Optimal Tree-decomposition Balancing and Reachability on Low Treewidth Graphs

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

[†] IST Austria

Abstract. We consider graphs with n nodes together with their tree-decomposition that has $b = O(n)$ bags and width t, on the standard RAM computational model with wordsize $W = \Theta(\log n)$. Our contributions are two-fold: Our first contribution is an algorithm that given a graph and its tree-decomposition as input, computes a binary and balanced tree-decomposition of width at most $4 \cdot t + 3$ of the graph in $O(b)$ time and space, improving a long-standing (from 1992) bound of $O(n \cdot \log n)$ time for constant treewidth graphs. Our second contribution is on reachability queries for low treewidth graphs. We build on our tree-balancing algorithm and present a data-structure for graph reachability that requires $O(n \cdot t^2)$ preprocessing time, $O(n \cdot t)$ space, and $O(\lceil t/\log n\rceil)$ time for pair queries, and $O(n \cdot t \cdot \log t/\log n)$ time for single-source queries. For constant t our data-structure uses $O(n)$ time for preprocessing, $O(1)$ time for pair queries, and $O(n/\log n)$ time for single-source queries. This is (asymptotically) *optimal* and is *faster than DFS/BFS* when answering more than a constant number of single-source queries.

Keywords: *Graph algorithms; Low treewidth graphs; Reachability; Balanced tree-decomposition.*

1 Introduction

In this work we consider two graph algorithmic problems where the input is a graph G with n nodes and a tree-decomposition $\text{Tree}(G)$ of G with $b = O(n)$ bags and width t. In the first problem we consider the computation of a binary and balanced (i.e., of height $O(\log n)$) tree-decomposition of G with width $O(t)$. In the second problem we present a data-structure to support reachability queries (pair and single-source queries) on G. We consider the computation on a 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) [27]. The treewidth of a graph is defined based on a *tree-decomposition* of the graph [20], 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 [30]. Also many chemical compounds have treewidth 3 [32]. For many other applications see the surveys [9,7]. 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 [2,5,8]). 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 [12].

Tree-decomposition balancing. Tree-decomposition balancing is an important problem for low treewidth graphs. For several problems over trees, it is much simpler to work on a balanced binary tree for algorithmic purposes (such as, searching, indexing problems). This general phenomenon is also true for tree-decompositions, e.g., balanced tree-decompositions are used to solve MSO (Monadic Second Order Logic) queries in log-space [15] as well as in other circuit complexity classes [16]. Besides theoretical complexity, the balanced tree-decomposition is also useful in practical problems such as recursive control-flow graphs with constant treewidth [30], and it was shown that reachability queries can be answered in time proportional to the height of the tree-decomposition [11]. We state the previous results for computing balanced tree-decomposition only for constant treewidth graphs. There exists two known algorithms for the problem: (1) an algorithm by [26] with running time $O(n \cdot \log n)$ and $O(n)$ space; and (2) an algorithm by [15] that requires $n^{O(1)}$ time[1] and $O(\log n)$ space. In either case, the dependency on the treewidth is exponential, and a binary and balanced tree-decomposition is constructed directly without any tree-decomposition being given (in contrast we consider that a tree-decomposition is given as input).

Reachability/distance problems. The *pair* reachability (resp. distance[2]) 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) [21], Dijkstra's algorithm (single-source) [14], Bellman-Ford algorithm (single-source) [3,19,24], Floyd-Warshall algorithm (all pairs) [18,31,28], and Johnson's algorithm (all pairs) [22] and others for various special cases, there exist in essence only two different algorithmic ideas for reachability: Fast matrix multiplication (all pairs) [17] and DFS/BFS (single-source) [13].

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 algorithms have been presented [1,12,25]. The previous results are incomparable, in the sense that the best algorithm depends on the treewidth and the number of queries. Despite the many results for constant (or low) treewidth graphs, none of the previous results improves the complexity for the basic reachability problem, i.e., the bound for DFS/BFS has not been improved in any of the previous works.

---

[1] the constant in the exponent depends on the treewidth

[2] we mention the distance problem here, because most previous papers for low treewidth graphs have focused on the distance problem instead of the reachability problem.

Our results. In this work, we present algorithms with optimal time complexity when used on constant treewidth graphs. For our problems the input is a graph G with n nodes and a tree-decomposition of width t and $O(n)$ bags. Our main contributions are as follows:

1. Our contribution is an algorithm that computes a binary and balanced tree-decomposition of G with width at most $4 \cdot t + 3$ in $O(n)$ time and space. We remark that improved algorithms for construction of tree-decompositions is still an active research area, see for instance [10]. An important strength of our algorithm is that it can use any tree-decomposition algorithm as a preprocessing step to compute an initial tree-decomposition, and then computes a binary and balanced approximate tree-decomposition using linear additional time and space. Given an input tree-decomposition of width t the output binary and balanced tree-decomposition of our algorithm has width at most $4 \cdot t + 3$ (i.e., the width increases by a constant factor), and we also present an example family of graphs where the treewidth is $2 \cdot t - 1$ but any balanced tree-decomposition must have width at least $3 \cdot t - 1$ (i.e., some constant factor increase is unavoidable in general).

2. Our second 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 (vi) $O(n \cdot t \cdot \log t/\log n)$ 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 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 algorithm 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 the so-called word-tricks, to the best of our knowledge, DFS/BFS have not been made faster using word-tricks. Table 1 presents a comparison of our results with DFS/BFS for general, and constant treewidth graphs.

While our main contributions are theoretical, we have implemented our data-structure for reachability and applied it on examples of constant treewidth graphs that arise in practice as control-flow graphs of various programs. In Appendix A we provide a comparison table which shows that the single-source query time of the data-structure is less than that of DFS/BFS, even for relatively small graphs.

Important techniques. Our improvements are achieved by introducing the following new techniques.

