{Bwd}'_u = \textsc{Bwd}_u$ for each $u \in V$. We rely on the property that T' is nice only for the following facts: (i) every bag is the root bag of at most one node, and (ii) the leaves and the root of T' have size 1. We say that a path $P : x_1, \ldots, x_k$, is U-shaped in a bag B if $x_1, x_k \in B$ and either $k = 2$, or for every $1 < i < k$, the root bag B_{x_i} of node x_i is in $T(B)$. The following lemma captures a property of U-shaped paths which is used for obtaining Theorem 1.

Lemma 6. *Given a bag B and nodes $u, v \in B$ such that exists a simple path (or simple cycle) $P : u \rightsquigarrow v$ which is* U*-shaped in B, either $|P| = 1$ or $P = (u, y_1, \ldots, y_k, v)$ and for $x = \arg\min_i \mathsf{Lv}(y_i)$, we have that B_x is a descendant of B and P is* U*-shaped in B_x.*

Proof. Decompose P to $P_1 : u \rightsquigarrow x$ and $P_2 : x \rightsquigarrow v$, and we first argue that each P_i is U-shaped in B_x. We only focus on P_1, as the proof is similar for P_2. For any intermediate node y of P_1, the LCA L of B_x and B_y is B_x, otherwise it follows from Lemma 1 that P_1 would go through some node of L that has smaller level than $\mathsf{Lv}(x)$, contradicting our choice of x. Hence, every B_y of intermediate nodes y of P_1 is contained in $T(B_x)$, and it remains to show that $u \in B_x$. Since P is U-shaped in B, we have that B_x is a descendant of B. If $B = B_x$ we are done, otherwise let B' be the parent of B_x. By Lemma 2, there is a node $y \in B' \cap B_x \cap P_1$, and it follows that $\mathsf{Lv}(y) < \mathsf{Lv}(x)$. The only such node in P_1 is u, thus P_1 is U-shaped in B_x. The same argument holds for P_2, and since P is simple, it follows that P is U-shaped in B_x. □

Lemma 7. *At the end of* LocDis*, for each node $u \in V$, we have $\textsc{Fwd}'(u) = \textsc{Fwd}(u)$ and $\textsc{Bwd}'(u) = \textsc{Bwd}(u)$.*

Proof. It is clear that for all nodes u and v, the map $\textsc{Fwd}'_u(v)$ (resp. $\textsc{Bwd}'_u(v)$) always stores the weight of a $u \rightsquigarrow v$ (resp. $v \rightsquigarrow u$) path. The proof focuses on showing that $z = \min_P \mathsf{wt}(P)$, where P is a simple $u \rightsquigarrow v$ (resp. $v \rightsquigarrow u$) path.

We first claim that after the first pass processes a bag B, for all $u, v \in B$ we have $\mathsf{LD}'_B(u, v) \leq \min_P \mathsf{wt}(P)$, where P ranges over simple $u \rightsquigarrow v$ paths that are U-shaped in B. The claim follows by induction on the levels processed by the bottom-up pass.

1. It is trivially true for B being a leaf since T' is nice and thus $|B| = 1$.

2. If B is not a leaf, by Lemma 6 either $|P| = 1$, or $P = (u, y_1, \ldots, y_k, v)$ and P is U-shaped in B_x, where $x = \arg\min_i \mathsf{Lv}(y_i)$. If $|P| = 1$, the claim follows from the initialization of $\textsc{Fwd}'$ and $\textsc{Bwd}'$. Otherwise, if $B_x \neq B$, the proof follows from the induction hypothesis, as by Lemma 6 every such P is also U-shaped in B_x, and B_x is a descendant of B. Finally, if $B_x = B$, decompose P to $P_1 : u \rightsquigarrow x$ and $P_2 : x \rightsquigarrow v$. Note that since P is simple, both P_1 and P_2 are U-shaped in B. Since x is not an intermediate node in either P_1 or P_2, both paths are U-shaped in some descendant B' of B_x and by the induction hypothesis we get that $\mathsf{LD}'_B(u, x) \leq \mathsf{wt}(P_1)$ and $\mathsf{LD}'_B(x, v) \leq \mathsf{wt}(P_2)$ after B' has been examined, hence the inequalities hold when LocDis starts to examine B. In all cases, it follows that after LocDis processes B, it will hold that $\mathsf{LD}'_B(u, v) \leq \mathsf{wt}(P)$.

We now claim that after the second pass processes a bag B_x that is the root bag of some node x, it holds that $\textsc{Fwd}'_x(v) = d(x, v)$ and $\textsc{Bwd}'_x(v) = d(v, x)$ for every $v \in B_x$. The claim follows by induction on the levels processed by the top-down pass.

1. The statement holds trivially if B_x is the root, since T' is nice and thus $|B_x| = 1$.
2. We now proceed inductively to some internal bag B_x examined by the algorithm in the second pass. We only focus on $\textsc{Fwd}'_x$ (the argument is similar for $\textsc{Bwd}'_x$). Consider any simple path $P : x \rightsquigarrow v$. Let u be the first node in P for which B_u is not in $T(B_x)$ and decompose P to $P_1 : x \rightsquigarrow u$ and $P_2 : u \rightsquigarrow v$. By the choice of u, we have that P_1 is U-shaped in B_x, thus by the first pass we have $\mathsf{LD}'_{B_x}(x, u) \leq \mathsf{wt}(P_1)$. By condition T3 of the tree-decomposition, B_v and B_u are ancestors of B_x. Since T' is nice, we have that B_v, B_u are strictly ancestors of B_x (i.e., neither B_v nor B_u is B_x) and hence the induction hypothesis applies to provide that $\textsc{Fwd}'_u(v) = d(u, v)$ or $\textsc{Bwd}'_v(u) = d(u, v)$, and thus $\mathsf{LD}'_{B_x}(u, v) = d(u, v)$ when LocDis starts to examine B_x. It follows that after the second pass processes B_x, we have $\textsc{Fwd}'_x(v) = d(x, v)$, as desired.

Figure 2 depicts the two passes. At the end of the computation, for all $x \in V$ we have $\textsc{Fwd}'(x) = \textsc{Fwd}(x)$ and $\textsc{Bwd}'(x) = \textsc{Bwd}(x)$, as desired. □

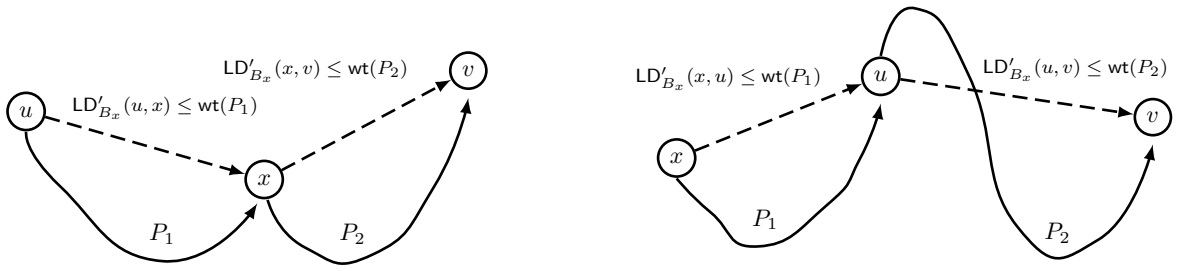


Figure 2: Illustration of the two passes for the local distance computation. In the left, P_1 and P_2 are any U-shaped paths in B_x. In the right, P_1 is any U-shaped path in B_x, and P_2 is any $u \rightsquigarrow v$ path. The inequalities hold when LocDis starts to examine B_x. Afterwards we have $\mathsf{LD}'_{B_x}(u, v) \leq \mathsf{wt}(P_1 \circ P_2)$ (left) and $\mathsf{LD}'_{B_x}(x, v) \leq \mathsf{wt}(P_1 \circ P_2)$ (right), where $P_1 \circ P_2$ is the concatenation of the two paths.

Lemma 8. *Algorithm* LocDis *requires $O(n \cdot t^2)$ time and $O(n \cdot t)$ space.*

Proof. For each $u \in V$, the maps $\textsc{Fwd}'_u$ and $\textsc{Bwd}'_u$ take $O(t)$ space, and represented as ordered sets, their initialization requires $O(t \cdot \log t)$ time, hence $O(n \cdot t \cdot \log t)$ time in total. By Lemma 5, the construction of T' is done in $O(n \cdot t \cdot \log t)$ time and $O(n \cdot t)$ space. The algorithm LocDis examines each of the $O(n \cdot t)$ bags B once in each pass, hence it spends $O(n \cdot t)$ time in traversing T'. For each bag B_x, LocDis spends $O(t^2)$ time to iterate over all pairs $u, v \in B_x$, and $O(t)$ time to update each of the at most $2 \cdot t$ maps $\textsc{Fwd}'$ and $\textsc{Bwd}'$, using the ordered set data-structure. Hence LocDis spends $O(t^2)$ time in total in B_x. There are n such bags B_x that are the root bags of some node x, hence the total time of LocDis is $O(n \cdot t^2)$. The space bound follows from the size of all forward and backward sets, and the size required to store T and T'. □

Lemma 7 and Lemma 8 lead to Theorem 1.

Remark 2. The algorithm LocDis detects the existence of negative cycles, by discovering that $\mathsf{LD}'_B(u,u) < 0$ for some u.

Remark 3 (Comparison to previous work). The concept of local distance has been used before, such as in [2], [18] and [31]. However, the algorithms in all cases use $\Omega(b \cdot t^2)$ space and $\Omega(b \cdot t^3)$ time (or $\Omega(b \cdot t^4)$ in case of [18]), where b is the number of bags in $\mathrm{Tree}(G)$, by storing explicitly all-pairs distances in each bag, and running Bellman-Ford or Floyd-Warshall type of algorithms in each pass. The contribution of this section is twofold: (i) the notion of forward and backward maps (FWD and BWD) shows that $O(n \cdot t)$ space is enough for storing local distances, and (ii) the notion of U-shaped paths is used to show that $O(b + n \cdot t^2)$ time is enough for computing them. In both cases we obtain an improvement by a factor of at least t, as typically $b = \Theta(n)$.

4 Optimal Reachability for Low-Treewidth Graphs

In this section we present a data-structure Reachability which takes as input a graph G of n nodes and treewidth t, and preprocess it in order to answer single-source and pair reachability queries fast. In particular, we establish the following.

Theorem 2. *Given a graph G of n nodes and treewidth t, let $\mathcal{T}(G)$ be the time and $\mathcal{S}(G)$ be the space required for constructing a balanced tree-decomposition $\mathrm{Tree}(G)$ of $O(n)$ bags and width $O(t)$ on a standard RAM with wordsize $W = \Theta(\log n)$. The data-structure* Reachability *correctly answers reachability queries and requires*

1. $O(\mathcal{T}(G) + n \cdot t^2)$ *preprocessing time;*
2. $O(\mathcal{S}(G) + n \cdot t)$ *preprocessing space;*
3. $O\left(\left\lceil \frac{t}{\log n} \right\rceil\right)$ *pair query time; and*
4. $O\left(\frac{n \cdot t}{\log n}\right)$ *single-source query time*

For constant treewidth graphs we have that $\mathcal{T}(G) = O(n)$ and $\mathcal{S}(G) = O(n)$ ([9, Lemma 2]), and thus along with Theorem 2 we obtain the following corollary.

Corollary 1. *Given a graph G of n nodes and constant treewidth, the data-structure* Reachability *requires $O(n)$ preprocessing time and space, and correctly answers (i) pair reachability queries in $O(1)$ time, and (ii) single-source reachability queries in $O\left(\frac{n}{\log n}\right)$ time.*

Intuition. Informally, the preprocessing consists of first obtaining a binary and balanced tree-decomposition T of G, and computing the local reachability information in each bag B (i.e., the pairs $(u,v) \in E^*$ with $u, v \in B$) using algorithm LocDis from Section 3. Then, the whole of preprocessing is done on T, by constructing two types of sets, which are represented as bit sequences and packed into words of length $W = \Theta(\log n)$. Initially, every node u receives an *index* i_u, such that for every bag B, the indices of nodes whose root bag is in $T(B)$ form a contiguous interval. Additionally, for every appearance of node u in a bag B, the node u receives a *local index* l_u^B in B. For brevity, we denote by $(A^i)_{0 \le i \le k}$ the sequence $(A^0, A^1, \ldots A^k)$. When k is implied, we simply write $(A^i)_i$. The following two types of sets are constructed.

