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

Krishnendu Chatterjee[1], Rasmus Ibsen-Jensen[2], and Andreas Pavlogiannis[3]

1 **IST Austria, Klosterneuburg, Austria**
`kchatterjee@ist.ac.at`
1 **IST Austria, Klosterneuburg, Austria**
`ribsen@ist.ac.at`
1 **IST Austria, Klosterneuburg, Austria**
`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 $\epsilon \in [\frac{1}{2}, 1]$, we provide a data-structure with polynomial preprocessing time that allows pair queries in $O(n^{1-\epsilon} \cdot \alpha(n))$ time, where α is the inverse of the Ackermann function, and at all times uses $O(n^\epsilon)$ space. The input graph G is not considered in the space complexity.

1998 ACM Subject Classification G.2.2 Graph Theory: Graph algorithms

Keywords and phrases Graph algorithms; Constant-treewidth graphs; Reachability queries; Distance queries

Digital Object Identifier 10.4230/LIPIcs.ESA.2016.28

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 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) [30]. The treewidth of a graph is defined based on a *tree decomposition* of the graph [24], see Section 2 for a formal definition. Beyond the mathematical elegance of the treewidth property for graphs, there are many classes of graphs

* The research was partly supported by Austrian Science Fund (FWF) Grant No P23499-N23, FWF NFN Grant No S11407-N23 (RiSE/SHiNE) and ERC Start grant (279307: Graph Games).

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 [32]. Also many chemical compounds have treewidth 3 [34]. For many other applications see the surveys [11, 10]. 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 [3, 7, 8]). 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 [16]. The constant treewidth of control flow graphs has also been shown to lead to faster algorithms for interprocedural analysis [14], quantitative verification [15], and analysis of concurrent programs [13].

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) [26], Dijkstra's algorithm (single-source) [19], Bellman-Ford algorithm (single-source) [5, 23, 28], Floyd-Warshall algorithm (all pairs) [22, 33, 31], and Johnson's algorithm (all pairs) [27] and others for various special cases, there exist in essence only two different algorithmic ideas for reachability: Fast matrix multiplication (all pairs) [21] and DFS/BFS (single-source) [18].

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, 16, 29, 1, 4, 17]. 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 [35]. 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. 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.
2. Second, we present a space-time tradeoff data-structure that supports distance pair queries in G and given a number $\epsilon \in [\frac{1}{2}, 1]$. The weights of G come from the set of

Table 1 Data-structures for pair and single-source reachability queries, on a directed graph G with n nodes, m edges, and a treewidth t. The model of computation is the standard RAM model with wordsize $W = \Theta(\log n)$. Space usage refers to the total space used during the preprocessing and query phase. 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)	[35] b)
2	–	$O(\lceil n/\log n \rceil)$	$O(m)$	$O(m)$	DFS/BFS [18]
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 6

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

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 $O(1)$ and height h. The number ϵ can be any fixed number in $[\frac{1}{2}, 1]$ and $\alpha(n)$ is the inverse Ackermann function. Space usage refers to the total space used during the preprocessing and query phase. 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	$O(n^2)$	$O(n^2)$	$O(1)$	$O(n)$	[29] a)
2	$O(n)$	$O(n)$	$O(\alpha(n))$	$O(n)$	[16]
3	$O(n \cdot \log h)$	$O(n)$	$O(\log \log n)$	$O(n \cdot \log \log n)$ b)	[2]
4	$O(n \cdot \log^2 n)$	$O(n \cdot \log n)$	$O(\log n)$	$O(n \cdot \log n)$ b)	[1]
5	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(\log^2 n)$	$O(n \cdot \log^2 n)$ b)	[4, 17]
6	Not given	$O(n^{\epsilon} \cdot \log^2 n)$ c)	$O(n^{1-\epsilon} \cdot \log n)$	– d)	[2] e)
i	polynomial	$O(n^{\epsilon})$	$O(n^{1-\epsilon} \cdot \alpha(n))$	– d)	Theorem 13

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/supported 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. Also, note that [2] requires a tree-decomposition to be part of the input, whereas our data-structure only requires that the graph G is part of the input.

integers $\mathbb{Z}$, but we do not allow negative cycles. For constant-treewidth graphs, our data-structure requires (i) polynomial preprocessing time; (ii) $O(n^{\epsilon})$ working space; and (iii) $O(n^{1-\epsilon} \cdot \alpha(n))$ time for pair queries.

The graph G is considered part of the input, and is not counted towards the space complexity.

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

1. 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 $O(\log n)$. 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 ([35, 16]) however those algorithms follow different approaches, where word tricks do not seem to be applicable (at least not without significantly modifying the algorithms).
2. For distance queries, we devise a procedure for shrinking a tree-decomposition of size $O(n)$ to one of size $O(n^{1-\epsilon})$, by partitioning the tree-decomposition to components of

sufficient size. A key property of this partitioning is that each component has only a constant number of neighbor components. We show how this shrank tree-decomposition can be preprocessed for answering pair distance queries in the stated bounds.