1. *(Balancing separator).* We provide a data-structure that accepts queries of the following form: We fix a tree-decomposition of a graph G, and the input to the queries is a connected component $\mathcal{C}$ of c bags defined by k edge-cuts, and requires the output to be a bag B such that each connected sub-components of $(\mathcal{C} \setminus B)$ consists of at most $\delta \cdot c$ bags, for δ some constant smaller than 1. In our data-structure $\delta \leq \frac{1}{2}$. We show that our data-structure can be constructed in $O(n)$ time and space and has query time $O(((\log c)^2 + k) \cdot \log k)$. Our main intuition is to use an iterated binary search over the height and number of leaves to achieve the above bounds. Previously, Reed [26] gave an algorithm that finds a balancing separator, with $\delta \leq \frac{3}{4}$, in $O(c)$ time for constant treewidth graphs. In the case k is constant, our algorithm requires $O((\log c)^2)$ time as compared to the previous $O(c)$-time algorithm. We then show how to use the above queries (while keeping k constant) to construct a binary and balanced tree-decomposition.

2. *(Local reachability).* We present a simple and fast algorithm for local reachability computation (i.e., for each pair of nodes u, v in the same bag of the tree-decomposition whether there is a path from u to v). Our algorithm uses a list data-structure to store sets of nodes of the tree-decomposition. The algorithm is based on two passes of the tree-decomposition, where we do path shortening in each. The algorithm uses $O(n \cdot t^2)$ time and $O(n \cdot t)$ space to compute local reachability for all n nodes. The concept of local reachability has been used before, such as in [1], [12] and [23]. The previous algorithms compute the local distance (i.e., for each pair of nodes u, v in the same bag, the distance from u to v), and our algorithm can also be extended to compute local distances with the same time and space usage as for reachability (see Remark 1). However, the previous algorithms use $\Omega(b \cdot t^2)$ space and $\Omega(b \cdot t^3)$ time (or $\Omega(b \cdot t^4)$ in case of [12]), where b is the number of bags, by storing explicitly all-pairs

2

reachability in each bag, and running Bellman-Ford or Floyd-Warshall type of algorithms in each pass. Note that our algorithm uses both less space and less time, since $n \le b/t$.

Given a bag B let S_B be the set of nodes in ancestor-bags or descendant-bags of B. Given a node u let B_u be the top bag containing u. Given a binary and balanced tree-decomposition along with the local reachability computation, it is straightforward to compute for each node u in each bag B which nodes in S_B can reach u, and which nodes in S_B are reachable from u, in $O(n \cdot t^2)$ time and $O(n \cdot t)$ space. Using this information we compute if a node u can reach node v, by first finding the lowest common ancestor (LCA) bag B of B_u and B_v and then see if there is a path from u to some node w in B and then from w to v in time $O(\lceil t/\log n \rceil)$, by storing the reachability information in words. Single-source queries from u are computed by traversing up the tree-decomposition from B_u, and by storing the reachability information in words we get a query time of $O(n \cdot t \cdot \log t/\log n)$.

Organization. Section 2 presents definitions of graphs and tree-decompositions, and two key lemmas on tree-decomposition properties. In Section 3 we present a data-structure for answering multiple balancing separator queries on binary trees. Using this data-structure, in Section 4 we describe how given a tree-decomposition T of any graph G, an approximate, balanced and binary tree-decomposition T' of G can be obtained in linear time in the size of T. Section 5 presents a family of graphs G for which the width of any balanced tree-decomposition must increase by a constant factor from the treewidth of G. Section 6 presents an algorithm for computing local reachability in a tree-decomposition. Building on our balancing (Section 4) and local reachability (Section 6) algorithms, in Section 7 we give a data-structure that preprocesses any constant treewidth graph in linear time and space, and supports answering single-source and pair reachability queries in sublinear and constant time, respectively.

Table 1: Algorithms for pair- and single-source reachability queries on a directed graph with n nodes, m edges, and a tree-decomposition of width t. The model of computation is the standard RAM model with wordsize $W = \Theta(\log n)$. Row 1a is the standard DFS/BFS, and row 1b is the result of this paper for any treewidth. Rows 2a and 2b are the results we obtain for constant treewidth.

Row	Preprocessing time	Space usage	Pair query time	Single-source query time	From
1a	–	$O(\lceil m/\log n \rceil)$	$O(m)$	$O(m)$	DFS/BFS [13]
1b	$O(n \cdot t^2)$	$O(n \cdot t)$	$O(\lceil \frac{t}{\log n} \rceil)$	$O(n \cdot t \cdot \log t/\log n)$	Theorem 5
2a	–	$O(\lceil n/\log n \rceil)$	$O(n)$	$O(n)$	DFS/BFS [13]
2b	$O(n)$	$O(n)$	$O(1)$	$O(n/\log n)$	Corollary 2

2 Definitions

Graphs. We consider a directed graph $G = (V, E)$ where V is a set of n nodes and $E \subseteq V \times V$ is an edge relation of m edges. Two nodes $u, v \in V$ are called *neighbors* if $(u, v) \in E$. Given a set $X \subseteq V$, we denote with $G \upharpoonright X$ 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 $1 \le i \le k - 1$ we have $(x_i, x_{i+1}) \in E$. The path P is *acyclic* 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 with $E^* \subseteq V \times V$ the transitive closure of E, i.e., $(u, v) \in E^*$ iff there exists a path $P : u \rightsquigarrow v$. Given a path P we use the set notation $u \in P$ to say that a node u appears in P, and $A \cap P$ to refer to the set of nodes that appear in both P and a set A.

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 acyclic 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 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 with $T(u)$ the subtree of T rooted in u (i.e., the tree consisting of all descendants of u), and with s_u the number of nodes in $T(u)$. 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 $O(\log n)$.