1. Sets that store information about subtrees. Specifically, for every node u, the set F_u stores the relative indices of nodes v that can be reached from u, and whose root bag is in $T(B_u)$. These sets are used to answer single-source queries.
2. Sets that store information about ancestors. Specifically, for every node u, two sequences of sets are stored $(\mathsf{F}_u^i)_{0 \le i \le \mathsf{Lv}(u)}$, $(\mathsf{T}_u^i)_{0 \le i \le \mathsf{Lv}(u)}$, such that F_u^i (resp., T_u^i) contains the local indices of nodes v in the ancestor bag B_u^i of B_u at level i, such that $(u,v) \in E^*$ (resp., $(v,u) \in E^*$). These sets are used to answer pair queries.

The sets of the first type are constructed by a bottom-up pass, whereas the sets of the second type are constructed by a top-down pass. Both passes are based on the separator property of tree decompositions (recall Lemma 1 and Lemma 2), which informally states that reachability properties between nodes in distant bags will be captured transitively, through nodes in intermediate bags. Figure 3 illustrates the constructed sets on a small example.

Reachability **Preprocessing.** We now give a formal description of the preprocessing of Reachability that takes as input a graph G of n nodes and treewidth t, and a balanced, binary tree-decomposition $T = \mathrm{Tree}(G)$ of width $O(t)$. After the preprocessing, Reachability supports single-source and pair reachability queries. We say that we "insert" set A to set A' meaning that we replace A' with $A \cup A'$. Sets are represented as bit sequences where 1 denotes membership in the set, and the operation of inserting a set A "at the i-th position" of a set A' is performed by taking the bit-wise logical OR between A and the segment $[i, i + |A|]$ of A'. The preprocessing consists of the following steps.

1. Preprocess T to answer LCA queries in $O(1)$ time [28].
2. Compute the local forward and backward maps of each node $u \in V$ wrt reachability (from Theorem 1). Thus for any bag B and nodes $u, v \in B$, we have $\mathsf{LD}_B(u, v) = 1$ iff $(u, v) \in E^*$.
3. Apply a pre-order traversal on T, and assign an incremental index i_u to each node u at the time the root bag B of u is visited. If there are multiple nodes u for which B is the root bag, assign the indices to those nodes in some arbitrary order. Additionally, store the number s_u of nodes whose root bag is in $T(B)$ and have index at least i_u. Finally, for each bag B and $u \in B$, assign a unique local index l_u^B to u, and store in B the number of nodes a_B contained in all ancestors of B, and the number b_B of nodes in B.
4. For every node u, initialize a bit set F_u of length s_u, pack it into words, and set the first bit to 1.
5. Traverse T bottom-up, and for every bag B execute the following step. For every pair of nodes $u, v \in B$ such that B is the root bag of v and $i_u < i_v$ and $\mathsf{LD}_B(u, v) = 1$, insert F_v to the segment $[i_v - i_u, i_v - i_u + s_v]$ of F_u (the nodes reachable from v now become reachable from u, through v).
6. For every node u initialize two sequences of bit sets $(\mathsf{T}_u^i)_{0 \le i \le \mathsf{Lv}(u)}$, $(\mathsf{F}_u^i)_{0 \le i \le \mathsf{Lv}(u)}$, and pack them into consecutive words. Each set T_u^i and F_u^i has size $b_{B_u^i}$, where B_u^i is the ancestor of B_u at level i.
7. Traverse T top-down, and for B the bag currently visited, for every node $x \in B$, maintain two sequences of bit sets $(\overline{\mathsf{T}}_x^i)_{0 \le i \le \mathsf{Lv}(B)}$ and $(\overline{\mathsf{F}}_x^i)_{0 \le i \le \mathsf{Lv}(B)}$. Each set $\overline{\mathsf{T}}_x^i$ and $\overline{\mathsf{F}}_x^i$ has size b_{B^i}, where B^i is the ancestor of B at level i. Initially, B is the root of T (hence $\mathsf{Lv}(B) = 0$), and set the position l_w^B of $\overline{\mathsf{F}}_x^0$ (resp., $\overline{\mathsf{T}}_x^0$) to 1 for every node w such that $\mathsf{LD}_B(x, w) = 1$ (resp., $\mathsf{LD}_B(w, x) = 1$). For each other bag B encountered in the traversal, do as follows. Let $S = B \cap B'$, where B' is the parent of B in T, and let x range over S.
   (a) For each set sequence of a node x, create a set $\overline{\mathsf{T}}_x$ (resp., $\overline{\mathsf{F}}_x$) of 0s of length b_B, and for every $w \in B$ such that $\mathsf{LD}_B(x, w) = 1$ (resp., $\mathsf{LD}_B(w, x) = 1$), set the l_w^B-th bit of $\overline{\mathsf{F}}_x$ (resp., $\overline{\mathsf{T}}_x$) to 1. Append the set $\overline{\mathsf{T}}_x$ (resp., $\overline{\mathsf{F}}_x$) to $(\overline{\mathsf{T}}_x^i)_i$ (resp., $(\overline{\mathsf{F}}_x^i)_i$). Now each set sequence $(\overline{\mathsf{T}}_x^i)_i$ and $(\overline{\mathsf{F}}_x^i)_i$ has size $a_B + b_B$.
   (b) For each $u \in B$ whose root bag is B, initialize set sequences $(\overline{\mathsf{F}}_u^i)_i$ and $(\overline{\mathsf{T}}_u^i)_i$ with 0s of length $a_B + b_B$ each, and set the bit at position l_u^B of $\overline{\mathsf{F}}_u^{\mathsf{Lv}(B)}$ and $\overline{\mathsf{T}}_u^{\mathsf{Lv}(B)}$ to 1. For every $w \in B$ with $\mathsf{LD}_B(u, w) = 1$ (resp., $\mathsf{LD}_B(w, u) = 1$), insert $(\overline{\mathsf{F}}_w^i)_i$ to $(\overline{\mathsf{F}}_u^i)_i$ (resp., $(\overline{\mathsf{T}}_w^i)_i$ to $(\overline{\mathsf{T}}_u^i)_i$). Finally, set $(\mathsf{F}_u^i)_i$ equal to $(\overline{\mathsf{F}}_u^i)_i$ (resp., $(\mathsf{T}_u^i)_i$ equal to $(\overline{\mathsf{T}}_u^i)_i$).

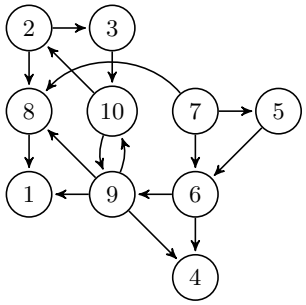
Figure 3 illustrates on a small example the sets F_u, $(\mathsf{F}_u^i)_i$ and $(\mathsf{T}_u^i)_i$ constructed during the preprocessing.

It is fairly straightforward that at the end of the preprocessing, the i-th position of each set F_u is 1 only if $(u, v) \in E^*$, where v is such that $i_v - i_u = i$. The following lemma states the opposite direction, namely that each such i-th position will be 1, as long as the path $P : u \rightsquigarrow v$ only visits nodes with certain indices.

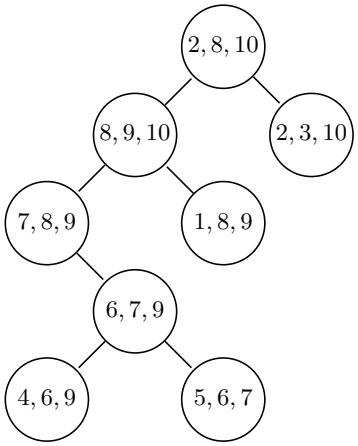
Lemma 9. *At the end of preprocessing, for every node u and v with $i_u \le i_v \le i_u + s_u$, if there exists a path $P : u \rightsquigarrow v$ such that for every $w \in P$, we have $i_u \le i_w \le i_u + s_u$, then the $(i_v - i_u)$-th bit of F_u is 1.*

Proof. We prove inductively the following claim. For every ancestor B of B_v, if there exists $w \in B$ and a path $P_1 : w \rightsquigarrow v$, then exists $x \in B \cap P_1$ such that $i_x \le i_v \le i_x + s_x$ and the $i_v - i_x$-th bit of F_x is 1. The proof is by induction on the length of the simple path $P_2 : B \rightsquigarrow B_v$.

1. If $|P_2| = 0$, the statement is true by taking $x = v$, since the 0-th bit of F_v is 1.
2. If $|P_2| > 0$, examine the child B' of B in P_2. By Lemma 2, there exists $x \in B \cap B' \cap P$, and let $P_3 : x \rightsquigarrow v$. By the induction hypothesis there exists some $y \in B' \cap P_3$ with $i_y \le i_v \le i_y + s_y$ and the $i_v - i_y$-th bit of F_y is 1. If $y \in B$, we take $x = y$. Otherwise, B' is the root bag of y, and by the local distance computation of Theorem 1, it is $\mathsf{LD}_{B'}(x, y) = 1$. Additionally, by the choice of x, y, we have $i_x < i_y$ and $s_x \ge s_y + i_y - i_x$,



(a)



(b)

u	i_u	Bit-set F_u									
		0	1	2	3	4	5	6	7	8	9
2	0	1	1	1	1	0	0	1	0	1	1
8	1		1	0	0	0	0	0	0	0	1
10	2			1	1	0	0	1	0	1	1
9	3				1	0	0	1	0	1	
7	4					1	1	1	1		
6	5						1	1	0		
4	6							1			
5	7								1		
1	8									1	
3	9										1

(c)

	$i=0$			$i=1$			$i=2$			$i=3$		
v	2	8	10	8	9	10	7	8	9	6	7	9
$l_v^{B_6^i}$	0	1	2	0	1	2	0	1	2	0	1	2
$(\mathsf{F}_6^i)_i$	1	1	1	1	1	1	0	1	1	1	0	1
$(\mathsf{T}_6^i)_i$	0	0	0	0	0	0	1	0	0	1	1	0

(d)

Figure 3: (a), (b): A graph G and a tree-decomposition $\mathrm{Tree}(G)$. (c): The sets F_u constructed from step 5 to answer single-source queries. The j-th bit of a set F_u is 1 iff $(u, v) \in E^*$, where v is such that $i_u - i_v = j$. (d): The set sequences $(\mathsf{F}_u^i)_i$ and $(\mathsf{T}_u^i)_i$ constructed from step 6 to answer pair queries, for $u = 6$. For every $i \in \{0, 1, 2, 3\}$ and ancestor B_6^i of B_6 at level i, every node $v \in B_u^i$ is assigned a local index $l_v^{B_6^i}$. The j-th bit of set F_6^i (resp. T_6^i) is 1 iff $(6, v) \in E^*$ (resp. $(v, 6) \in E^*$), where v is such that $l_v^{B_6^i} = j$.

thus $i_x \leq i_v \leq i_x + s_x$. Then in step 5, F_y is inserted in position $i_y - i_x$ of F_x, thus the bit at position $i_y - i_x + i_v - i_y = i_v - i_x$ of F_x will be 1, and we are done.

When B_u is examined, by the above claim there exists $x \in P$ such that $i_x \leq i_v$ and the $i_v - i_x$-th bit of F_x is 1. If $x = u$ we are done. Otherwise, by the choice of P, we have $i_u < i_x$, which can only happen if B_u is also the root bag of x. Then in step 5, F_x is inserted in position $i_x - i_u$ of F_u, and hence the bit at position $i_x - i_u + i_v - i_x = i_v - i_u$ of F_x will be 1, as desired. □

Lemma 10. *At the end of preprocessing, for every node u, for every $v \in B_u^i$ where B_u^i is the ancestor of B_u at level i, we have that if $(u, v) \in E^*$ (resp., $(v, u) \in E^*$), then the $l_v^{B_u^i}$-th bit of F_u^i (resp., T_u^i) is* 1 .