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 $(x_1, \dots, x_k)$ 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 $P = (x_1, \dots, x_k)$ 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* $C \subseteq V_T$ of T is a set of nodes of T such that for every pair of nodes $u, v \in C$, the unique simple path P_u^v in T visits only nodes in 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: $V_T = \{B_1, \dots, B_b : \text{ for all } 1 \leq i \leq b.\ B_i \subseteq V\}$ 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). 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 $(C_i)_{1 \leq i \leq 3}$ 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 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).*

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 $P' : B_1, B_2, \dots, B_j$ be the unique simple path in T from B_1 to B_j. For each $i \in \{2, \dots, j\}$ 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 $i \in \{2, \dots, j\}$. 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 $P'' : B_1, \dots, B_{i-1}, B, B_i, \dots, B_j$ 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 $P' : B_1, B_2, \dots, B_j$ 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 $d(u, v) = \min_{(x_1, \dots, x_{j+1}) \in A} \sum_{i=1}^{j} d(x_i, x_{i+1})$.*

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 $i \in \{1, \dots, j\}$. It easily follows that $d(u, v) = \sum_{i=1}^{j} d(x_i, x_{i+1})$ with $x_1, \dots x_{j+1} \in A$. ◀

Small tree decompositions. A tree-decomposition $T = \mathrm{Tree}(G) = (V_T, E_T)$ is called *small* if $|V_T| = O(\frac{n}{t})$.

▶ **Lemma 4.** *Given a tree decomposition $\mathrm{Tree}(G)$ of G of width $O(t)$ and $O(n)$ bags, a small, binary tree decomposition $\mathrm{Tree}'(G)$ of width $O(t)$ can be constructed in $O(n \cdot t)$ time. Moreover, if $\mathrm{Tree}(G)$ is balanced, then so is $\mathrm{Tree}'(G)$.*

Proof. Let $k = O(t)$ be the width of $\mathrm{Tree}(G)$. The construction is achieved using the following steps.

1. Following the steps of [9, Lemma 2.4], we turn $\mathrm{Tree}(G)$ to a *smooth* tree-decomposition $T_1 = (V_1, E_1)$, which has the properties that (i) for every bag $B \in V_1$ we have $|B| = k+1$, and (ii) for every pair of bags $(B_1, B_2) \in E_1$ we have $|B_1 \cap B_2| = k$. The process of [9, Lemma 2.4] can be performed $O(n \cdot t)$ time and increases the height by at most a factor 2, hence if $\mathrm{Tree}(G)$ is balanced, T_1 is also balanced, and by [9, Lemma 2.5], we have $|V_1| = O(n)$.
2. We turn T_1 to a binary tree-decomposition $T_2 = (V_2, E_2)$, by a standard tree-binarization process [16, Fact 3], which increases the size and the height of T_2 by at most a factor 2.
3. We construct a tree-decomposition $T_3 = (V_3, E_3)$ by partitioning T_2 to disjoint connected components of size between $\frac{k}{2}$ and k each (the last component might have size less than $\frac{k}{2}$) and contracting each such component to a single bag in T_3. Since T_2 is smooth, the number of nodes in the union of the bags of each component is at most $2 \cdot k$. Hence the width of T_3 is $O(k)$. The partitioning is done as follows. We traverse T_2 bottom-up and group bags into components in a greedy way. In particular, given that the traversal is on a current bag B, we keep track of the number of bags i_B below B (not including B) that have not been grouped to a component yet. The first time we find $i_B \geq t$, let B' be the child of B with the largest number $i_{B'}$ among the children of B. We group B' and its ungrouped descendants into a new component C, and continue with the traversal. Observe that the size of C is $\frac{k}{2} \leq |C| < k$.
4. Finally, we construct $\mathrm{Tree}'(G)$ by turning T_3 to a binary tree-decomposition as in Step 2.

Note that all steps above require $O(n \cdot t)$ time. The desired result follows. ◀

▶ **Lemma 5** ([16]). *Given a weighted graph $G = (V, E, \mathsf{wt})$ of treewidth t and a tree-decomposition $T = (V_T, E_T)$ of G of width $O(t)$, we can compute for all bags $B \in V_T$ a* local distance map $\mathsf{LD}_B : B \times B \to \mathbb{Z}$ *with $\mathsf{LD}_B(u, v) = d(u, v)$ in total time $O(|V_T| \cdot t^3)$ and space $O(|V_T| \cdot t^2)$.*

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 3) uses so called "word tricks" heavily. We use constant-time LCA queries which also use word tricks [25, 6].

3 Optimal Reachability for Low-Treewidth Graphs

In this section we present algorithms for building and querying a data-structure $\mathsf{Reachability}$, which handles single-source and pair reachability queries over an input a graph G of n nodes and treewidth t. In particular, we establish the following.

▶ **Theorem 6.** *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 the standard RAM with wordsize $W = \Theta(\log n)$. The data-structure* $\mathsf{Reachability}$ *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 $\mathcal{S}(G) = O(n)$ ([12, Lemma 2]), and thus along with Theorem 6 we obtain the following corollary.