Connected components in trees. 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 acyclic path P_u^v in T visits only nodes in $\mathcal{C}$. Given a node u and a connected component $\mathcal{C}$ we denote with $s_u^{\mathcal{C}}$ the number of nodes in $T(u) \upharpoonright \mathcal{C}$. A set of nodes $X \subseteq V_T$ is called a *border* if for all pairs $(u, v) \in X \times X$ such that $u \neq v$ we have that neither u nor v is an ancestor of the other. Given a rooted tree T, we represent a connected component $\mathcal{C}$ of T as a *component pair* $(r_{\mathcal{C}}, X_{\mathcal{C}})$ where $r_{\mathcal{C}}$ is the unique node of $\mathcal{C}$ with the smallest level (called the *root* of $\mathcal{C}$) and $X_{\mathcal{C}}$ is a border. A node u is considered to belong to $\mathcal{C}$ iff u is a descendant of $r_{\mathcal{C}}$ in T and the path $r_{\mathcal{C}} \rightsquigarrow u$ does not contain any node of the border $X_{\mathcal{C}}$ (i.e., $\mathcal{C}$ is obtained from T by cutting the edge of each node from $\{r_{\mathcal{C}}\} \cup X_{\mathcal{C}}$ to its parent).

Balancing separator. For a binary tree T and a connected component $\mathcal{C}$, a *balancing separator* of $\mathcal{C}$ is a node $u \in \mathcal{C}$ such that removal of u splits $\mathcal{C}$ to at most three connected components $(\mathcal{C}_i)_{1 < i \leq 3}$ with $|\mathcal{C}_i| \leq \frac{|\mathcal{C}|}{2}$ for all i.

Tree-decomposition. 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 i, j, k such that there exist paths $B_i \rightsquigarrow B_k$ and $B_k \rightsquigarrow B_j$ 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 with $\mathsf{Lv}(u) = \mathsf{Lv}(B_u)$. 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 the this work we consider that every tree-decomposition $\mathrm{Tree}(G)$ is binary (if not, a tree-decomposition can be made binary by a standard process that increases the size of the tree-decomposition 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.
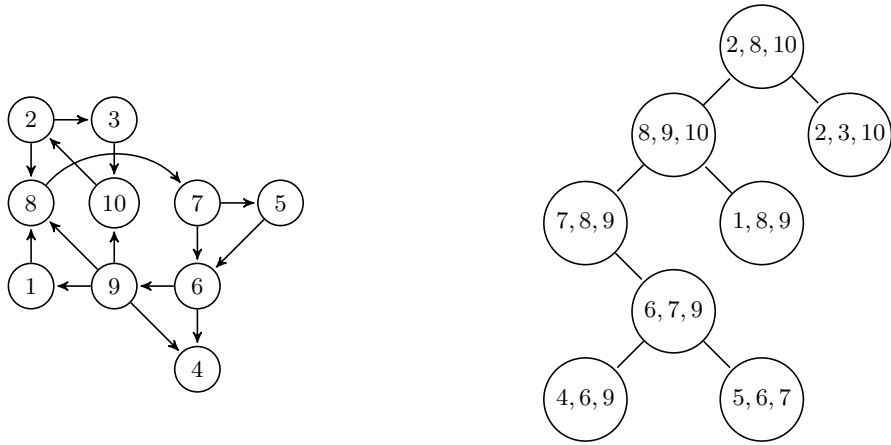

Fig. 1: A graph G with treewidth 2 (left) and a corresponding tree-decomposition $\mathrm{Tree}(G)$ (right).

4

Lemma 1. *Consider a graph $G = (V, E)$, a tree-decomposition $T = \mathrm{Tree}(G)$ and a bag B of T. Denote with $(\mathcal{C}_i)_{1 \leq i \leq 3}$ 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 $P : u \rightsquigarrow v$, we that have $P \cap B \neq \emptyset$ (i.e., all paths between u and v go through some node in B).*

Using Lemma 1, we can prove the following version, which will also 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 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 acyclic path in T from B_1 to B_j. For each $i \in \{1, \ldots, j-1\}$ and for each path $P : u \rightsquigarrow v$, there exists a node $x_i \in (B_i \cap B_{i+1} \cap P)$.*

Proof. Let $T = \mathrm{Tree}(G)$. Fix a number $i \in \{1, \ldots, j-1\}$. We argue that for each path $P : u \rightsquigarrow v$, there exists a node $x_i \in (B_i \cap B_{i+1} \cap P)$. We construct a tree T', which is similar to T except that instead of having an edge between bag B_i and bag B_{i+1}, there is a new bag B, that contains the nodes in $B_i \cap B_{i+1}$, and there is an edge between B_i and B and one between B and B_{i+1}. It is easy to see that T' forms a tree-decomposition of G, from the definition. By Lemma 1, each bag B' in the unique path $P'' : B_1, \ldots, B_i, B, B_{i+1}, \ldots, B_j$ 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. $\qquad\square$

Nice tree-decomposition. 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 in this paper, we also require that the root of a nice tree-decomposition has size 1. Note that in a nice tree-decomposition, every bag is the root bag of at most one node.

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

3   A Data-structure for Balancing Separator Queries in Trees

In this section we consider the problem of finding balancing separators in a fixed binary tree $T = (V_T, E_T)$. When the problem needs to be solved several times on T for different components, T can be preprocessed once such that each balancing separator is obtained potentially faster (without traversing the whole component). We provide a data-structure for this purpose.

Problem description. Consider a fixed binary tree $T = (V_T, E_T)$. Given a component $\mathcal{C}$ of T as input, the problem consists of finding a balancing separator of $\mathcal{C}$.

Classic algorithmic solution. The problem admits a well-known $O(|\mathcal{C}|)$ time algorithm described as follows.

First, apply a post-order traversal on $T \upharpoonright \mathcal{C}$, and store in each node u the size $s_u^{\mathcal{C}}$ of $T(u) \upharpoonright \mathcal{C}$. Then, start from the root of $T \upharpoonright \mathcal{C}$, and for current node u, and as long as u is not a balancing separator move to the neighbor v of u in the largest component $\mathcal{C}_i$ created by removing u. In particular, if v_1, v_2 are the children of u, proceed to the child v_i with $s_{v_i}^{\mathcal{C}} \geq \frac{|\mathcal{C}|}{2}$.