Proof. The proof is by application of Lemma 2 inductively on the path $B_u^i \rightsquigarrow B$, similarly to Lemma 9. □

Lemma 11. *Given a graph G with n nodes and treewidth t, let $\mathcal{T}(G)$ be the time and $\mathcal{S}(G)$ be the space required for constructing a balanced binary tree-decomposition of G with $O(n)$ bags and width $O(t)$. The preprocessing phase of* Reachability *on G requires $O(\mathcal{T}(G) + n \cdot t^2)$ time and $O(\mathcal{S}(G) + n \cdot t)$ space.*

Proof. First, we construct a binary tree-decomposition $\mathrm{Tree}(G)$ of G with $b = O(n)$ bags, height $h = O(\log n)$ and width $t' = O(t)$ in $\mathcal{T}(G)$ time and $\mathcal{S}(G)$ space. We establish the complexity of each preprocessing step separately.

1. By a standard construction for balanced trees, preprocessing T to answer LCA queries in $O(1)$ time requires $O(b) = O(n)$ time.
2. By Theorem 1, this step requires $O(n \cdot t'^2) = O(n \cdot t^2)$ time and $O(n \cdot t') = O(n \cdot t)$ space.
3. Every bag B is visited once, and each operation on B takes constant time. We make $O(t')$ such operations in B, hence this step requires $O(b \cdot t') = O(n \cdot t)$ time in total.

4-5. The space required in this step is the space for storing all the sets F_u of size s_u each, packed into words of length W:

$$\sum_{u \in V} \left\lceil \frac{s_u}{W} \right\rceil = \sum_{i=0}^{h} \sum_{u:\mathsf{Lv}(u)=i} \left\lceil \frac{s_u}{W} \right\rceil \leq \sum_{i=0}^{h} \sum_{u:\mathsf{Lv}(u)=i} \left(\frac{s_u}{W} + 1 \right)$$
$$= \frac{1}{W} \cdot \sum_{i=0}^{h} \sum_{u:\mathsf{Lv}(u)=i} s_u + \sum_{i=0}^{h} \sum_{u:\mathsf{Lv}(u)=i} 1 \leq \frac{1}{W} \cdot \sum_{i=0}^{h} n \cdot (t' + 1) + n = O(n \cdot t)$$

since $h = O(\log n)$, $t' = O(t)$ and $W = \Theta(\log n)$. Note that we have $\sum_{u:\mathsf{Lv}(u)=i} s_u \leq n \cdot (t' + 1)$ because $|\bigcup_u \mathsf{F}_u| \leq n$ (as there are n nodes) and every element of $\bigcup_u \mathsf{F}_u$ belongs to at most $t' + 1$ such sets F_u (i.e., for those u that share the same root bag at level i). The time required in this step is $O(n \cdot t)$ in total for iterating over all pairs of nodes (u, v) in each bag B such that B is the root bag of either u or v, and $O(n \cdot t^2)$ for the set operations, by amortizing $O(t)$ operations per word used.

6. The time and space required for storing each sequence of the sets $(\mathsf{F}_u^i)_{0 \leq i \leq \mathsf{Lv}(u)}$ and $(\mathsf{T}_u^i)_{0 \leq i \leq \mathsf{Lv}(u)}$ is:

$$\sum_{u \in V} 2 \cdot \left\lceil \frac{a_{B_u} + b_{B_u}}{W} \right\rceil \leq 2 \cdot n \cdot \left\lceil \frac{(t' + 1) \cdot h}{W} \right\rceil = O(n \cdot t)$$

since $a_{B_u} + b_{B_u} \leq (t' + 1) \cdot h$, $h = O(\log n)$ and $W = \Theta(\log n)$.

7. The space required is the space for storing the set sequences $(\overline{\mathsf{T}}_v^i)_i$ and $(\overline{\mathsf{F}}_v^i)_i$, which is $O(t^2)$ by a similar argument as in the previous item. The time required is $O(t)$ for initializing every new set sequence $(\overline{\mathsf{T}}_u^i)_i$ and $(\overline{\mathsf{F}}_u^i)_i$ and this will happen once for each node u at its root bag B_u, hence the total time is $O(n \cdot t)$.

□

Reachability **Querying.** Given the preprocessing of Reachability, each query is answered as follows.

Pair query. Given a pair query (u, v), find the LCA B of bags B_u and B_v. Obtain the sets $\mathsf{F}_u^{\mathsf{Lv}(B)}$ and $\mathsf{T}_v^{\mathsf{Lv}(B)}$ of size b_B. Both sets start in bit position a_B of the sequences $(\mathsf{F}_u^i)_i$ and $(\mathsf{T}_v^i)_i$. Return True iff the logical-AND of the sets $\mathsf{F}_u^{\mathsf{Lv}(B)}$ and $\mathsf{T}_v^{\mathsf{Lv}(B)}$ contains an entry which is 1.

Single-source query. Given a single-source query u, create a bit set A of size n, initially all 0s. For every node $x \in B_u$ with $i_x \leq i_u$, if the $l_x^{B_u}$-th bit of $\mathsf{F}_u^{\mathsf{Lv}(u)}$ is 1, insert F_x to the segment $[i_x, i_x + s_x]$ of A. Then traverse the path from B_u to the root of T, and let B_u^i be the ancestor of B_u at level $i < \mathsf{Lv}(B_u)$. For every node $x \in B_u^i$, if the $l_x^{B_u^i}$-th bit of F_u^i is 1, set the i_x-th bit of A to 1. Additionally, if B_u^i has two children, let B be the child of B_u^i that is not ancestor of B_u, and $j_{\min}$ and $j_{\max}$ the smallest and largest indices, respectively, of nodes whose root bag is in $T(B)$. Insert the segment $[j_{\min} - i_x, j_{\max} - i_x]$ of F_x to the segment $[j_{\min}, j_{\max}]$ of A. Report that the set of nodes v reached from u is those v for which the i_v-th bit of A is 1.

Lemma 12. *After the preprocessing phase of* Reachability*, pair and single-source reachability queries are answered correctly in* $O\left(\left\lceil \frac{t}{\log n} \right\rceil\right)$ *and* $O\left(\frac{n \cdot t}{\log n}\right)$ *time respectively.*

Proof. The correctness of the pair query comes immediately from Lemma 10 and Lemma 1, which implies that every path $u \rightsquigarrow v$ must go through the LCA of B_u and B_v. The time complexity follows from the $O\left(\left\lceil \frac{t}{W} \right\rceil\right)$ word operations on the sets $\mathsf{F}_u^{\mathsf{Lv}(B)}$ and $\mathsf{T}_v^{\mathsf{Lv}(B)}$ of size $O(t)$ each.

Now consider the single-source query from a node u and let v be any node such that there is a path $P : u \rightsquigarrow v$. Let B be the LCA of B_u, B_v, and by Lemma 1, there is a node $y \in B \cap P$. Let x be the last such node in P, and let $P' : x \rightsquigarrow v$ be the suffix of P from x. It follows that P' is a path such that for every $w \in P'$ we have $i_x \leq i_w \leq i_x + s_x$.

1. If B_v is an ancestor of B_u, then necessarily $x = v$, and by Lemma 10, the l_v^B-th bit of $\mathsf{F}_u^{\mathsf{Lv}(B)}$ is 1. Then the algorithm sets the i_v-th bit of A to 1.
2. Else, B_x is an ancestor of B_v (recall that a bag is an ancestor of itself), and by Lemma 9, the $(i_v - i_x)$-th bit of F_x is 1.
   (a) If B is B_u, the algorithm will insert F_x to the segment $[i_x, i_x + s_x]$ of A, thus the $i_x + i_v - i_x = i_v$-th bit of A is set to 1.
   (b) If B is not B_u, it can be seen that $j_{\min} \leq i_v \leq j_{\max}$, where $j_{\min}$ and $j_{\max}$ are the smallest and largest indices of nodes whose root bag is in $T(B')$, with B' the child of B that is not ancestor of B_u. Since the $(i_v - i_x)$-th bit of F_x is 1, the $(i_v - j_{\min})$-th bit of the $[j_{\min}, j_{\max}]$ segment of F_x is 1, thus the $j_{\min} + i_v - j_{\min} = i_v$-th bit of A is set to 1.

Regarding the time complexity, the algorithm performs $O(h \cdot t') = O(h \cdot t)$ set insertions to A. For every position j of A, the number of such set insertions that overlap on j is at most $t' + 1$ (once for every node in the LCA of B_u and B_v, where v is such that $i_v = j$). Hence if H_i is the size of the i-th insertion in A, we have $\sum_i H_i \leq n \cdot (t' + 1)$. Since the insertions are word operations, the total time spent for the single source query is

$$\sum_{i=0}^{h} \left\lceil \frac{H_i}{W} \right\rceil \leq h + \sum_{i=0}^{h} \frac{H_i}{W} \leq h + \frac{n \cdot (t' + 1)}{W} = O\left(\frac{n \cdot t}{\log n}\right)$$

since $h = O(\log n)$, $t' = O(t)$ and $W = \Theta(\log n)$. □

5 Distance Queries in Low-Treewidth Graphs

In this section we describe a method for preprocessing low-treewidth graphs in order to answer distance queries. We provide a data-structure called Distance which is parametric on some $\lambda \in \mathbb{N}$, and yields the following result.

Theorem 3. *Given a graph G of n nodes and treewidth t, let $\mathcal{T}(G)$ be the time and $\mathcal{S}(G)$ be the space required for constructing a nice tree-decomposition $\mathrm{Tree}(G)$ of $O(n)$ bags and width t. For any $\lambda \in \mathbb{N}$, the data-structure* Distance *correctly answers distance queries on G and requires*

1. $O\left(\mathcal{T}(G) + (\lambda+2)\cdot n \cdot t^2 \cdot \log^{(\lambda)*} n\right)$ *preprocessing time;*
2. $O\left(\mathcal{S}(G) + (\lambda+2)\cdot n \cdot t \cdot \log^{(\lambda)*} n\right)$ *space;*
3. $O\left((\lambda+1)\cdot t^2\right)$ *pair query time; and*
4. $O(n\cdot t)$ *single-source query time.*

The algorithms of this section are based on Lemma 3 which states that for nodes that appear in bags B, B' of the tree-decomposition $T = \mathrm{Tree}(G)$, their distance can be written as a sum of distances $d(x_i, x_{i+1})$ between pairs of nodes (x_i, x_{i+1}) that appear in bags B_i that constitute the unique $B \rightsquigarrow B'$ path in T. Recall that the local distance computation of Section 3 provides a way for computing such distances $d(x_i, x_{i+1})$ fast. Here we build on top of the local distance computation to develop a data-structure called Distance for answering distance queries. The main part of the preprocessing consists of manipulating *summary trees*. Intuitively, given a tree-decomposition T, a summary tree $\overline{T}$ of T consists of a subset of bags of T, such that:

1. The tree $\overline{T}$ is obtained from T by removing some bags and adding edges from their parents to their children.
2. For bags B with children B' in $\overline{T}$, for all $u \in B$ and $v \in B'$, the distances $d(u,v)$ and $d(v,u)$ are stored in B'.

Hence, in such a summary tree $\overline{T}$, the distances between all nodes in B and B' have been summarized and can be retrieved fast, regardless of the length of the $B \rightsquigarrow B'$ path in T (which we would otherwise have to pay as a cost for retrieving them using Lemma 3). The preprocessing of the data-structure applies recursive summarizations of T, so that in the end, for any two nodes $u, v \in V$, the distance $d(u,v)$ can be written as a sum of $O(\lambda+1)$ summarized distances, which can be retrieved by looking up $O((\lambda+1)\cdot t^2)$ such summarized distances. The algorithmic technique is an adaptation of [3]. A direct application results in higher preprocessing time, space, and query time complexities by at least a factor of t (as in [18]).