▶ **Corollary 7.** *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 small, balanced and binary 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 Lemma 5. 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, a sequence $(A^0, A^1, \dots A^k)$ will be denoted by $(A^i)_{0 \leq i \leq k}$. 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 $(\mathsf{F}_u^i)_{0 \leq i \leq \mathsf{Lv}(u)}$, $(\mathsf{T}_u^i)_{0 \leq i \leq \mathsf{Lv}(u)}$, 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 Lemma1 and Lemma 2), which informally states that reachability properties between nodes in distant bags will be captured transitively, through nodes in intermediate bags.

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 tree-decomposition $T = \text{Tree}(G)$ 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. Turn T to a small, balanced binary tree-decomposition of G of width $O(t)$, using Lemma 4.
2. Preprocess T to answer LCA queries in $O(1)$ time [25].
3. Compute the local distance map $\mathsf{LD}_B : B \times B \to \mathbb{Z}$ for every bag B w.r.t reachability, i.e., for any bag B and nodes $u, v \in B$, we have $\mathsf{LD}_B(u, v) = 1$ iff $(u, v) \in E^*$.
4. 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 (with multiplicities) a_B contained in all ancestors of B, and the number b_B of nodes in B.
5. 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.

6. 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).
7. For every node u initialize two sequences of bit sets $(\mathsf{T}^i_u)_{0\leq i\leq \mathsf{Lv}(u)}$, $(\mathsf{F}^i_u)_{0\leq i\leq \mathsf{Lv}(u)}$, and pack them into consecutive words. Each set T^i_u and F^i_u has size $b_{B^i_u}$, where B^i_u is the ancestor of B_u at level i.
8. Traverse T top-down, and for B the bag currently visited, for every node $x \in B$, maintain two sequences of bit sets $(\overline{\mathsf{T}}^i_x)_{0\leq i\leq \mathsf{Lv}(B)}$ and $(\overline{\mathsf{F}}^i_x)_{0\leq i\leq \mathsf{Lv}(B)}$. Each set $\overline{\mathsf{T}}^i_x$ and $\overline{\mathsf{F}}^i_x$ 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^B_w of $\overline{\mathsf{F}}^0_x$ (resp., $\overline{\mathsf{T}}^0_x$) 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 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^B_w-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 $(\overline{\mathsf{T}}^i_x)_i$ (resp., $(\overline{\mathsf{F}}^i_x)_i$). Now each set sequence $(\overline{\mathsf{T}}^i_x)_i$ and $(\overline{\mathsf{F}}^i_x)_i$ has size $a_B + b_B$.
   b. For each $u \in B$ whose root bag is B, initialize set sequences $(\overline{\mathsf{F}}^i_u)_i$ and $(\overline{\mathsf{T}}^i_u)_i$ with 0s of length $a_B + b_B$ each, and set the bit at position l^B_u of $\overline{\mathsf{F}}^{\mathsf{Lv}(B)}_u$ and $\overline{\mathsf{T}}^{\mathsf{Lv}(B)}_u$ to 1. For every $w \in B$ with $\mathsf{LD}_B(u, w) = 1$ (resp., $\mathsf{LD}_B(w, u) = 1$), insert $(\overline{\mathsf{F}}^i_w)_i$ to $(\overline{\mathsf{F}}^i_u)_i$ (resp., $(\overline{\mathsf{T}}^i_w)_i$ to $(\overline{\mathsf{T}}^i_u)_i$). Finally, set $(\mathsf{F}^i_u)_i$ equal to $(\overline{\mathsf{F}}^i_u)_i$ (resp., $(\mathsf{T}^i_u)_i$ equal to $(\overline{\mathsf{T}}^i_u)_i$).

Figure 1 illustrates the constructed sets on a small example.

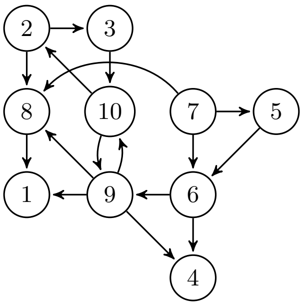
It is fairly straightforward that at the end of the preprocessing, the i-th position of each set F_u is 1 only if $(u, v) \in E^*$, where v is such that $i_v - i_u = i$. The following lemma states the opposite direction, namely that each such i-th position will be 1, as long as the path $P : u \rightsquigarrow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

▶ **Lemma 8.** *At the end of preprocessing, for every pair of nodes u and v with $i_u \leq i_v \leq i_u + s_u$, if there exists a path $P : u \rightsquigarrow v$ such that for every $w \in P$, we have $i_u \leq i_w \leq i_u + s_u$, 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 $i_x \leq i_v \leq i_x + s_x$ 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 $i_y \leq i_v \leq i_y + s_y$ 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 Lemma 5, it is $\mathsf{LD}_{B'}(x, y) = 1$. By the choice of x, y we have that B_x is an ancestor of B_y. Thus, by construction we have $i_x < i_y$ and $s_x \geq s_y + i_y - i_x$, and hence $i_x \leq i_v \leq i_x + s_x$. 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.