The correctness follows from the fact with each step the size of the largest component $\mathcal{C}_i$ reduces at least by 1, and hence after at most $\frac{|\mathcal{C}|}{2}$ steps, all components $(\mathcal{C}_i)_{1 < i \leq 3}$ created by removing u have size at most $\frac{|\mathcal{C}|}{2}$, and u is a balancing separator.

A data structure for balancing separator queries. The preprocessing phase of the data structure receives the binary tree $T = (V_T, E_T)$ as input. After preprocessing, the data structure supports queries for any component $\mathcal{C}$ of T, represented as a component pair $(r_\mathcal{C}, X_\mathcal{C})$. The goal is to preprocess T in linear time, and answer each balancing separator query $(r_\mathcal{C}, X_\mathcal{C})$ in time that depends only logarithmically on $|\mathcal{C}|$, and slightly superlinearly on $|X_\mathcal{C}|$. We present a data-structure BalSep that achieves the desired preprocessing and querying time bounds.

BalSep **Preprocessing**. The preprocessing of BalSep consists of the following steps.

1. Apply a post-order traversal on T and assign to each leaf v a *leaf-index* l_v that equals the number of leaves that have been visited before v (i.e., the leftmost leaf is assigned leaf-index 0, the leftmost of the remaining leaves is assigned leaf-index 1 etc). To each node u assign an index number i_u that equals its visit time. Additionally, store the size s_u of $T(u)$, and numbers l_u^ℓ, l_u^r containing the smallest and largest leaf-index of leaves in $T(u)$ respectively.
2. Preprocess T to return for any node u and number k, the ancestor of u at level k in $O(1)$ time [4]. This is similar to preprocessing for answering LCA queries in constant time.

Note that a node u is an ancestor of a node v iff $l_u^\ell \leq l_v^\ell$ and $l_u^r \geq l_v^r$ and $\mathsf{Lv}(u) \leq \mathsf{Lv}(v)$, which can be checked in $O(1)$ time. Figure 2 shows an example of the preprocessing.

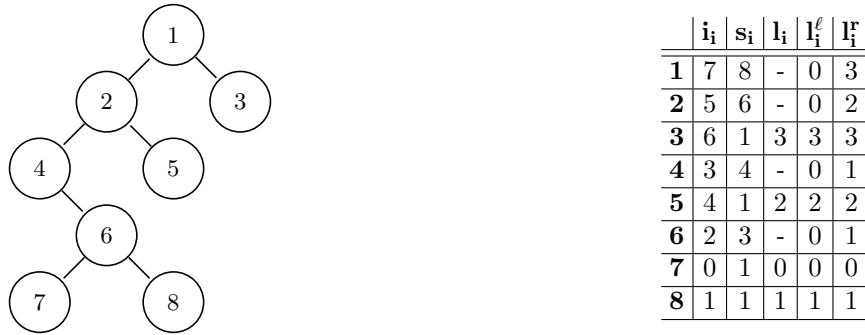|	$\mathbf{i_i}$	$\mathbf{s_i}$	$\mathbf{l_i}$	$\mathbf{l_i^\ell}$	$\mathbf{l_i^r}$
1	7	8	-	0	3
2	5	6	-	0	2
3	6	1	3	3	3
4	3	4	-	0	1
5	4	1	2	2	2
6	2	3	-	0	1
7	0	1	0	0	0
8	1	1	1	1	1

Fig. 2: Preprocessing of BalSep on the tree-decomposition of Figure 1, where the bags have been numbered from 1 to 8.

Query intuition. Now we turn our attention to querying. We first make some claims on how after the above preprocessing, certain operations on any component $\mathcal{C}$ can be made fast. Afterwards we give a formal description of the querying based on such operations.

Fact 1. Given a node $u \in \mathcal{C}$, the indexes i_v of nodes $v \in X_\mathcal{C}$ that are descendants of u form contiguous interval. Formally, if $v_1, v_2 \in X_\mathcal{C}$ are descendants of u with $i_{v_1} \leq i_{v_2}$, then every $v \in X_\mathcal{C}$ with $i_{v_1} \leq i_v \leq i_{v_2}$ is a descendant of u.

Median leaf. Given a node $u \in \mathcal{C}$, let v range over all leaves of $T(u) \upharpoonright \mathcal{C}$ (note that v is either a leaf of T, or the parent of a node in $X_\mathcal{C}$). A node w is called the *median leaf* of $T(u) \upharpoonright \mathcal{C}$ if i_w is the median of all i_v (we take the median of a sorted sequence $(x_1, \ldots x_k)$ to be the element at position $\lfloor \frac{k}{2} \rfloor + 1$). Consider a component pair $(r_\mathcal{C}, X_\mathcal{C})$ with $|X_\mathcal{C}| = k$. We make the following claims.

Claim 1. The pair $(r_\mathcal{C}, X_\mathcal{C})$ can be processed in $O(k \cdot \log k)$ time, so that for any $u \in \mathcal{C}$, the size $s_u^\mathcal{C}$ of $T(u) \upharpoonright \mathcal{C}$ can be determined in $O(\log k)$ time.