A note on preprocessing. Most of this section focuses on answering pair distance queries $d(u,v)$. Such a query is computed by retrieving the distances $d(u,x)$ and $d(x,v)$, where x ranges over nodes of the bag B that is the LCA of B_u and B_v in $\mathrm{Tree}(G)$. For ease of presentation, we describe the preprocessing phase of Distance for handling queries $d(u,v)$ where B_u is an ancestor of B_v. The case where B_u is a descendant of B_v is follows by straightforward modifications of the former case, and is omitted.

Summary trees. Given a tree-decomposition $T = \mathrm{Tree}(G)$, a summary tree of T is a pair $(\overline{T}, \mathsf{BS})$ where:

1. $\overline{T} = (V_{\overline{T}}, E_{\overline{T}})$ is a (connected) tree with $V_{\overline{T}} \subseteq V_T$, and each bag of $\overline{T}$ is a child of its lowest ancestor in T that appears in $\overline{T}$.
2. BS are collections of *summary maps* defined as follows. For each node u such that B is a smallest-level bag in $\overline{T}$ that contains u, and B' is the parent of B in $\overline{T}$, the backward summary map of u in $\overline{T}$ is defined as $\mathsf{BS}_u^B : B' \to \mathbb{Z}$ with $\mathsf{BS}_u^B(v) = d(v,u)$.

Observe that for any bag B and ancestor B' of B in $\overline{T}$, every intermediate bag B'' in the simple path $B \rightsquigarrow B'$ in $\overline{T}$ is also an intermediate bag in the simple path $B \rightsquigarrow B'$ in T. It follows from property *T3* of tree-decompositions that $B'' \subseteq B \cap B'$. Similarly to tree-decompositions, we say that a bag B "forgets" a node u, if $u \in B$ and $u \notin B'$ for every ancestor B' of B in $\overline{T}$. In contrast to tree-decompositions, u might have more than one forget bags.

Operations on summary trees. The preprocessing phase of Distance on the tree-decomposition $T = \mathrm{Tree}(G)$ constructs a summary tree $\overline{T}$ out of T, and further preprocesses it, using the following operations. Given a summary tree $\overline{T}$, we partition it into components and apply recursive summarizations on the components. To achieve certain bounds on the partitions, we first make $\overline{T}$ binary, and then apply the partitioning. Then, the root and leaf distance map computation is applied in each component, to calculate the distances to nodes appearing in the root and leaves of the component. Finally the summarization procedure constructs a new summary tree from the roots of the components, and the process repeats recursively. The operations of tree binarization, tree $f(k)$-partitioning, root and leaf distance map

computation, and tree summarization are described below. Based on them, we afterwards give a formal description of the preprocessing.

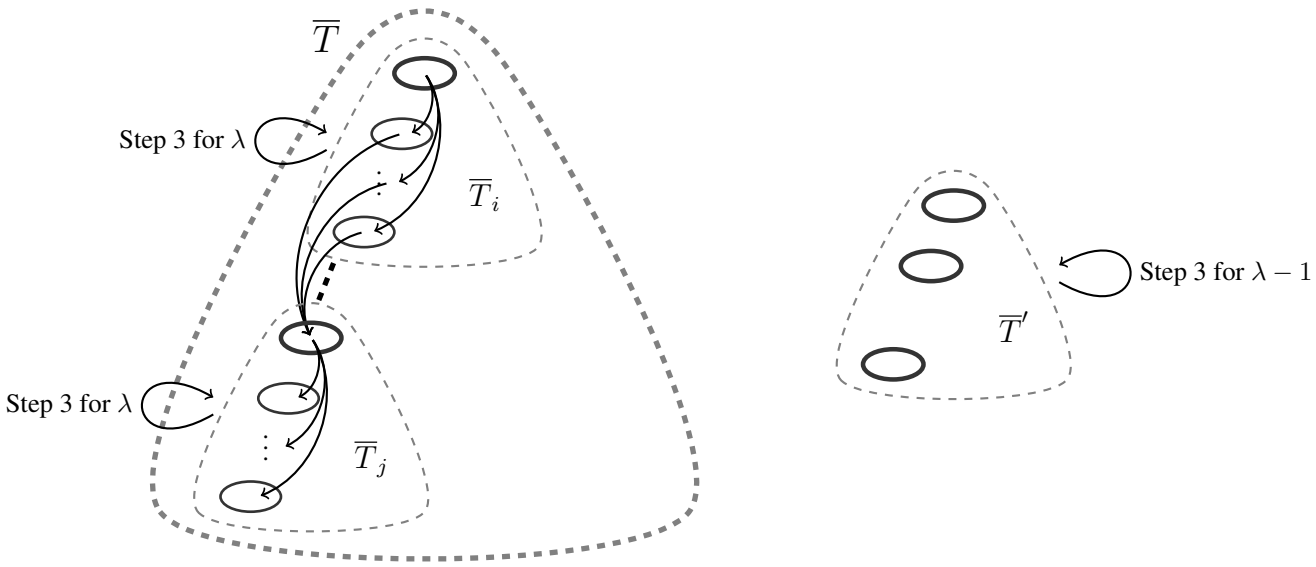


Figure 4: Scheme of the Step 3 recursion of the preprocessing of Distance for a given λ. A summary tree $\overline{T}$ of size k is partitioned into components of size $f(k)$ each, and the root and leaf distance set computation is executed. The root bags are shown in boldface. The recursion is then two-fold: (1) the roots of the components form a new summary tree $\overline{T}'$ which is recursively processed for $\lambda - 1$, and (2) each component is itself a summary tree, recursively processed for λ.

Tree binarization. Given a summary tree $(\overline{T}, \mathsf{BS})$ of some $T = \mathrm{Tree}(G)$, the binarization of $\overline{T}$ is done as follows. For every bag B with children $B_1, \dots B_j$, if $j \geq 3$, then introduce $j-2$ copies of B, namely, $\widehat{B}_2, \widehat{B}_3, \dots, \widehat{B}_{j-1}$. The transformation is as follows: B has two children, $\widehat{B}_2$ and B_1; for all $2 \leq i \leq j-2$ the two children of $\widehat{B}_i$ are B_i and $\widehat{B}_{i+1}$; and finally, the last copy $\widehat{B}_{j-1}$ has bags B_j and B_{j-1} as children. If $\overline{T}$ has k bags, this process takes $O(k \cdot t)$ time, and the size of the new tree is at most $2 \cdot k$. Note that the binarized tree $(\overline{T}, \mathsf{BS})$ is also a summary tree of T. In the sequel we only consider summary trees $(\overline{T}, \mathsf{BS})$ where $\overline{T}$ is a binary tree.

Tree $f(k)$-partitioning. Given a function $f : \mathbb{N} \to \mathbb{N}$, a $k \in \mathbb{N}$ and a binary summary tree $(\overline{T}, \mathsf{BS})$ of some tree-decomposition T with $|V_{\overline{T}}| = k$ bags, the $f(k)$-partitioning of $\overline{T}$ consists of partitioning $\overline{T}$ into $O\left(\frac{k}{f(k)}\right)$ connected components that contain at most $f(k)$ bags each. The partitioning is performed as follows: use a DFS to keep track of the number of nodes in each subtree $\widetilde{T}$ of $\overline{T}$, and whenever the number of nodes in $\widetilde{T}$ becomes at least $\frac{f(k)}{2}$ cut $\widetilde{T}$ off into its own component. Note that since $(\overline{T}, \mathsf{BS})$ is binary, no component becomes larger than $f(k)$. This partitioning takes linear time in the size of $\overline{T}$. Each component $\overline{T}_i$ of the partitioning is a summary tree of T.

Root and leaf distance map computation. Consider a summary tree $(\overline{T}, \mathsf{BS})$ and an $f(k)$- partitioning on $\overline{T}$, and examine each component $\overline{T}_i$ with root B_0^i. For every node u that appears in some bag of $\overline{T}_i$, we compute the root and leaf distance maps $R_u : B_0^i \to \mathbb{Z} \cup \{\infty\}$ and $L_u^j : B_0^j \to \mathbb{Z} \cup \{\infty\}$ of u defined as $R_u(v) = d(v, u)$ and $L_u^j(v) = d(u, v)$ where B_0^j is the root of a child component $\overline{T}_j$ of $\overline{T}_i$ in the partitioning, and that B_0^j has an ancestor bag B in $\overline{T}_i$ that forgets u. In the following we describe the construction of these maps. We consider that the local distance computation from Section 3 has been carried out (i.e., for each bag B we have constructed the map LD_B with $\mathsf{LD}_B(u, v) = d(u, v)$ for all $u, v \in B$).

- *(Root distance map computation).* For each $v \in B_0^i$, to determine the distance $R_u(v) = d(v, u)$ for all nodes u that appear in bags of $\overline{T}_i$, execute the following steps. Execute a BFS in $\overline{T}_i$, and upon examining a bag B, for

each node u that B forgets, assign

$$R_u(v) = \min_{y \in B} \left(R_y(v) + \mathsf{LD}_B(y, u) \right)$$

- *(Leaf distance map computation).* The leaf distance computation is similar to the root distance computation: For every root B_0^j of a child component $\overline{T}_j$ of $\overline{T}_i$ and every $v \in B_0^j$, to determine the distance $L_u^j(v)$ for all nodes u that appear in bags of $\overline{T}_i$, traverse the path $B_0^j \rightsquigarrow B_0^i$. For every encountered bag B and $u \in B$, let y range over nodes that B forgets. Assign

$$L_u^j(v) = \min_y \left(L_y^j(v) + \mathsf{LD}_B(u, y) \right)$$

The correctness of the construction follows from Lemma 3 and a simple induction on the BFS. The time and space requirement of the root and leaf distance map computation is as follows: Each root of each component is the source of at most $2 \cdot (t+1)$ traversals, leading to $O\left(t \cdot \frac{k}{f(k)}\right)$ traversals of length $O(f(k))$ each. If each bag of $\overline{T}$ forgets at most n' nodes, then the cost of each step of each traversal is $O(t \cdot n')$ for updating the set A and the distance maps of the n' nodes for which the current bag of the traversal forgets. Hence the total time spent is $O(k \cdot t^2 \cdot n')$. The space required is dominated by the space used for storing the root and leaf distance maps, which is bounded by $O(k \cdot t \cdot n')$, since there exist $O\left(\frac{k}{f(k)} \cdot t\right)$ nodes v that appear in the root bag of some component $\overline{T}_i$, and each such node appears in $O(f(k) \cdot n')$ distance maps.

Tree summarization. Consider a summary tree $(\overline{T}, \mathsf{BS})$ of size k that has been partitioned into $\frac{k}{f(k)}$ components, for some f, and the root and leaf distance map computation has been carried out. The summarization of $\overline{T}$ is done by constructing a new summary tree $(\overline{T}', \mathsf{BS}')$ as follows. The bag set $V_{\overline{T}'}$ contains all the bags B_0^i that appeared as the root of some component $\overline{T}_i$ of the partitioning. A bag B_0^i is a parent of B_0^j in $\overline{T}'$ iff $\overline{T}_j$ is a child component of $\overline{T}_i$ in the partitioning of $\overline{T}$. For every node u such that there is a bag B_0^j in $\overline{T}'$ that forgets u, let B_0^i be the parent of B_0^j in $\overline{T}'$. Let B be the parent bag of B_0^j in $\overline{T}$. We construct the new backwards summary map $\mathsf{BS}_u^{'B_0^j} : B_0^i \to \mathbb{Z} \cup \{\infty\}$ as

$$\mathsf{BS}_u^{'B_0^j}(v) = \min_{y \in B} (\mathsf{BS}_u^{B_0^j}(y) + R_y(v))$$

if the bag B_0^j forgets u in $\overline{T}$, otherwise $\mathsf{BS}'_u = R_u$.