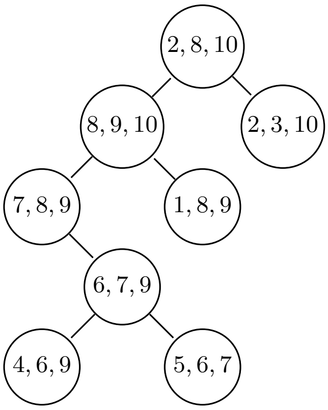
When B_u is examined, by the above claim there exists $x \in P$ such that $i_x \leq i_v$ and the $i_v - i_x$-th bit of F_x is 1. If $x = u$ we are done. Otherwise, by the choice of P, we have $i_u < i_x$, which can only happen if B_u is also the root bag of x. Then in step 5, F_x is inserted



(a)

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

(b)



(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 1 a, c: A graph G and a tree-decomposition $\mathrm{Tree}(G)$. b: 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 $i_v - i_u = j$. 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 $l_v^{B_6^i} = j$.

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. ◀

Similarly, given a node u and an ancestor bag B_u^i of B_u at level i, the j-th position of the set F_u^i (resp., T_u^i) is 1 only if $(u,v) \in E^*$ (resp., $(v,u) \in E^*$), where $v \in B_u^i$ is such that $l_v^{B_u^i} = j$. The following lemma states that the inverse is also true.

▶ **Lemma 9.** *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 .*

▶ **Lemma 10.** *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 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 balanced tree-decomposition $T = \mathrm{Tree}(G)$ of G in $\mathcal{T}(G)$ time and $\mathcal{S}(G)$ space. We establish the complexity of each preprocessing step separately.

1. Using Lemma 4, this step requires $O(n \cdot t)$ time. From this point on, T consists of $b = O(\frac{n}{t})$ bags, has height $h = O(\log n)$, and width $t' = O(t)$.
2. By a standard construction for balanced trees, preprocessing T to answer LCA queries in $O(1)$ time requires $O(b) = O(\frac{n}{t})$ time.
3. By Lemma 5, this step requires $O(b \cdot t'^3) = O(\frac{n}{t} \cdot t^3) = O(n \cdot t^2)$ time and $O(b \cdot t'^2) = O(\frac{n}{t} \cdot t^2) = O(n \cdot t)$ space.
4. 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 $O(b \cdot t') = O(n)$ time in total.
5–6. 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.
7. 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 $W = \Theta(\log n)$.
8. 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. We now turn our attention to the querying phase.

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. Each set starts in bit position a_B of the corresponding sequence $(\mathsf{F}_u^i)_i$ and $(\mathsf{T}_v^i)_i$. Return True iff the logical-AND of $\mathsf{F}_u^{\mathsf{Lv}(B)}$ 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 nodes v reached from u are those v for which the i_v-th bit of A is 1.

The following lemma establishes the correctness and complexity of the query phase.

▶ **Lemma 11.** *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. Let $t' = O(t)$ be the width of the small tree-decomposition constructed in Step 1. The correctness of the pair query comes immediately from Lemma 9 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 9, 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 8, 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 $W = \Theta(\log n)$. ◀

4 Space vs Query Time Tradeoff for Sub-linear Space

In this section we present the data-structure LowSpDis, for low-space distance queries. Our results make use of the following lemma.

▶ **Lemma 12** ([16]). *Consider a weighted graph* $G = (V, E, \mathsf{wt})$ *of* n *nodes and constant-treewidth, and a tree-decomposition* T *of* G *of* $O(n)$ *nodes and constant width be given. There exists a data-structure* DistanceLP *that answers distance queries on* G *and requires*

1. $O(n)$ *preprocessing time and space; and*
2. $O(\alpha(n))$ *pair query time.*

Throughout this section we fix a constant $\epsilon \in [\frac{1}{2}, 1]$. The main idea is to partition the initial tree-decomposition T to sufficiently large components, and discard all bags that don't appear in the boundary of their component. We use Lemma 12 to preprocess $\overline{T}$ and the induced graph. Answering a pair query (u, v) is performed similarly as in Lemma 12, but requires additional time for processing the components in which u and v appear (since they have not been preprocessed). The challenge comes in performing these computations within the targeted space and time bounds. We establish the following theorem.

▶ **Theorem 13.** *Let (1) a constant $\epsilon \in [\frac{1}{2}, 1]$; and (2) a weighted graph $G = (V, E, \mathsf{wt})$ with n nodes and of constant treewidth, be given. The data structure* LowSpDis *correctly answers pair distance queries on G and requires*
1. *Polynomial in n preprocessing time;*
2. *$O(n^{\epsilon})$ working space; and*
3. *$O(n^{1-\epsilon} \cdot \alpha(n))$ pair query time.*

▶ Remark. The data-structure LowSpDis accesses the graph in the input space, i.e., the graph and is not counted for the working space bound of LowSpDis.