Proof. Let $Y_\mathcal{C} = (v_1, \ldots, v_k)$ be the list of nodes of $X_\mathcal{C}$ sorted according to their index, and $\overline{S}_\mathcal{C} = (\overline{s}_0, \overline{s}_1, \ldots, \overline{s}_k)$ such that $\overline{s}_0 = 0$ and $\overline{s}_i = \sum_{j=1}^{i} s_{v_j}$ (recall that s_{v_j} is the size of $T(v_j)$). The list $\overline{S}_\mathcal{C}$ can be constructed in $O(k \cdot \log k)$ time, and $\overline{s}_i$ is the sum of the sizes of the subtrees rooted at the border nodes $v_1, \ldots v_i$. By Fact 1, there is a leftmost v_ℓ and a rightmost v_r node in $Y_\mathcal{C}$ such that for all $\ell \leq i \leq r$, we have that v_i is a descendant of u. It follows that the size of $T(u) \upharpoonright \mathcal{C}$ is $s_u^\mathcal{C} = s_u - \overline{s}_r + \overline{s}_{\ell-1}$. Finally, both v_ℓ and v_r can be determined by a binary search in $Y_\mathcal{C}$, and thus in $O(\log k)$ time. $\qquad\square$

Claim 2. The pair $(r_\mathcal{C}, X_\mathcal{C})$ can be processed in $O(k \cdot \log k)$ time, so that for any $u \in \mathcal{C}$, the median leaf w of $T(u) \upharpoonright \mathcal{C}$ can be determined in $O(\log k)$ time.

Proof. Let $Y_\mathcal{C} = (v_1, \ldots, v_k)$ be the list of nodes of $X_\mathcal{C}$ sorted according to their index. For a border node $v_j \in X_\mathcal{C}$, let $b_{v_j} = 1$ if the parent of v_j is a leaf of $\mathcal{C}$ and v_j is either a right child or the only child, else $b_{v_j} = 0$. Let $\overline{L}_\mathcal{C} = (\overline{l}_0, \overline{l}_1, \ldots, \overline{l}_k)$ such that $\overline{l}_0 = 0$ and $\overline{l}_i = \sum_{j=1}^{i} (l_{v_j}^r - l_{v_j}^\ell + 1 - b_{v_j})$, (recall that $l_{v_j}^r$ and $l_{v_j}^r$ are respectively the smallest and largest leaf-indexes of $T(v_j)$). The list $\overline{L}_\mathcal{C}$ can be constructed in $O(k \cdot \log k)$ time, and $\overline{l}_i$ is the sum of the number of leaves of the subtrees rooted at the border nodes $v_1, \ldots v_i$ minus the number of parents of $v_1, \ldots v_i$ that are leaves in $T \upharpoonright \mathcal{C}$ (i.e., it is the number of leaves of $T(r_\mathcal{C})$ that are "hidden" by the border up to v_i, minus the new leaves that the border introduces until v_i). Given a node $u \in \mathcal{C}$, let j_1 and j_2 be such that v_{j_1} and v_{j_2} are respectively the leftmost and rightmost nodes in $Y_\mathcal{C}$ that are descendants of u. For each $j_1 \leq i \leq j_2$, the number of leaves in $T(u) \upharpoonright \mathcal{C}$ with index $< i_{v_i}$ is $f_u(i) = l_{v_i}^\ell - l_u^\ell - (\overline{l}_{i-1} - \overline{l}_{j_1-1})$. The number of leaves in $T(u) \upharpoonright \mathcal{C}$ is $g(u) = l_u^r - l_u^\ell + 1 - (\overline{l}_{j_2} - \overline{l}_{j_1-1})$. Let j be such that v_j is the rightmost descendant of u in $Y_\mathcal{C}$ with $f_u(j) \leq \lfloor \frac{g(u)}{2} \rfloor$. There are two cases:

1. If $f_u(j) = \lfloor \frac{g(u)}{2} \rfloor$ and $b_{v_j} = 1$, let w be the parent of v_j and thus a leaf of $T(u) \upharpoonright \mathcal{C}$. Then there are $\lfloor \frac{g(u)}{2} \rfloor + 1$ leaves in $T(u) \upharpoonright \mathcal{C}$ with index $\leq i_w$, and hence w is the median leaf of $T(u) \upharpoonright \mathcal{C}$.

2. Else, let w be the leaf of T with leaf-index $l_w = l_{v_j}^r + \lfloor \frac{g(u)}{2} \rfloor - f_u(j) + 1$. Note that w is a leaf of $T(u) \upharpoonright \mathcal{C}$, otherwise w is a leaf of $T(v_{j'})$ for some $j' > j$, which contradicts our choice of j because then $f_u(j') - f_u(j) \leq \lfloor \frac{g(u)}{2} \rfloor - f_u(j)$ thus $f_u(j') \leq \lfloor \frac{g(u)}{2} \rfloor$. There are $\lfloor \frac{g(u)}{2} \rfloor + 1$ leaves of $T(u) \upharpoonright (\mathcal{C})$ with index $\leq i_w$, hence w is the median leaf of $T(u) \upharpoonright \mathcal{C}$.

Finally, each j_1, j_2, j can be determined by a binary search on $Y_\mathcal{C}$, thus in $O(\log k)$ time. $\qquad\square$

BalSep **Querying.** We now give a formal description of the querying. Given a query $(r_\mathcal{C}, X_\mathcal{C})$, the balancing separator u of $\mathcal{C}$ is found by the following recursive process, initially starting with $w = r_\mathcal{C}$.

Step 1. Given a node $w \in \mathcal{C}$, find the median leaf v of $T(w) \upharpoonright \mathcal{C}$.

Step 2. Given w and v, obtain the node x that is ancestor of v in level $\lfloor \frac{\mathsf{Lv}(w) + \mathsf{Lv}(v)}{2} \rfloor$ (observe that x is the middle node of the path $w \rightsquigarrow v$).

Step 3. Execute according to the following conditions.

  Cond. C1: If $s_x^\mathcal{C} \geq \frac{|\mathcal{C}|}{2}$ and for each child x_i of x we have $s_{x_i}^\mathcal{C} \leq \frac{|\mathcal{C}|}{2}$, take $u = x$ and terminate.

  Cond. C2: If $s_x^\mathcal{C} \geq \frac{|\mathcal{C}|}{2}$ and for some child x_i of x we have $s_{x_i}^\mathcal{C} > \frac{|\mathcal{C}|}{2}$, set $w = x_i$ and goto Step 1 (i.e., repeat the process for the new w).