If there are at most n' nodes that each bag of $\overline{T}'$ forgets, the computation of the summary maps requires $O\left(\frac{k}{f(k)} \cdot t^2 \cdot (n'+1)\right)$, since there are $O\left(\frac{k}{f(k)}\right)$ bags in $\overline{T}'$, and we spend $O(t^2)$ time for each of the $O(n')$ nodes that each such bag forgets (plus some linear in $\frac{k}{f(k)}$ term for all other nodes). The space required is $O\left(\frac{k}{f(k)} \cdot t \cdot (n'+1)\right)$ for storing the newly computed summary maps.

Preprocessing $T = \mathrm{Tree}(G)$. Now we describe the preprocessing of T in order to answer distance queries of the form (u, v), where B_u is an ancestor of B_v. We later show how to answer general pair queries. The preprocessing is parametric on an arbitrarily chosen $\lambda \in \mathbb{N}$, which results in $O((\lambda+2) \cdot n \cdot t^2 \log^{(\lambda)*} n)$ preprocessing time and $O\left((\lambda+1) \cdot t^2\right)$ query time (see Figure 4). The preprocessing phase of Distance is performed as follows.

Step 1 First, use LocDis from Section 3 to compute the local distance maps in T, and construct the summary tree $(\overline{T}, \mathsf{BS})$, of T with $V_{\overline{T}} = V_T$. The computation of the summary edges BS is done by traversing T via DFS, and for each bag B_u with parent B, constructing LD_B. Lemma 3 implies that for each $v \in B$, we have $d(u,v) = \min_{y \in B_u \cap B} (d(u,y) + d(y,v))$ and $d(v,u) = \min_{y \in B_u \cap B} (d(v,y) + d(y,u))$. These distances exist in LD_B, and are used to construct the backward summary map BS_u.

Step 2 Apply the root and leaf distance map computation on $\overline{T}$, recursively for $\log^{(\lambda+1)*} n$ levels, and $f(k) = t \cdot \log^{(\lambda)*} k$. That is, each time consider a summary tree of k bags, partition it into $O\left(\frac{k}{t \cdot \log^{(\lambda)*} k}\right)$ components of size $f(k)$ each, and compute the root and leaf distance maps. The next level processes summary trees $\overline{T}_i$

corresponding to components in the current level. Initially we have $k = O(n)$. For every partitioned summary tree $\overline{T}_i$ constructed in this recursion, perform a tree summarization, and let $\overline{T}_i'$ be the resulting summary tree. Execute Step 3 on $\overline{T}_i'$ for $\lambda - 1$.

Step 3 Given a summary tree $\overline{T}$ of size k and some λ execute the following steps:

(a) If $\lambda \geq 0$, perform an $f(k) = \log^{(\lambda)*} k$ partitioning of $\overline{T}$, and compute the root and leaf distance maps for each component $\overline{T}_i$. If $\overline{T}_i$ has size more than one, execute Step 3 on $\overline{T}_i$ for λ. Perform a summarization on the partitioned tree $\overline{T}$, and let $\overline{T}'$ be the resulting summary tree. Then, execute Step 3 on $\overline{T}'$ for $\lambda - 1$.

(b) If $\lambda = -1$, perform an $f(k) = \frac{2 \cdot k}{3}$ partitioning of $\overline{T}$, and compute the root and leaf distance maps for each component $\overline{T}_i$. If $\overline{T}_i$ has size more than one, execute Step 3 on $\overline{T}_i$ for λ.

Step 4 For every summary tree $\overline{T}$ in the last level of the recursion of Step 2, perform an all-pairs distance computation on the subgraph of G induced by nodes u that appear in $\overline{T}$.

Step 5 Preprocess each recursion tree generated in Steps 2 and 3 to answer LCA queries in constant time [28].

The time and space of preprocessing. Here we analyze the time and space requirements of Steps 1 to 5 of the preprocessing.

Lemma 13. *Given a nice tree-decomposition T of G and some $\lambda \in \mathbb{N}$, the preprocessing requires $O\left((\lambda+2) \cdot t^2 \cdot n \cdot \log^{(\lambda)*} n\right)$ time and $O\left((\lambda+2) \cdot t \cdot n \log^{(\lambda)*} n\right)$ space.*

Proof. We discuss the time and space complexity of each step below.

Step 1 LocDis requires $O(n \cdot t^2)$ time and $O(n \cdot t)$ space (Lemma 8). The construction of the summary maps BS_u happens at most once for each node u, requiring $O(t^2)$ time for building the local distance map LD_B of the parent bag B of B_u, and $O(t^2)$ time for calculating $d(u, v)$ for all $v \in B$. Hence this step requires $O(n \cdot t^2)$ time. The space required is $O(n \cdot t)$ for storing the computed summary tree.

Step 3 Given a summary tree of k bags, each bag is the root bag of at most $t+1$ nodes, so we substitute $n' = (t+1)$ for the cost of the distance map computation and tree summarization. Then, the time spent for root and leaf distance map computation, as well as tree summarization is $O(k \cdot t^3)$. Let $\mathcal{T}_\lambda(k)$ denote the time spent in Step 3 on a summary tree of size k for a parameter λ. It is easy to verify that for $\lambda = -1$, it is $\mathcal{T}_\lambda(k) = O\left(t^3 \cdot k \cdot \log k\right)$. For $\lambda \geq 0$, it is

$$\mathcal{T}_\lambda(k) \leq \frac{k}{\log^{(\lambda)*} k} \cdot \mathcal{T}_\lambda\left(\log^{(\lambda)*} k\right) + \mathcal{T}_{\lambda-1}\left(\frac{k}{\log^{(\lambda)*} k}\right) + O\left(k \cdot t^3\right)$$

and thus $\mathcal{T}_\lambda(k) = O\left((\lambda+2) \cdot t^3 \cdot k \log^{(\lambda+1)*} k\right)$.

Similarly, the space used for Step 3 is $O\left((\lambda+2) \cdot t^2 \cdot k \cdot \log^{(\lambda+1)*} k\right)$.

Step 2 In each level of the recursion of Step 2, every summary tree is a subtree of the nice tree-decomposition T. It follows that for the distance map computation and tree summarization $n' = 1$, since every bag is the root bag of at most one node. For a summary tree of size k in some level i of the recursion, the time spent for the distance map computation, summarization, and calls to Step 3 is then

$$O(k \cdot t^2) + \mathcal{T}_{\lambda-1}\left(\frac{k}{t \cdot \log^{(\lambda+1)*} k}\right) = O\left((\lambda+2) \cdot k \cdot t^2\right)$$

and since there are $O\left(\frac{n}{k}\right)$ summary trees in level i, the total time spent in processing level i is $O((\lambda + 2) \cdot n \cdot t^2)$. Finally, there are $O\left(\log^{(\lambda+1)*} n\right)$ such levels, and the total time spent in Step 2 is $O\left((\lambda+2) \cdot t^2 \cdot n \log^{(\lambda+1)*} n\right)$. Similarly, the space used is $O\left((\lambda+2) \cdot t \cdot n \cdot \log^{(\lambda+1)*} n\right)$

Step 4 Note that every summary tree $\overline{T}$ in the last level of the recursion of Step 2 has size at most

$$t \cdot \underbrace{\log^{(\lambda)*}\left(t \cdot \log^{(\lambda)*}\left(\ldots t \cdot \log^{(\lambda)*} O(n)\right)\right)}_{\log^{(\lambda+1)*} n \text{ applications}} = t \cdot \left(\log^{(\lambda)*} t + \log^{(\lambda)*}\log^{(\lambda)*} t + \cdots + \log^{((\lambda)*)^{\log^{(\lambda+1)*} n - 1}} t + O(1)\right)$$

$$= O\left(t \cdot \log^{(\lambda)*} t\right)$$

since in discrete context, for all $x \geq 0$, we have $\log^{(\lambda)*} x \leq \frac{2 \cdot x}{3}$, and hence $\sum_i \log^{((\lambda)*)^i} t \leq 3 \cdot \log^{(\lambda)*} t$. Since $\overline{T}$ is a subtree of a nice tree-decomposition, the total number of nodes that appear in $\overline{T}$ is $O\left(t \cdot \log^{(\lambda)*} t\right)$. It follows by the way edges are stored in T that the number of edges in $\overline{T}$ is $O\left(t^2 \cdot \log^{(\lambda)*} t\right)$. We conclude that the all pairs distance computation in $\overline{T}$ requires $O\left(t^3 \cdot \left(\log^{(\lambda)*} t\right)^2\right)$ time, and there are $O\left(\frac{n}{t \cdot \log^{(\lambda)*} t}\right)$ such summary trees $\overline{T}$, resulting in $O\left(t^2 \cdot n \cdot \log^{(\lambda)*} t\right)$ total time. The space required is $O\left(t \cdot n \cdot \log^{(\lambda)*} t\right)$ for storing $\frac{n}{t \cdot \log^{(\lambda)*} t}$ lookup matrices of size $O\left(\left(t \cdot \log^{(\lambda)*} t\right)^2\right)$ each.

Step 5 We can preprocess each recursion tree in time and space proportional to its size [36] so this step adds no overhead to the complexity.

The desired result follows. □

Ancestor pair query. Given $u, v \in V$ with B_u being an ancestor of B_v, the task is to retrieve $d(u, v)$. First, test whether the query can be answered by the lookup tables constructed in Step 4. If not, perform an LCA query on the recursion tree of Step 2 to find the smallest component $\overline{T}$ of T that contains both bags B_u and B_v, and it follows that B_u and B_v appear in two different sub-components $\overline{T}_u$ and $\overline{T}_v$. We obtain the corresponding root distance map R_v of v and the leaf distance map L_u^j of u, such that B_0^j is the root of the last component $\overline{T}_j$ on the path between $\overline{T}_u$ and $\overline{T}_v$ (possibly $\overline{T}_j = \overline{T}_v$)[1]. We consider the following cases:

1. If $\overline{T}_v$ is a child component of $\overline{T}_u$, then $d(u, v) = \min_{y \in B_0^j}(d(u, y) + d(y, v))$, where both distances have been computed in L_u^j and R_v respectively. This requires $O(t)$ time.
2. If $\overline{T}_v$ is not a child component of $\overline{T}_u$, the process repeats recursively for the recursion of Step 3 and bags B_0^j and B_0^v, where B_0^v is the root of $\overline{T}_v$. Lemma 3 implies that

$$d(u, v) = \min_{u' \in B_0^j, v' \in B_0^k}(d(u, u') + d(u', v') + d(v', v))$$

and the goal is to retrieve all distances $d(u', v')$, as $d(u, u')$ and $d(v', v)$ have been computed in L_u^j and R_v respectively. Using the same process as for u, v, all $O(t^2)$ distances $d(u', v')$ are retrieved recursively. The process might be repeated further on the recursion of Step 3, for up to $\lambda + 1$ levels. Hence the worst case time for answering the query is $O((\lambda + 1) \cdot t^2)$. The core process is depicted in Figure 5.

Pair query. The preprocessing and query phases for answering distance queries (u, v) where B_v is an ancestor of B_u is similar to that where B_u is ancestor of B_v. In order to handle general pair queries, additionally preprocess T to answer LCA queries in constant time. Let B be the LCA of B_u and B_v. Using the ancestor pair queries from above, we can compute the maps $M, N : B \to \mathbb{Z} \cup \{\infty\}$ such that $M(y) = d(u, y)$ and $N(y) = d(y, v)$ for all $y \in B$. Given these maps, we have $d(u, v) = \min_y(M(y) + N(y))$.

Single-source queries. The query from a node u consists of accessing the bags of $\mathrm{Tree}(G)$ via DFS, starting in the bag B_u. The algorithm maintains a map $d'_u(v)$ for all $v \in V$, initialized with $d_u(v) = \mathsf{LD}_{B_u}(u, v)$ for all $v \in B_u$, $d'_u(v) = \infty$ for all other v. Upon examining a bag B_v for some $v \in V$ for which $d'_u(v) = \infty$, it updates $d'_u(v) = \min_{x \in B_v}(d'_u(x) + \mathsf{LD}_{B_v}(x, v))$. Finally, it returns the map d'_u.