Informal description. Here we outline the key steps required for LowSpDis to achieve the bounds stated in Theorem 13. The preprocessing consists of the following conceptual steps.
1. A binary tree-decomposition $T = \text{Tree}(G)$ of $O(n)$ bags is constructed in polynomial time and logarithmic space, using [20]. Hence, LowSpDis does not store T explicitly, but uses the logspace construction of [20] to traverse T and access its bags.
2. A tree-partitioning algorithm LowSpTreePart is used to partition T into $O(n^{1-\epsilon})$ components C of size $O(n^{\epsilon})$ each. A key point in this construction is that every such component C contains a constant number of bags on its boundary.
3. Given a list of components $\mathcal{C} = (C_1, \dots, C_\ell)$ constructed in the previous step, a tree of bags called *summary tree* $\overline{T}$ is constructed. The summary tree occurs by contracting every component C_i of T to a single bag $\mathcal{B}_i$. Moreover, $\mathcal{B}_i$ contains precisely the nodes that appear in the bags of the boundary of C_i. Since there are $O(1)$ such bags for every component, each $\mathcal{B}_i$ has constant size. The key point in this step is that $\overline{T}$ is a tree-decomposition of G restricted on the nodes that appear in bags of $\overline{T}$. Moreover, $\overline{T}$ has size $O(n^{1-\epsilon})$ instead of $O(n)$, which is the size of the initial tree-decomposition T.
4. Since $\overline{T}$ is a tree-decomposition, Lemma 12 applies to preprocess $\overline{T}$ in the stated bounds.
5. An algorithm LowSpLD is used to compute the distance $d(u, v)$ between any pair of nodes u, v that appear together in some boundary bag of a component C_i. This is achieved by traversing T in a particular way, and applying a standard, linear-space computation on each component C_i separately. Since $|C_i| = O(n^{\epsilon})$, this requires $O(n^{\epsilon})$ space. Since the boundary bags of C_i are constantly many, the algorithm only needs to store constant-size information per component, and thus $O(n^{1-\epsilon}) = O(n^{\epsilon})$ information in total.
6. Finally, given a node u, it is crucial to obtain the set V_u of nodes that u can reach going through nodes v that appear in bags of $\overline{T}$. Moreover, this set needs to be obtained in linear time in the size of the component, i.e., $O(n^{1-\epsilon})$. This is achieved by a graph traversal on G starting from u, in combination with perfect hashing for testing in $O(1)$ time whether a node v appears in bags of $\overline{T}$.

A query u, v is answered by LowSpDis using the following conceptual steps.
1. First, the algorithm retrieves the sets V_u and V_v. If $v \in V_u$, then the distance $d(u, v)$ is retrieved by constructing a tree-decomposition T_u of $G[V_u]$, and using standard methods for solving the problem in T_u, in $O(n^{\epsilon})$ time. Similarly if $u \in V_v$.
2. If $v \notin V_u$ and $u \notin V_v$, then the algorithm again constructs the tree-decompositions T_u and T_v of $G[V_u]$ and $G[V_v]$ respectively. The algorithm retrieves two bags $\mathcal{B}_u$ and $\mathcal{B}_v$ of $\overline{T}$ with $\mathcal{B}_u \subseteq V_u$ and $\mathcal{B}_v \subseteq V_v$, and uses the standard methods of the previous item to obtain the distances $d(u, x)$ and $d(y, v)$, for every node $x \in \mathcal{B}_u$ and $\mathcal{B}_v$. Additionally, the

algorithm uses Lemma 12 to obtain the distance $d(x, y)$ between every such pair x, y. Finally, the algorithm returns the value $\min_{x\in\mathcal{B}_u, y\in\mathcal{B}_v}(d(u, x) + d(x, y) + d(y, v))$.

In the remaining of this section we describe in detail the above phases of LowSpDis.

Tree partitioning: The algorithm LowSpTreePart. We first describe algorithm LowSpTreePart, which operates on a binary tree-decomposition $T = (V_T, E_T)$ of $O(n)$ bags. Given a constant ϵ, LowSpTreePart splits T to $O(n^{1-\epsilon})$ connected components $C \subseteq V_T$ of size $|C| = O(n^\epsilon)$. Each component C is implicitly represented as a list of bags $C(B_1, \dots, B_k)$, which mark the boundaries of C in T. The *root* of $C(B_1, \dots, B_k)$ is $B = \arg\min_{B_i} \mathsf{Lv}(B_i)$, i.e., the smallest-level bag among all B_i. We will consider w.l.o.g. that B_1 is always the root bag of component $C(B_1, \dots, B_k)$. A bag B' belongs to C iff the $\mathsf{Lv}(B') \geq \mathsf{Lv}(B_1)$ and the unique simple path $B \rightsquigarrow B_1$ in T does not contain any of the B_i as intermediate bags.

The algorithm traverses T in post-order, and maintains a two variables $x, y \in \mathbb{N}$, that represent the size of the current component C and the number of components that appear directly below C. As the algorithm backtracks to a bag B, it updates $x = x_1 + x_2 + 1$ and $y = y_1 + y_2$, where x_i, y_i is the pair corresponding to the child B'_i of B (recall that T is binary), or sets $x = x_1 + 1$ and $y = y_1$ if B has only one child B'_1. If $x \geq n^\epsilon$ or $y \geq 3$, the algorithm creates a new component $C(B_1, \dots, B_k)$, where B_1 is the current bag B, and $B_2, \dots, B_k$ are parents of roots of components that have been constructed already (or leaves of T). Finally, the algorithm sets $x = 0$ and $y = 1$, and proceeds to the parent of B.

▶ **Lemma 14.** LowSpTreePart *constructs* $O(n^{1-\epsilon})$ *components. For every constructed component* $C(B_1, \dots, B_k)$ *we have* $|C| \leq 2 \cdot n^\epsilon - 1$ *and* $k \leq 5$.