  Cond. C3: Finally, if $s_x^\mathcal{C} < \frac{|\mathcal{C}|}{2}$, set $v = x$ and goto Step 2 (i.e., continue the binary search in the new (sub) path $w \rightsquigarrow v$).

For the nodes w, v of the above process, we denote with $\mathcal{C}_v^w$ the component pair $(w, X \cup \{v\})$, where $X \subseteq X_\mathcal{C}$ contains all nodes of $X_\mathcal{C}$ that do not exist in $T(v)$. The following two lemmas establish the correctness and time complexity of a query on BalSep.

Lemma 3. *On query* $(r_\mathcal{C}, X_\mathcal{C})$, BalSep *correctly returns a balancing separator* u *of* $\mathcal{C}$.

7

Proof. First note that a node u is a balancing separator of $\mathcal{C}$ iff condition C1 holds. Every time condition C2 or C3 is executed, the component $\mathcal{C}_v^w$ reduces in size. We argue inductively that $\mathcal{C}_v^w$ always contains a node that satisfies condition C1. The claim is true initially, since $\mathcal{C}_v^w = \mathcal{C}$. Now assume the claim holds for some w and v, and let x be the current node examined in conditions C1-C3, and w', v' be the new pair of nodes obtained in the next iteration, because C1 did not hold. In case of C2, every node $x' \in \mathcal{C}_v^w \setminus \mathcal{C}_{v'}^{w'}$ either has $s_{x'}^{\mathcal{C}} > \frac{|C|}{2}$ (if x' is an ancestor of x), or $s_{x'}^{\mathcal{C}} < \frac{|C|}{2}$ (if x' is a descendant of x and $x' \neq x$). In case of C3, every node $x' \in \mathcal{C}_v^w \setminus \mathcal{C}_{v'}^{w'}$ has $s_{x'}^{\mathcal{C}} < \frac{|C|}{2}$. Thus in both cases the component $\mathcal{C}_v^w \setminus \mathcal{C}_{v'}^{w'}$ does not contain a balancing separator. By the induction hypothesis, such a separator exists in $\mathcal{C}_v^w$ and hence we get that $u \in \mathcal{C}_{v'}^{w'}$. $\qquad\square$

Lemma 4. *The query phase of* BalSep *requires* $O(\log^2 |\mathcal{C}| \cdot \log |X_\mathcal{C}| + |X_\mathcal{C}| \cdot \log X_\mathcal{C})$ *time.*

Proof. Given a pair of nodes w, v, there will be $O(\log |\mathcal{C}|)$ choices of x on the path $P : w \rightsquigarrow v$, since every such path has length at most $|\mathcal{C}|$, and every such choice of x halves the length of P. Every time the search moves from a component $\mathcal{C}_v^w$ to a component $\mathcal{C}_{v'}^{w'}$ via a child $x_i = w'$ of x that is not in the path $w \rightsquigarrow v$, the number of leaves in $T(w) \upharpoonright \mathcal{C}_{v'}^{w'}$ is half of that of $T(w) \upharpoonright \mathcal{C}_v^w$, and hence such choice for x_i can be made $O(\log |\mathcal{C}|)$ times. This concludes that the above conditions C2 and C3 will hold $O(\log^2 |\mathcal{C}|)$ times. Finally, by Claims 1 and 2, after processing $\mathcal{C}$ in $O(|X_\mathcal{C}| \cdot \log |X_\mathcal{C}|)$ time, every median leaf v and sizes $s_x^{\mathcal{C}}$, $s_{x_i}^{\mathcal{C}}$ can be obtained in $O(\log |X_\mathcal{C}|)$ time. Hence every execution of conditions C2 or C3 requires $O(\log |X_\mathcal{C}|)$ time, and the desired result follows. $\qquad\square$

We conclude the results of this section with the following theorem.

Theorem 1. *Consider a binary tree T with n nodes, and a sequence of balancing separator queries of the form $(r_\mathcal{C}, X_\mathcal{C})$. The data-structure* BalSep *preprocesses T in $O(n)$ time, and answers each query with a balancing separator of $\mathcal{C}$ in $O(\log^2 |\mathcal{C}| \cdot \log |X_\mathcal{C}| + |X_\mathcal{C}| \cdot \log |X_\mathcal{C}|)$ time.*

4 Balancing a Tree-Decomposition in Linear Time

In this section we show how given a graph G and a tree-decomposition $\mathrm{Tree}(G)$ of b bags and width t, we can construct in $O(b)$ time and space a balanced binary tree-decomposition with $O(b)$ bags and width at most $4 \cdot t + 3$. The process has two conceptual steps. First, we construct a *rank tree* $\mathsf{R_G}$ of G that is balanced, and then show how to turn $\mathsf{R_G}$ to a tree-decomposition $\widehat{\mathsf{R_G}}$ by a simple modification.

Constructing a rank tree. In the following, we consider that $\mathrm{Tree}(G) = (V_T, E_T)$ is binary, has width t, and $|V_T| = b$ bags. Given $\mathrm{Tree}(G)$, we assign to each bag $B \in V_T$ a rank $\mathsf{r}(B) \in \mathbb{N}$ according to the following recursive algorithm Rank. Rank operates on input $(\mathcal{C}, f, k)$ where $\mathcal{C}$ is a component of $\mathrm{Tree}(G)$ represented as a component pair $(r_\mathcal{C}, X_\mathcal{C})$, $f \in \{\mathsf{False}, \mathsf{True}\}$ a is flag, and $k \in \mathbb{N}$ is the rank to be assigned, as follows.