[1] This can be done using the algorithm in [36] which we currently use for LCA queries

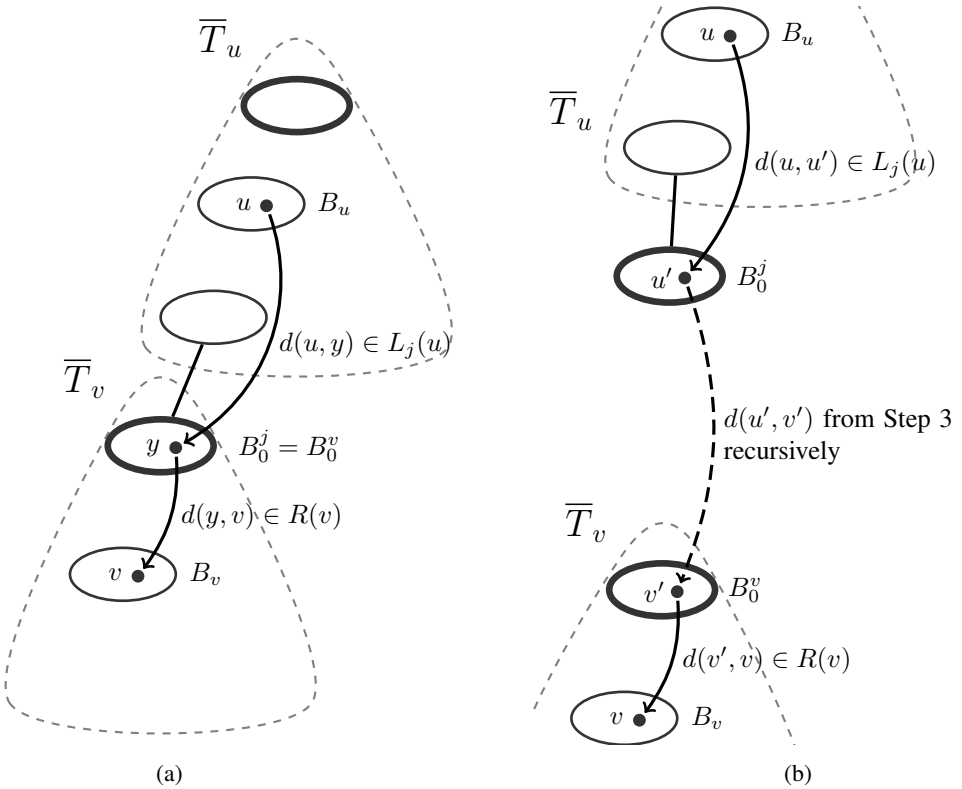


Figure 5: The two cases described in the query (u, v). Boldface bags are the root bags of their components. (a) If $\overline{T}_v$ is a child component of $\overline{T}_u$, the answer is retrieved from Step 2 recursion, by combining $L_j(u)$ and $R(v)$.(b) If $\overline{T}_v$ is not a child component of $\overline{T}_u$, the additional distances $d(u', v')$ are retrieved from Step 3 recursion.

Correctness. The preprocessing consists of summarizing distances along paths $B \rightsquigarrow B'$ of T, for all $u \in B$ and $v \in B'$, where B and B' are chosen conveniently to allow for fast queries. The correctness of the queries then follows directly from Lemma 3.

Remark 4. The algorithmic technique of distance summarization on trees has been developed in [3], and has been adapted on tree decompositions for the purpose of distance queries in [18], and more general semiring queries in [26]. However, our algorithm Distance utilizes our improved local distance computation (Section 3) and the notion of summary trees, and leads to a factor t improvement of all resource bounds w.r.t. the previous results, i.e. preprocessing time, space and query time.

6 Preprocessing Linear in n

In this section we describe a modification of the preprocessing described in Section 5 that reduces the preprocessing time to linear in n (i.e., $O\left(n \cdot t^2\right)$), at the expense of increasing the pair query time to $O(t^2 \cdot \alpha(n) \cdot \min(\log t, \alpha(n))$. We first describe the result for query time $O(t^2 \cdot \alpha^2(n))$, and then remark the required modifications for obtaining $O(t^2 \cdot \alpha(n) \cdot \min(\log t, \alpha(n))$ time.

Note that by using $\lambda = \alpha(n) - 1$ in the preprocessing phase of algorithm Distance of Section 5, we achieve preprocessing time $O\left(t^2 \cdot n \cdot \alpha^2(n)\right)$. Intuitively, the super-linear bound in n arises because of $O(\alpha(n))$ levels of recursion in Step 2, each one spawning $\alpha(n)$ recursions in Step 3, until the parameter λ becomes -1. In this section we modify Distance to obtain a data-structure called DistanceLP (linear preprocessing for distance) that slightly alters Step 2 to

remove the dependency on $\alpha^2(n)$. In this direction, we consider only the case where $t \leq \frac{n}{\alpha^2(n)}$. The preprocessing phase of DistanceLP is obtained by applying the following modifications to that of Distance.

1. Instead of Step 2, partition $\overline{T}$ into $O\left(\frac{n}{t \cdot \alpha^2(n)}\right)$ components of size $O\left(t \cdot \alpha^2(n)\right)$ each. Perform a summarization on $\overline{T}$, and apply Step 3 on the resulting summary tree $\overline{T}'$ for $\lambda = \alpha(n) - 1$.
2. Skip Step 4.

It follows easily from the analysis of Lemma 13 that the preprocessing of DistanceLP requires $O\left(n \cdot t^2\right)$ time and $O(n \cdot t)$ space.

Ancestor pair query. Given a query (u, v), where B_u is an ancestor of B_v, proceed as follows. If B_u and B_v belong to the same component $\overline{T}_i$ of the modified Step 2, the query is answered by performing the single-source distance search in the component $\overline{T}_i$ starting from B_u. This is run as the single-source query of algorithm Distance, with additionally restricting the DFS to the nodes that appear in $\overline{T}_i$. If B_u and B_v appear in different components $\overline{T}_u$ and $\overline{T}_v$ respectively, a single-source distance algorithm is executed in each one to determine the distances $d(u, u')$ and $d(v', v)$ for all $u' \in B_0^j$ and $v' \in B_0^v$, where B_0^j is the the root of the unique child component of $\overline{T}_u$ on the path $B_u \rightsquigarrow B_v$, and B_0^v is the root of $\overline{T}_v$. Then for all such u', v', the distances $d(u', v')$ are determined as in the algorithm Distance, from the recursion of Step 3, and $d(u, v)$ is then

$$d(u, v) = \min_{u' \in B_0^j, v' \in B_0^v} \left(d(u, u') + d(u', v') + d(v', v)\right).$$

The query time is then $O(\alpha^2(n) \cdot t^2 + \alpha(n) \cdot t^2) = O(\alpha^2(n) \cdot t^2)$, where the first term is for the single-source queries inside each component of size $O(\alpha^2(n) \cdot t)$, and the second term is for determining $d(u', v')$ for all described u', v', using at most $\alpha(n)$ levels of recursion of Step 3.

Pair query. The preprocessing and query phases for answering shortest path queries (u, v) where B_v is an ancestor of B_u is similar to that where B_u is ancestor of B_v. In order to handle general pair queries, additionally preprocess T to answer LCA queries in constant time. Let B_i be the LCA of B_u and B_v. Using the ancestor pair queries from above, we can compute the maps $M, N : B \to \mathbb{Z} \cup \{\infty\}$ such that $M(y) = d(u, y)$ and $N(y) = d(y, v)$ for all $y \in B$. Given these maps, we have $d(u, v) = \min_y(M(y) + N(y))$.

Correctness. The correctness of DistanceLP follows from the correctness of DistanceLP and Distance, and we thus obtain the following theorem.

Theorem 4. *Given a graph G of n nodes and treewidth t, let $\mathcal{T}(G)$ be the time and $\mathcal{S}(G)$ be the space required for constructing a nice tree-decomposition* $\mathrm{Tree}(G)$ *of $O(n)$ bags and width t, where $t \leq \frac{n}{\alpha^2(n)}$. The data-structure* DistanceLP *correctly answers distance queries on G and requires*

1. *$O(\mathcal{T}(G) + n \cdot t^2)$ preprocessing time;*
2. *$O(\mathcal{S}(G) + n \cdot t)$ space;*
3. *$O(t^2 \cdot \alpha^2(n))$ pair query time; and*
4. *$O(n \cdot t)$ single-source query time.*

Remark 5. Note that the data-structure DistanceLP requires $O(t^2 \cdot \alpha^2(n))$ pair query time compared to the query time of $O(t^4 \cdot \alpha(n))$ from [18], and thus, is slower when $t^2 \leq \alpha(n)$. To obtain an algorithm which is faster, even for such small t, in each component created in Step 2, we preprocess the component similarly to the algorithm in [3, Section 3] with linear preprocessing time in n' and logarithmic query time in n' where n' is the size of the component. By using the technique of local distance computation introduced in this paper, we obtain an $O(n' \cdot t^2)$ preprocessing time and $O(n' \cdot t)$ space for a component of size n' and query time of $O(\log(n') \cdot t^2)$. Since each component has size $O(\alpha^2(n) \cdot t)$ and there are $O(\frac{n}{\alpha^2(n) \cdot t})$ of them, this requires $O(n \cdot t^2)$ time for preprocessing, $O(t^2 \cdot \log t \cdot \log(\alpha^2(n))) = O(t^2 \cdot \log t \cdot \alpha(n))$ time for a pair query, and $O(n \cdot t)$ space. Thus we obtain bounds that are better than the previous preprocessing time and space, and pair query time [18].

7 Space vs Query Time Tradeoff for Sub-linear Space

The algorithm in Section 6 suggests a way to trade query time for space. Here we present the data-structure LowSpDis (for low space). The idea is to create sufficiently large components in the initial partitioning of $\overline{T}$, for which no preprocessing is done. Then, a summarization $\overline{T}'$ of $\overline{T}$ is of sufficiently small size to be preprocessed by DistanceLP. Answering a query (u, v) is handled similarly as in DistanceLP, but requires additional time for processing the components in which u and v appear (since they have not been preprocessed). The results of this section are stated in the following theorem and corollary.

Theorem 5. *Consider (1) a constant $\epsilon \in (0, 1]$; (2) a weighted graph $G = (V, E, \mathsf{wt})$ with n nodes; and (3) a nice tree-decomposition $\mathrm{Tree}(G)$ of G of width t, for $t \leq \frac{n}{\alpha^2(n)}$, that consists of $O(n)$ bags; and (4) for each bag $B \in \mathrm{Tree}(G)$ the number of bags in the subtree under B. The data-structure* LowSpDis *correctly answers pair distance queries on G and requires*

1. *$O(n \cdot t^2)$ preprocessing time;*
2. *$O(n^\epsilon \cdot t^2)$ working space; and*
3. *$O(n^{1-\epsilon} \cdot t^2 \cdot \alpha^2(n))$ pair query time*

In many cases it might not be reasonable to assume that so many things are part of the input tape as described in the remark. Alternately, if only a weighted graph is in the input we can still trade time for space, but only down to $\sqrt{n}$ space.

Corollary 2. *Let (1) a constant $\epsilon \in [1/2, 1]$; and (2) a weighted graph $G = (V, E, \mathsf{wt})$ with n nodes and of constant treewidth, be given. Then there exists an algorithm that correctly answers pair distance queries on G and requires*

1. *Polynomial in n preprocessing time;*
2. *$O(n^\epsilon)$ working space; and*
3. *$O(n^{1-\epsilon} \cdot \alpha^2(n))$ pair query time.*

We first describe modifications in the tree partitioning and local distance computation that allows LowSpDis to operate in the desired space bounds.