Proof. If $|C| > 2 \cdot n^\epsilon - 1$, then, before backtracking to B_1, the algorithm examined a child B of B_1 with value $x \geq j$, and thus would have grouped B and B_1 in different components. It is easy to see that every root of a component appears in the same component with its children, a contradiction. A similar argument holds for showing that $k \leq 5$. We now argue that LowSpTreePart constructs $O(n^{1-\epsilon})$ components. We say that the algorithm "performs a type A cut" and "performs a type B cut" when it constructs a component based on the criterion $x \geq j$ and $y \geq 3$ respectively. Let X and Y be the number of type A and type B cuts. Every type A cut constructs a component of size at least j, hence $X = O(n^{1-\epsilon})$. Additionally, we have $Y \leq X$, hence $X + Y = O(n^{1-\epsilon})$, as desired. To see that $Y \leq X$, let Z be a counter that counts the sum of the y values that LowSpTreePart maintains at any point in the traversal. Observe that a type A cut increases Z by at most one, and a type B cut decreases Z by at least one. Since Z is always non-negative, we have that there is at least one type A cut for each type B cut, thus $Y \leq X$. The desired result follows. ◀

We denote by $\mathsf{Root}(C)$ the root bag of a component C. Given two components C_1, C_2 constructed by LowSpTreePart, we say that C_1 is the *parent* of C_2 if $\mathsf{Root}(C_1)$ is the lowest ancestor of $\mathsf{Root}(C_2)$ among all bags that appear as roots in some component. In such case, C_2 is a *child* of C_1. Given a component C that is the parent of components $C_1, \dots, C_i$, we let $\mathsf{Merge}(C) = C \cup \bigcup_j C_j$.

The summary tree construction SummaryTree**.** Let $\mathcal{C} = (C_1, \dots, C_\ell) = \mathsf{LowSpTreePart}(T)$ be the list of components that LowSpTreePart returns, where each component is implicitly represented by the bags of its boundary, i.e., $C_i = C_i(B^i_1, \dots, B^i_{k_i})$. We construct a *summary tree* of bags $\overline{T} = \mathsf{SummaryTree}(\mathcal{C}) = (\overline{V}, \overline{E})$ as follows.

1. $\overline{V}$ consists of bags $\mathcal{B}_i$ for $1 \leq i \leq \ell$, where $\mathcal{B}_i = B^i_1 \cup \cdots \cup B^i_{k_i}$, i.e., $\mathcal{B}_i$ is the union of all bags in the boundary of C_i.
2. We have $(\mathcal{B}_i, \mathcal{B}_j) \in \overline{E}$ if C_i is a parent of C_j.

The following lemma follows easily from Lemma 14 and the above construction.

▶ **Lemma 15.** *Let $V_S = \bigcup_{\mathcal{B}_i \in \overline{V}} \mathcal{B}_i$ be the set of nodes of G that appear in bags of the summary tree $\overline{T}$. Then $\overline{T}$ is a tree-decomposition of the graph $G[V_S]$ induced by V_S. $\overline{T}$ has $O(n^{1-\epsilon})$ bags and constant width.*

Local distance computation in low space LowSpLD. Let $\mathcal{C} = (C_1, \dots, C_\ell) = \mathsf{LowSpTreePart}(T)$ be the list of components constructed by LowSpTreePart. We describe algorithm LowSpLD, which computes the distance $d(u, v)$ between any pair of nodes u, v that appear in the root bag $\mathsf{Root}(C_i)$ of some component C_i. Let $T_i = \mathrm{Tree}(G)[\mathsf{Merge}(C_i)]$ be the subtree of $\mathrm{Tree}(G)$ restricted in the bags of component C_i and its children components, and $V_i = \bigcup_{B \in \mathsf{Merge}(C_i)} B$ the set of nodes that appear in bags of $\mathsf{Merge}(C_i)$. It is easy to verify that T_i is a subtree of T, and thus a tree decomposition of the graph $G[V_i] = (V_i, E_i)$ induced by V_i. The algorithm LowSpLD operates as follows. For every component C, it maintains a local distance map $\mathsf{LD}_{\mathsf{Root}(C)} : \mathsf{Root}(C) \times \mathsf{Root}(C) \to \mathbb{Z}$. Initially, $\mathsf{LD}_{\mathsf{Root}(C)}(u, v) = \mathsf{wt}(u, v)$ for every component C and pair of nodes $u, v \in \mathsf{Root}(C)$. Then, LowSpLD performs the following two passes.

1. Traverse $\overline{T}$ bottom-up, and for every encountered bag $\mathcal{B}$ that corresponds to component C, let $C_1, \dots, C_k$ be the children components of C. Obtain the tree-decomposition T_i, and construct a weight function $\mathsf{wt}_i : E_i \to \mathbb{Z}$ defined as follows:
$$\mathsf{wt}_i(u, v) = \begin{cases} \mathsf{LD}_{\mathsf{Root}(C)}(u, v) & \text{if } u, v \in \mathsf{Root}(C) \\ \mathsf{LD}_{\mathsf{Root}(C_i)}(u, v) & \text{if } u, v \in \mathsf{Root}(C_i) \text{ for some } 1 \leq i \leq k \\ \mathsf{wt}(u, v) & \text{otherwise} \end{cases}$$
and execute the local distance computation of Lemma 5 Afterwards, update $\mathsf{LD}_{\mathsf{Root}(C)}$ and $\mathsf{LD}_{\mathsf{Root}(C_i)}$ for all $1 \leq i \leq k$ with the newly discovered distances.
2. Traverse $\overline{T}$ top-down, and for every encountered bag $\mathcal{B}$ execute the steps of Step 1.