1. If $\mathcal{C}$ contains a single bag B, assign $\mathsf{r}(B) = k$ and terminate.
2. Else, if $f = \mathsf{True}$, find the bag B that is a balancing separator of $\mathcal{C}$. Assign $\mathsf{r}(B) = k$, and call Rank recursively on input $(\mathcal{C}_i, \neg f, k+1)$ for each component $(\mathcal{C}_i)_{1 < i \leq 3}$ created from $\mathcal{C}$ by removing B.
3. Else, if $f = \mathsf{False}$, let L range over all the LCAs of every pair of bags $B_1, B_2 \in X_\mathcal{C}$ and $B = \arg\max_L \mathsf{Lv}(B)$. If $|X_\mathcal{C}| \geq 2$ then such B exists, and assign $\mathsf{r}(B) = k$, and call Rank recursively on input $(\mathcal{C}_i, \neg f, k+1)$ for each component $(\mathcal{C}_i)_{1 < i \leq 3}$ created from $\mathcal{C}$ by removing B. Else, call Rank recursively on input $(\mathcal{C}, \neg f, k)$.

Algorithm Rank induces a ternary *rank tree* $\mathsf{R_G}$ of G such that the root is the unique bag B with $\mathsf{r}(B) = 0$, and a bag B' is the i-th child of a bag B iff B' is the separator of the sub-component $\mathcal{C}_i$ of the component $\mathcal{C}$ of which B is a separator. The level i of $\mathsf{R_G}$ contains all bags B with $\mathsf{r}(B) = i$. Given the rank function r denote the *neighborhood* of a bag B with

$$\mathsf{Nh}(B) = \{B' \in V_T : \mathsf{r}(B') < \mathsf{r}(B) \text{ and for all intermediate bags}$$
$$B'' \text{ in the path } B \rightsquigarrow B', \ \mathsf{r}(B) \leq \mathsf{r}(B'')\}$$

i.e, $\mathsf{Nh}(B)$ consists of the bags B' that have smaller rank than B, and that B can reach through a path that traverses intermediate bags of larger rank. The following facts are easy to obtain.

Fact 2. For each bag B that is a separator of a component represented as a component pair $(r_{\mathcal{C}}, X_{\mathcal{C}})$ in Rank, if $\mathsf{r}(r_{\mathcal{C}}) < \mathsf{r}(B)$ then $\mathsf{Nh}(B) = X_{\mathcal{C}} \cup \{r_{\mathcal{C}}\}$ otherwise $\mathsf{Nh}(B) = X_{\mathcal{C}}$. Hence we always have $X_{\mathcal{C}} \subseteq \mathsf{Nh}(B)$ and $|X_{\mathcal{C}}| \leq |\mathsf{Nh}(B)| \leq |X_{\mathcal{C}}| + 1$.

Fact 3. Let B and B' be respectively a bag and its parent in $\mathsf{R_G}$. Then $\mathsf{Nh}(B) \subseteq \mathsf{Nh}(B') \cup \{B'\}$, and thus $|\mathsf{Nh}(B)| \leq |\mathsf{Nh}(B')| + 1$.

Intuitively, for a bag B, the set $\mathsf{Nh}(B)$ is the set of bags separating the component for which B was chosen as a separator by Rank from the rest of the graph. Since every bag in $\mathsf{R_G}$ corresponds to a bag in $\mathrm{Tree}(G)$, the bags of $\mathsf{R_G}$ already cover all nodes and edges of G (i.e., properties T1 and T2 of a tree-decomposition). In the following we show how $\mathsf{R_G}$ can be modified to also satisfy condition T3, i.e., that every node u appears in a contiguous subtree of $\mathsf{R_G}$. Given a bag B, we denote with $\mathsf{NhV}(B) = B \cup \bigcup_{B' \in \mathsf{Nh}(B)} B'$, i.e., $\mathsf{NhV}(B)$ is the set of nodes of G that appear in B and its neighborhood. In Lemma 5 we will show the crucial property that for all nodes u and paths $P : B_1 \rightsquigarrow B_2$ in $\mathsf{R_G}$ such that B_1 is ancestor of B_2 and $u \in (B_1 \cap B_2)$, for all bags $B \in P$ we have that $u \in \mathsf{NhV}(B)$. We start with a basic claim that will be useful throughout this section.

Claim 3. For all $B_1, B_2 \in V_T$, let B be their LCA in $\mathsf{R_G}$ and P be the unique acyclic path $B_1 \rightsquigarrow B_2$ in $\mathrm{Tree}(G)$. Then $B \in P$ and all B' in P are descendants of B in $\mathsf{R_G}$.

Proof. Let $\mathcal{C}$ be the smallest component processed by Rank that contains both B_1 and B_2. Then B was chosen as a separator for $\mathcal{C}$, hence $B \in P$, and by the recursion, all B' in $B_1 \rightsquigarrow B_2$ will be descendants of B in $\mathsf{R_G}$. □

We say that a pair of bags (B_1, B_2) *form a gap* of some node u in a tree T of bags (e.g., $\mathsf{R_G}$) if $u \in B_1 \cap B_2$ and for the unique acyclic path $P : B_1 \rightsquigarrow B_2$ in T we have that $|P| \geq 2$ (i.e., there is at least one intermediate bag in P) and for all intermediate bags B in P we have $u \notin B$.

Lemma 5. *For every node u, and pair of bags (B_1, B_2) that form a gap of u in $\mathsf{R_G}$, such that B_1 is an ancestor of B_2, for every intermediate bag B in $P : B_1 \rightsquigarrow B_2$ in $\mathsf{R_G}$, we have that $u \in \mathsf{NhV}(B)$.*