Tree partitioning: The algorithm LowSpTreePart**.** We first present the algorithm LowSpTreePart for computing a tree partitioning in little space. Given a nice tree-decomposition $T = \mathrm{Tree}(G)$ for a given graph G over n nodes and a number $j \leq n$ the algorithm LowSpTreePart partitions T into $O(\frac{n}{j})$ connected components forming a component tree, each containing $O(j)$ bags, in $O(n)$ time and $O(\frac{n}{j})$ extra space. The partitioning is, like in Section 5, based on a post-order DFS traversal of the tree. In the DFS traversal, the algorithm has a *component forest* (which forms a component tree at the end of the traversal), a set of trees, where each tree consists of bags, which are roots of components. Also, in the DFS traversal, the algorithm has a stack of triples (ℓ, c, L) where ℓ is a level, c is a number (of bags), and L is a list of trees in the component forest. Consider a step of the DFS traversal, in which the algorithm considers some bag B at level i:

1. First, the DFS traversal partitions the children of B recursively, because it is post-order.
2. Second, consider the (at most) two triples on the stack with level i. Let them be (i_ℓ, c_ℓ, L_ℓ) and (i_r, c_r, L_r) respectively if they exist.
3. If (i_k, c_k, L_k) was not in the stack for $k \in \{\ell, r\}$, then let $(i_k, c_k, L_k) = (i, 0, \emptyset)$.
4. Let $c' = c_\ell + c_r$ be the number of bags cut off below B and let c'' be the number of bags in $T(B)$.
5. If $c'' - c' \geq j$, then
   (a) Add B to the component forest, with each $B' \in L_k$, for $k \in \{\ell, r\}$ as children and remove B' from the forest.
   (b) Add $(i - 1, c'', (B))$ to the stack, where (B) is the list that contains only B.
6. Otherwise, if $c'' - c' < j$ and $c' > 0$, add $(i - 1, c', L_\ell \circ L_r)$ to the stack, where $L_\ell \circ L_r$ is the concatenation of the lists L_ℓ and L_r.
7. Otherwise, if $c'' - c' < j$ and $c' = 0$, add nothing to the stack.
8. Follow the post-order DFS traversal to the parent of B.

Lemma 14. *Given a number j, the algorithm* LowSpTreePart *computes a partitioning of T into $O(n/j)$ partitions each of size between j and $2 \cdot j$, except for one component, in time $O(n)$ and requiring $O(n/j)$ extra space.*

Proof. Observe that the number of edges in the component forest is at most the number of cuts, that is at most $O(n/j)$. Also, at all points, each triple (i, c, L) in the stack, is such that L is not the empty list and each component tree in the component forest is in precisely one triple of the stack. This shows that it contains at most $O(n/j)$ triples and the sum of the length of the lists used is also at most $O(n/j)$. Note that the total time used is $O(n)$ for the DFS traversal. □

The extended component $\mathrm{Ext}(C)$**.** In a partition with a component C, let component $\mathrm{Ext}(C)$ be the component C together with the bags which are the roots of the child-components of C in the component tree. Note that $\mathrm{Ext}(C)$ contains at most twice the number of bags of C, because the tree-decomposition T was nice (and thus binary).

Local root distance computation: The algorithm LowSpLocDis**.** Let $\epsilon > 0$ be a given constant, and let $A = n^\epsilon \cdot t^2$. Consider some component tree $\overline{T}$. We now describe how to recursively perform the local distance computation (as described in Section 3) of roots of components in $\overline{T}$ (and find negative cycles anywhere). The resulting algorithm will be called LowSpLocDis and will require $O(n \cdot t^2)$ time and $O(A)$ space. We will describe the computation on a (sub)-component, using recursion. Consider some partitioning of $\overline{T}$ and a component C and let K be the size of C. Let $\{B_1, B_2, \ldots B_j\}$ be the set of roots of components in $\mathrm{Ext}(C)$. We have two parts, each corresponding to a pass of the algorithm LocDis. The first pass is as follows:

Base case: If $K \cdot t \leq n^\epsilon$, execute the following steps:
- (a) Compute the first pass of local distance computation in $G[C]$, following LocDis.
- (b) Check if there is a negative cycle by testing if $d(u, u) < 0$ for some $u \in C$. If so terminate the recursion and return "Negative cycle".
- (c) Store the local distances $d(u, v)$ for $u, v \in B_i$ as a $((t + 1) \times (t + 1))$-matrix M_i in B_i for each i.
- (d) Discard everything, but the matrices M_i constructed in the previous step.

Recursive case: Otherwise, if $K \cdot t > n^\epsilon$, execute the following steps:

Partition step: Partition C up into $O(n^\epsilon)$ sub-components $\{C_1, C_2, \ldots, C_j\}$ forming a component tree $\widehat{T}$, such that each C_i has size $O(\frac{K}{n^\epsilon})$ using LowSpTreePart.
- (a) Consider repeatedly C_i, such that the first pass of local root distances has been computed for the roots of all children of C_i in $\widehat{T}$ (this is initially the case for the leaves).
  - i. Compute recursively the first pass of local root distances on $\mathrm{Ext}(C_i)$.
  - ii. Store the local distances $d(u, v)$ for $u, v \in B_i$ as a $(t \times t)$-matrix M_i in B_i for each i.
  - iii. Discard everything, but the matrices M_i constructed in the previous step.

The second pass is similar (the difference is that the access of sub-components is top-down instead of bottom-up and that we execute both passes instead of just the first) and formally as follows:

Base case: If $K \cdot t \leq n^\epsilon$, execute the following steps:
- (a) Compute both passes of local distance computation in $G[C]$, following LocDis.
- (b) Check if there is a negative cycle by testing if $d(u, u) < 0$ for some $u \in C$. If so terminate the recursion and return "Negative cycle".
- (c) Store the local distances $d(u, v)$ for $u, v \in B_i$ as a $(t \times t)$-matrix M_i in B_i for each i.
- (d) Discard everything, but the matrices M_i constructed in the previous step.

Recursive case: Otherwise, if $K \cdot t > n^\epsilon$, execute the following steps:

Partition step: Partition C up into $O(n^\epsilon)$ sub-components $\{C_1, C_2, \ldots, C_j\}$ forming a component tree $\widehat{T}$, such that each C_i has size $O(\frac{K}{n^\epsilon})$ using LowSpTreePart.
- (a) Consider repeatedly C_i, such that the first pass of local root distances has been computed for the parent of C_i in $\widehat{T}$ (this is initially the case for the root).
  - i. Compute recursively both passes of local root distances on $\mathrm{Ext}(C_i)$.
  - ii. Store the local distances $d(u, v)$ for $u, v \in B_i$ as a $(t \times t)$-matrix M_i in B_i for each i.
  - iii. Discard everything, but the matrices M_i constructed in the previous step.

Computing the local distances in the roots of the components $\overline{T}$ then consists of running the above two passes on T, where we partition according to $\overline{T}$ in each Partition step on T.

Lemma 15. *Given an $\epsilon > 0$, the algorithm* LowSpLocDis *requires $O(n \cdot t^2)$ time and $O(n^\epsilon \cdot t^2)$ space.*

Proof. Note that our algorithm for tree partitioning is deterministic and thus we always get the same partitioning when we recompute it. Also, notice that the second pass recursively calls both the second and the first pass on the sub-components, but the first pass only recursively calls the first. Since the depth of the recursion is $O(\frac{1}{\epsilon})$ there are at most $O(\frac{1}{\epsilon})$ recursive calls on a fixed component.

We will now prove the following claim:

Claim 1. *A given bag B of the tree decomposition is in at most 2 components at the lowest level.*

Proof. Consider a fixed sub-component C with root B'. Let $\{B_1, B_2, \ldots, B_j\}$ be the roots of components which are in $\mathrm{Ext}(C)$ but are not B'. We see that no B_i, for any i, will become the root of a sub-component (at any level of the recursion) in $\mathrm{Ext}(C)$. This is because B_i is a leaf in C, and each sub-component (at any level of the recursion) has size at least t (since $K \cdot t > A$, and thus $\frac{K}{n^\epsilon} \geq t$) and can therefore not be made out of a leaf alone. Thus, if a bag is a root of a sub-component at some level, but not the root of the whole tree at the start, then it is in 2 components at the lowest level, otherwise it is only in 1. □

The time to compute the two passes in the base case on a component C is $O(\widehat{n} \cdot t^2)$, where $\widehat{n}$ is the size of the component. Hence, the total time for the base case is $O(\frac{1}{\epsilon} \cdot n \cdot t^2)$, using the claim. Also, in the recursive case, we spend linear time (for the partitioning) and therefore use $O(\frac{1}{\epsilon} \cdot n)$ time for that in total on a fixed level and $O(\epsilon^{-2} \cdot n)$ time in total over all levels. Hence, overall we use $O(\frac{1}{\epsilon} \cdot n \cdot t^2 + \epsilon^{-2} \cdot n) = O(n \cdot t^2)$ time in total.

The space usage is $O(\frac{A}{\epsilon})$ because, whenever we are at the lowest level of recursion, we store a partitioning on each of the $\frac{1}{\epsilon}$ levels and such a partitioning requires $O(A)$ space (for the matrices in the roots of the $O(n^\epsilon)$ many components at that level). Furthermore, on the lowest level we use $O(\widehat{n} \cdot t) = O(A)$ space (because of our criteria for stopping the recursion), where the size of the component is $\widehat{n}$. □

Correctness. The correctness of the base case follows directly from Lemma 7 (which shows the correctness of LocDis). Note that instead of starting the first pass from the leaves and processing bottom-up in LocDis, it suffices to iterate over bags, such that all bags below the bag have already been processed by the first pass. This gives us the ordering used in LowSpLocDis on the components. Furtheremore, since LowSpLocDis do not recurse on components C, but on the extended component $\mathrm{Ext}(C)$, we see that all leaves of $\mathrm{Ext}(C)$ (which are either leaves of the tree decomposition or roots of some lower component) have either been processed by the first pass of LowSpLocDis in case they are roots of some lower components, or are leaves in the tree decomposition. It follows that the first pass of LowSpLocDis is correct. The correctness of the second pass is similar. We run both passes in the second pass because the matrices computed in the sub-components would otherwise be thrown away between passes. This give us the following lemma.

Lemma 16. *Given a constant $\epsilon > 0$, and a component tree $\overline{T}$, the algorithm* LowSpLocDis *finds a negative cycle if it exists in G, and otherwise computes the local distances for the roots of the components in $\overline{T}$ and in either case requires $O(n \cdot t^2)$ time and $O(n^\epsilon \cdot t^2)$ space.*

Similarly to LowSpLocDis one can also compute the root and leaf distances, see Section 5, in $O(n \cdot t^2)$ time and $O(n^\epsilon \cdot t^2)$ space.

The preprocessing of LowSpDis**.** We are now ready to describe the preprocessing as performed by LowSpDis. Let $\epsilon > 0$ be given and let $s = \max(n^{1-\epsilon} \cdot \alpha^2(n), t \cdot \alpha^2(n))$ (which is less than n by assumption on t). The preprocessing is as follows:

Step 1 Partition $\mathrm{Tree}(G)$ into $O(\frac{n}{s})$ components of size $O(s)$ each.

Step 2 For each root of each component apply the local distance computation algorithm LowSpLocDis, and construct the partitioned summary tree $\overline{T}$ where each bag corresponds to a root of a component.

Step 3 Preprocess $\overline{T}$ according to DistanceLP.

Pair querying in LowSpDis**.** To solve a query from u to v, let C_u be the component that contains B_u, and C_v be the component that contains B_v.