▶ **Lemma 16.** *At the end of* LowSpLD*, for every component C and nodes $u, v \in \mathsf{Root}(C)$ we have $\mathsf{LD}_{\mathsf{Root}(C)}(u, v) = d(u, v)$. Moreover,* LowSpLD *operates in $O(n^\epsilon)$ space and polynomial time.*

Proof. The correctness of LowSpLD follows straightforwardly from Lemma 5 and Lemma 3. Since T has constant width, the size of each local distance map $\mathsf{LD}_{\mathsf{Root}(C)}$ has constant size. Hence the space used by the algorithm is asymptotically the space required for storing $\overline{T}$, plus the space for constructing each tree-decomposition T_i. By Lemma 15 the former requires $O(n^{1-\epsilon})$ space, while by Lemma 14 the latter $O(n^\epsilon)$ space. Since $\epsilon \geq \frac{1}{2}$, we conclude that the space usage is $O(n^\epsilon)$. The polynomial time bound follows from the space bound. ◀

Fast component retrieval GetCompNodes. Given a node u of G, we are interested in retrieving the set V_u of nodes that u can reach in G without going through nodes v that appear in bags of $\overline{T}$. The desired set V_u can be obtained in $O(n^\epsilon)$ time by a performing any standard graph traversal on G starting from u, and making sure that the traversal never expands a node v that appears in the bags of $\overline{T}$. This can be done if testing whether v appears in any of the bags of $\overline{T}$ can be performed in constant time. Let $V_S = \bigcup_{\mathcal{B}_i \in \overline{V}} \mathcal{B}_i$ be the set of all such nodes, and $k = |V_S| = O(n^{1-\epsilon})$. We cannot store V_S as a standard bit-set which allows $O(1)$ membership testing, as this would require linear space (i.e., beyond our space bound $O(n^{\epsilon})$). The problem can be solved using standard techniques from perfect hashing to store the set V_S. In the query phase, given a node u, GetCompNodes detects that $u \in V_S$ by testing whether u equals its entry in the hash table.

LowSpDis Preprocessing. We now describe the preprocessing phase of LowSpDis. The input is a weighted graph $G = (V, E, \mathsf{wt})$ of constant treewidth, and a constant $\epsilon \in [\frac{1}{2}, 1]$.

1. Construct a binary tree-decomposition $T = \text{Tree}(G)$ in logspace [20].
2. Use LowSpTreePart to construct a list of components $\mathcal{C} = (C_1, \dots, C_\ell) =$ LowSpTreePart(T), with $\ell = n^{1-\epsilon}$ (i.e., LowSpTreePart is executed with $j = n^{\epsilon}$).
3. Construct the local distance maps $\mathsf{LD}_{\mathsf{Root}(C)}$ using LowSpLD.
4. Construct the summary tree $\overline{T} = \mathsf{SummaryTree}(\mathcal{C}) = (\overline{V}, \overline{E})$. For every component C_i that corresponds to $\mathcal{B}_i$ in $\overline{T}$, find a node $z \notin \mathcal{B}_i$ that appears in bags of C_i, and associate z with $\mathcal{B}_i$.
5. Use Lemma 12 to build a data-structure DistanceLP on $G[V_S]$ and $\overline{T}$.
6. Let $V_S = \bigcup_{\mathcal{B}_i \in \overline{V}} \mathcal{B}_i$ be the set of nodes of G that appear in bags of the summary tree $\overline{T}$. Construct the data-structure GetCompNodes on V_S.

LowSpDis Querying. We now turn our attention to the query phase of LowSpDis.

1. Use the data-structure GetCompNodes to construct the sets V_u and V_v.
2. Construct the tree-decompositions T_u and T_v of the graphs $G[V_u]$ and $G[V_v]$ induced by V_u and V_v. This is done using some standard linear-time algorithm, e.g. [12, Lemma 2]. If $u \in V_v$, insert u to every bag of T_v, and use Lemma 5 to obtain the distance $d(u, v)$. Similarly if $v \in V_u$.
3. If $u \notin V_v$ and $v \notin V_u$ let $\mathcal{B}_u$ be the unique bag of $\overline{T}$ with that is associated with a node $z_u \in V_u$, and $\mathcal{B}_v$ the unique bag of $\overline{T}$ that is associated with a node $z_v \in V_v$. Insert every node of $\mathcal{B}_u$ in every bag of T_u, and every node of $\mathcal{B}_v$ in every bag of T_u, and use Lemma 5 to obtain the distances $d(u, x)$ and $d(y, v)$ for every node $x \in \mathcal{B}_u$ and $y \in \mathcal{B}_v$. Return the value $\min_{x \in \mathcal{B}_u, y \in \mathcal{B}_v}(d(u, x) + d(x, y) + d(y, v))$ where for every pair x, y the distance $d(x, y)$ is obtained by querying DistanceLP.

Proof of Theorem 13. It is clear from Lemma 12, Lemma 14, Lemma 15 and Lemma 16 that the preprocessing of LowSpDis requires polynomial time and $O(n^{\epsilon})$ space, where $\epsilon \geq \frac{1}{2}$. In the query phase, LowSpDis uses $O(n^{\epsilon})$ time and space for extracting the sets V_u and V_v, since each has size $O(n^{\epsilon})$. Using a linear time and space algorithm for constructing the tree-decompositions T_u and T_v, this step also requires $O(n^{1-\epsilon})$ time and space. If $u \in V_v$ or $v \in V_u$, applying Lemma 5 on T_u and T_v is also done in $O(n^{1-\epsilon})$ time and space.

If $u \notin V_v$ and $v \notin V_u$, note that by Lemma 15 $\mathcal{B}_u$ and $\mathcal{B}_v$ have constant size, hence after inserting every node of $\mathcal{B}_u$ to every bag of T_u and every node of $\mathcal{B}_v$ to every bag of T_v, T_u and T_v still have constant width. Hence all distances $d(u, x)$ and $d(v, y)$ can be obtained using Lemma 5 in $O(n^{1-\epsilon})$ time and space. Finally, DistanceLP will be queried for the distances $d(x, y)$ of a constant number of pairs x, y, and by Lemma 12, all such queries can be served in $O(n^{1-\epsilon} \cdot \alpha(n))$ time. ◀

References

1 Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *SIGMOD'13*, SIGMOD'13, pages 349–360, 2013.

2 Takuya Akiba, Christian Sommer, and Ken-ichi Kawarabayashi. Shortest-Path Queries for Complex Networks: Exploiting Low Tree-width Outside the Core. In *EDBT*, pages 144–155, 2012.

3 Stefan Arnborg and Andrzej Proskurowski. Linear time algorithms for NP-hard problems restricted to partial k-trees . *Discrete Applied Mathematics*, 23(1):11–24, 1989.

4 Reinhard Bauer, Tobias Columbus, Ignaz Rutter, and Dorothea Wagner. Search-space size in contraction hierarchies. In *ICALP 13*, pages 93–104, 2013.

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

6 Michael A. Bender and Martín Farach-Colton. The LCA problem revisited. In *LATIN 2000: Theoretical Informatics*. Springer Berlin Heidelberg, 2000.

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

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

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

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

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

12 HansL. Bodlaender and Torben Hagerup. Parallel algorithms with optimal speedup for bounded treewidth. *SIAM Journal on Computing*, 27:1725–1746, 1995.

13 Krishnendu Chatterjee, Amir Kafshdar Goharshady, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis. Algorithms for algebraic path properties in concurrent systems of constant treewidth components. In *POPL*, pages 733–747, 2016.

14 Krishnendu Chatterjee, Rasmus Ibsen-Jensen, Prateesh Goyal, and Andreas Pavlogiannis. Faster algorithms for algebraic path properties in recursive state machines with constant treewidth. In *POPL*, 2015.

15 Krishnendu Chatterjee, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis. Faster algorithms for quantitative verification in constant treewidth graphs. In *CAV*, 2015.

16 Shiva Chaudhuri and Christos D. Zaroliagis. Shortest Paths in Digraphs of Small Treewidth. Part I: Sequential Algorithms. *Algorithmica*, 27:212–226, 1995.

17 Tobias Columbus. Search space size in contraction hierarchies. Master's thesis, Karlsruhe Institute of Technology, 2012.

18 T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction To Algorithms*. MIT Press, 2001.

19 Edsger. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

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

21 Michael J. Fischer and Albert R. Meyer. Boolean Matrix Multiplication and Transitive Closure. In *SWAT (FOCS)*, pages 129–131. IEEE Computer Society, 1971.

22 Robert W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962.

23 Lester R. Ford. Network Flow Theory. Report P-923, The Rand Corporation, 1956.

24 Rudolf Halin. S-functions for graphs. *Journal of Geometry*, 8(1-2):171–186, 1976.

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

26 Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. on Systems Science and Cybernetics*, 4(2):100–107, 1968.

27 Donald B. Johnson. Efficient Algorithms for Shortest Paths in Sparse Networks. *J. ACM*, 24(1):1–13, January 1977.

28 Edward 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.

29 Leon R. Planken, Mathijs M. de Weerdt, and Roman P.J. van der Krogt. Computing all-pairs shortest paths by leveraging low treewidth. In *ICAPS-11*, pages 170–177. AAAI Press, 2011.

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

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

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

33 Stephen Warshall. A Theorem on Boolean Matrices. *J. ACM*, 9(1):11–12, January 1962.

34 Atsuko Yamaguchi, Kiyoko F. Aoki, and Hiroshi Mamitsuka. Graph complexity of chemical compounds in biological pathways. *Genome Informatics*, 14:376–377, 2003.

35 Yosuke Yano, Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths. In *CIKM'13*, pages 1601–1606, 2013.