1. Test if $C_u = C_v$, by proceeding upward from B_u and B_v in the tree decomposition until the root of C_u and C_v are reached (the roots of components are marked).
2. If $C_u = C_v$, execute the following steps:
   (a) Find the LCA bag L of B_u and B_v using the tree decomposition together with the level of B_u and B_v.
   (b) Partition C_u into $O(s)$ partitions, such that B_u and B_v and L is the root of their corresponding subcomponent, using LowSpTreePart.
   (c) Compute local, root and leaf distances on C_u.
   (d) Use the root and leaf distance computation to compute $d(u, v) = \min_{w \in L} d(u, w) + d(w, v)$ and return.
3. Otherwise, if $C_u \neq C_v$, execute the following steps:
   (a) Let B_0^u (resp. B_0^v) be the root bag of C_u (resp. C_v).
   (b) Compute the LCA *component* L of B_0^u and B_0^v using a constant time LCA query on the component tree.
   (c) If $L = B_0^u$ execute the following steps:
      i. Find C'_v the last component on the path from B_0^v to L in T (using the algorithm for constant time LCA queries). Let B'_v be the root of C'_v.
      ii. Partition $\mathrm{Ext}(C_u)$ into $O(n^\epsilon)$ components such that B_0^u and B'_v are the root of their respective components.
      iii. Partition $\mathrm{Ext}(C_v)$ into $O(n^\epsilon)$ components such that B_0^v is the root of the component that contains it.
      iv. Find local, root and leaf distances in $\mathrm{Ext}(C_u)$ and $\mathrm{Ext}(C_v)$ based on the partitioning.
      v. Use the root and leaf distances to compute (1) $d(u, w_1)$, for each $w_1 \in B'_v$; and (2) $d(w_1, w_2)$ for each $w_1 \in B'_v$ and $w_2 \in B_0^v$ (this root and leaf distance computation was computed as a part of the preprocessing); and (3) $d(w_2, v)$, for each $w_2 \in B_0^v$.
      vi. Compute $d(u, v) = \min_{w_1 \in B'_v, w_2 \in B_0^v} d(u, w_1) + d(w_1, w_2) + d(w_2, v)$ and return.
   (d) If $L = B_0^v$ it is similar to the above.
   (e) If $B_0^v \neq L \neq B_0^u$ execute the following steps:
      i. Find C'_v (resp. C'_u) the last component on the path from B_0^v (resp. B_0^u) to L in T (using the algorithm for constant time LCA queries). Let B'_v (resp. B'_u) be the root of C'_v (resp. C'_u).
      ii. Partition $\mathrm{Ext}(L)$ into $O(n^\epsilon)$ components such that B'_u and B'_v are the root of their respective components.
      iii. Partition $\mathrm{Ext}(C_u)$ into $O(n^\epsilon)$ components such that B_0^u is the root of the component that contains it.
      iv. Partition $\mathrm{Ext}(C_v)$ into $O(n^\epsilon)$ components such that B_0^v is the root of the component that contains it.
      v. Find local, root and leaf distances in $\mathrm{Ext}(C_u)$ and $\mathrm{Ext}(C_v)$ and $\mathrm{Ext}(L)$ based on the partitioning.
      vi. Use the root and leaf distances to compute (1) $d(u, w_1)$, for each $w_1 \in B_0^u$; and (2) $d(w_1, w_2)$ for each $w_1 \in B_0^u$ and $w_2 \in B'_u$ (this root and leaf distance computation was computed as a part of the preprocessing); and (3) $d(w_2, w_3)$ for each $w_2 \in B'_u$ and B'_v using $\mathrm{Ext}(L)$; and (4) $d(w_3, w_4)$ for each $w_3 \in B'_v$ and $w_4 \in B_0^v$ (this root and leaf distance computation was computed as a part of the preprocessing); and (5) $d(w_4, v)$, for each $w_4 \in B_0^v$. Then inductively compute $d(u, w_{i+1}) = \min_{w_i} d(u, w_i) + d(w_i, w_{i+1})$ for each w_{i+1} (note that $d(u, w_1)$ is already computed).
      vii. Return $d(u, v) = \min_{w_4}(d(u, w_4) + d(w_4, v))$

In all cases, after the computation of some query, remove all the data-structures used.

Correctness. In each case of the algorithm we find the shortest path among paths of the form $w_0 = u \rightsquigarrow w_1 \rightsquigarrow w_2 \rightsquigarrow \cdots \rightsquigarrow v = w_k$, where each of w_i ranges over some bag B_i. It is easy to see that the bags B_i are bags on the path from B_u to B_v in T. We see that in any path from u to v there must be a node in each B_i following Lemma 2. Also, it is straightforward to see that we compute the distance between each pair w_i, w_{i+1} correctly for all i, using either pair queries from algorithm Distance in Section 5 or using LowSpLocDis. Finally, it is easy to see that we compute the

distance from u to v correctly given that we computed the distance between each pair w_i, w_{i+1} correctly.

Time and space requirements. It is clear that our preprocessing can be done as described in time $O(n \cdot t^2)$ and $O(n^\epsilon \cdot t)$ space. In regards to the query, we see that the local and root and leaf distance preprocessing (of which we do at most 4) requires $O(n^{1-\epsilon} \cdot t^2 \cdot \alpha^2(n))$ time (the size of each component times t^2) similarly to Lemma 4 and $O(n^\epsilon \cdot t^2)$ space, using the algorithms LowSpTreePart and LowSpLocDis. In the query we also use our data-structure (computed in the preprocessing) upto two times, which takes $O(t^2 \cdot \alpha^2(n))$ time each and then at the end we use $O(t^2)$ time to answer the query (we only find $O(t^2)$ edges and only need to consider certain paths of length at most 5). This then takes $O(n^{1-\epsilon} \cdot t^2 \cdot \alpha^2(n))$ time and $O(n^\epsilon \cdot t^2)$ space.

The above establish Theorem 5. The data-structure LowSpDis uses access to some precomputed tree decomposition of the graph, and the tree decomposition (and the graph) is not counted for the working space bound of LowSpDis. Since for constant treewidth graphs, the treewidth can be computed in deterministic logspace [22], LowSpDis combined with the logspace algorithm leads to Corollary 2.

Corollary 2. *Let (1) a constant $\epsilon \in [1/2, 1]$; and (2) a weighted graph $G = (V, E, \mathsf{wt})$ with n nodes and of constant treewidth, be given. Then there exists an algorithm that correctly answers pair distance queries on G and requires*

1. *Polynomial in n preprocessing time;*
2. *$O(n^\epsilon)$ working space; and*
3. *$O(n^{1-\epsilon} \cdot \alpha^2(n))$ pair query time.*

Proof. In our preprocessing, instead of having access to the tree decomposition in the input, we can use [22] to compute each bag of the tree decomposition in deterministic logarithmic space (and thus polynomial time) and then use that in LowSpDis to compute the summary trees (i.e., whenever a bag is required, it is recomputed using the logspace algorithm).

In the query from u to v we compute a tree decomposition of the components containing u and v in space and time linear in their size, using e.g. [12] and then otherwise proceed as LowSpDis. Since the size of the component is $O(n^{1-\epsilon}) \leq O(n^\epsilon)$ (recall that $\epsilon \geq \frac{1}{2}$), we get an data-structure using polynomial in n preprocessing time, but using $O(n^\epsilon)$ space and $O(n^{1-\epsilon} \cdot \alpha^2(n))$ pair query time. □

References

[1] T. Akiba, Y. Iwata, and Y. Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *SIGMOD'13*, SIGMOD '13, pages 349–360, 2013.

[2] T. Akiba, C. Sommer, and K. Kawarabayashi. Shortest-Path Queries for Complex Networks: Exploiting Low Tree-width Outside the Core. In *EDBT*, pages 144–155, 2012.

[3] N. Alon and B. Schieber. Optimal preprocessing for answering on-line product queries. Technical report, Tel Aviv University, 1987.

[4] S. Arnborg and A. Proskurowski. Linear time algorithms for NP-hard problems restricted to partial k-trees . *Discrete Applied Mathematics*, 23(1):11 – 24, 1989.

[5] R. Bauer, T. Columbus, I. Rutter, and D. Wagner. Search-space size in contraction hierarchies. In *ICALP 13*, pages 93–104, 2013.

[6] R. Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[7] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *LATIN 2000: Theoretical Informatics*. Springer Berlin Heidelberg, 2000.

[8] M. Bern, E. Lawler, and A. Wong. Linear-time computation of optimal subgraphs of decomposable graphs. *Journal of Algorithms*, 8(2):216 – 235, 1987.

[9] H. Bodlaender and T. Hagerup. Parallel algorithms with optimal speedup for bounded treewidth. volume 27, pages 1725–1746. 1995.

[10] H. L. Bodlaender. Dynamic programming on graphs with bounded treewidth. In *ICALP*, volume LNCS 317, pages 105–118. 1988.

[11] H. L. Bodlaender. A tourist guide through treewidth. *Acta Cybern.*, 11(1-2):1–21, 1993.

[12] H. L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25(6), Dec. 1996.

[13] H. L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theoretical Computer Science*, 209, 1998.

[14] H. L. Bodlaender. Discovering treewidth. In *SOFSEM'05*, volume LNCS 3381, pages 1–16. 2005.

[15] K. Chatterjee, A. K. Goharshady, R. Ibsen-Jensen, and A. Pavlogiannis. Algorithms for algebraic path properties in concurrent systems of constant treewidth components. In *POPL*, pages 733–747, 2016.

[16] K. Chatterjee, R. Ibsen-Jensen, P. Goyal, and A. Pavlogiannis. Faster algorithms for algebraic path properties in recursive state machines with constant treewidth. In *POPL*, 2015.

[17] K. Chatterjee, R. Ibsen-Jensen, and A. Pavlogiannis. Faster algorithms for quantitative verification in constant treewidth graphs. In *CAV*, 2015.

[18] S. Chaudhuri and C. D. Zaroliagis. Shortest Paths in Digraphs of Small Treewidth. Part I: Sequential Algorithms. *Algorithmica*, 27:212–226, 1995.

[19] T. Columbus. Search space size in contraction hierarchies. Master's thesis, Karlsruhe Institute of Technology, 2012.

[20] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction To Algorithms*. MIT Press, 2001.

[21] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[22] M. Elberfeld, A. Jakoby, and T. Tantau. Logspace versions of the theorems of Bodlaender and Courcelle. In *FOCS*, 2010.

[23] M. J. Fischer and A. R. Meyer. Boolean Matrix Multiplication and Transitive Closure. In *SWAT (FOCS)*, pages 129–131. IEEE Computer Society, 1971.

[24] R. W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962.

[25] L. R. Ford. Network Flow Theory. Report P-923, The Rand Corporation, 1956.

[26] T. Hagerup. Dynamic algorithms for graphs of bounded treewidth. *Algorithmica*, 27:292–315, 2000.

[27] R. Halin. S-functions for graphs. *Journal of Geometry*, 8(1-2):171–186, 1976.

[28] D. Harel and R. Tarjan. Fast Algorithms for Finding Nearest Common Ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.

[29] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[30] D. B. Johnson. Efficient Algorithms for Shortest Paths in Sparse Networks. *J. ACM*, 24(1):1–13, Jan. 1977.

[31] A. Maheshwari and N. Zeh. I/O-Efficient Algorithms for Graphs of Bounded Treewidth. *Algorithmica*, 54(3):413–469, 2009.

[32] E. F. Moore. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching, and Annals of the Computation Laboratory of Harvard University*, pages 285–292. Harvard University Press, 1959.

[33] L. R. Planken, M. M. de Weerdt, and R. P. van der Krogt. Computing all-pairs shortest paths by leveraging low treewidth. In *ICAPS-11*, pages 170–177. AAAI Press, 2011.

[34] N. Robertson and P. Seymour. Graph minors. III. planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49 – 64, 1984.

[35] B. Roy. Transitivité et connexité. *C. R. Acad. Sci. Paris*, 249:216–218, 1959.

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

[37] M. Thorup. All Structured Programs Have Small Tree Width and Good Register Allocation. *Information and Computation*, 142(2):159 – 181, 1998.

[38] S. Warshall. A Theorem on Boolean Matrices. *J. ACM*, 9(1):11–12, Jan. 1962.

[39] A. Yamaguchi, K. F. Aoki, and H. Mamitsuka. Graph complexity of chemical compounds in biological pathways. *Genome Informatics*, (14):376–377, 2003.

[40] Y. Yano, T. Akiba, Y. Iwata, and Y. Yoshida. Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths. In *CIKM'13*, pages 1601–1606, 2013.