Proof. The proof is by showing that $B_1 \in \mathsf{Nh}(B)$. Fix arbitrarily such a bag B, and since B is ancestor of B_2 we have that $\mathsf{r}(B_2) > \mathsf{r}(B)$. Let $P_1 : B \rightsquigarrow B_2$ and $P_2 : B_2 \rightsquigarrow B_1$ be paths in $\mathrm{Tree}(G)$. We argue that for all intermediate bags B' in P_1 and P_2, we have that $\mathsf{r}(B') > \mathsf{r}(B)$. When B' is an intermediate bag of P_1, the statement follows from Claim 3 since the LCA of B and B_2 in $\mathsf{R_G}$ is B. Now consider that B' is an intermediate bag in P_2. From property T3 of a tree-decomposition, since $u \in B_1$ and $u \in B_2$, then $u \in B'$. Since B_1 is the LCA of B_1 and B_2 in $\mathsf{R_G}$, by Claim 3 B' is a descendant of B_1 in $\mathsf{R_G}$. Let B'' be the LCA of B_2 and B' in $\mathsf{R_G}$, and B'' is also a descendant of B_1. By Claim 3, B'' is a node in the path $P_3 : B_2 \rightsquigarrow B'$ in $\mathrm{Tree}(G)$, hence $u \in B''$. Since (B_1, B_2) form a gap of u, it follows that $B'' = B_2$, and B' is a descendant of B_2, hence $\mathsf{r}(B') > \mathsf{r}(B_2) > \mathsf{r}(B)$.

The above conclude that all intermediate nodes B' in the path $B \rightsquigarrow B_1$ in $\mathrm{Tree}(G)$ have rank $\mathsf{r}(B') > \mathsf{r}(B)$, and since $\mathsf{r}(B_1) < \mathsf{r}(B)$, we have that $B_1 \in \mathsf{Nh}(B)$. Since, $u \in B_1$ it is $u \in \mathsf{NhV}(B)$ and the desired result follows. □

Procedure Replace**.** Lemma 5 suggests a way to turn the rank tree $\mathsf{R_G}$ to a tree-decomposition. Let $\widehat{\mathsf{R_G}} = \mathsf{Replace}(\mathsf{R_G})$ be the tree obtained by replacing each bag B of $\mathsf{R_G}$ with $\mathsf{NhV}(B)$. For a bag B in $\mathsf{R_G}$ let $\widehat{B}$ be the corresponding bag in $\widehat{\mathsf{R_G}}$ and vice versa. The following lemma states that $\widehat{\mathsf{R_G}}$ is a tree-decomposition of G.

Lemma 6. $\widehat{\mathsf{R_G}} = \mathsf{Replace}(\mathsf{R_G})$ *is a tree-decomposition of G.*

Proof. It is straightforward to see that the bags of $\widehat{\mathsf{R_G}}$ cover all nodes and edges of G (properties T1 and T2 of the definition of tree-decomposition), because for each bag B, we have that $\widehat{B}$ is a superset of B. It remains to show that every node u appears in a contiguous subtree of $\widehat{\mathsf{R_G}}$ (i.e., that property T3 is satisfied). Assume towards contradiction otherwise, and it follows that there exist bags $\widehat{B}_1$ and $\widehat{B}_2$ in $\widehat{\mathsf{R_G}}$ that form a gap of some node u and let $\widehat{P} : \widehat{B}_1 \rightsquigarrow \widehat{B}_2$ be the path between them. Observe that for any bag B, if $u \notin B$, but $u \in \widehat{B}$, then u is also in the parent of $\widehat{B}$, by Fact 3. Hence, if $u \notin (B_1 \cap B_2)$, then u is in a parent of either $\widehat{B}_1$ or $\widehat{B}_2$.

First we establish that one $\widehat{B}_i$ must be ancestor of the other. If not, let $\widehat{L}$ be the LCA of $\widehat{B}_1$ and $\widehat{B}_2$ in $\mathsf{R_G}$. Since $\widehat{L} \in \widehat{P}$, we get that $u \notin \widehat{L}$. Hence, $u \notin L$ and thus $u \notin (B_1 \cap B_2)$ by Claim 3 and property T3 of tree-decomposition. This indicates that u is in a parent of either $\widehat{B}_1$ or $\widehat{B}_2$. But $\widehat{P}$ goes through both parents, contradicting that $\widehat{B}_1$ and $\widehat{B}_2$ form a gap.

Next, consider that $\widehat{B}_1$ is an ancestor of $\widehat{B}_2$ (the case where $\widehat{B}_1$ is a descendant of $\widehat{B}_2$ is symmetric). Observe that $u \notin (B_1 \cap B_2)$, since otherwise we get a contradiction from Lemma 5. We have that $u \in B_2$, since otherwise u also appears in the parent of $\widehat{B}_2$, which is in $\widehat{P}$. Hence we have that $u \notin B_1$. Let bag $B' \in \mathsf{Nh}(B_1)$ be such that $u \in B'$, and let L' be the LCA of B' and B_2 in $\mathsf{R_G}$. It follows that L' is an ancestor of B_1. By Claim 3, L' appears in the path $B' \rightsquigarrow B_2$ in $\mathrm{Tree}(G)$, and hence, by property T3 of tree-decomposition we have that $u \in L'$. Let B'' be the first bag in the path $B_1 \rightsquigarrow L'$ in $\mathsf{R_G}$ that contains u. Observe that u does not appear in any intermediate bag of the path $P : B_1 \rightsquigarrow B_2$, because otherwise it contradicts that $\widehat{B}_1$ and $\widehat{B}_2$ form a gap of u. Hence, u does not appear in any intermediate bag of the path $P' : B'' \rightsquigarrow B_1 \rightsquigarrow B_2$, thus B'' and B_2 form a gap of u in $\mathsf{R_G}$. By Lemma 5, for each B in P', and thus especially for the ones in P, we have that $\widehat{B}$ contains u, contradicting that $\widehat{B}_1$ and $\widehat{B}_2$ form a gap of u. The desired result follows. $\qquad\square$

Lemma 6 states that $\widehat{\mathsf{R_G}}$ obtained by replacing each bag of $\mathsf{R_G}$ with $\mathsf{NhV}(B)$ is a tree-decomposition of G. The remaining of the section focuses on showing that $\widehat{\mathsf{R_G}}$ is a balanced approximate tree-decomposition of G, and that it can be constructed in $O(b)$ time and space. The following lemma states that $\mathsf{R_G}$ is balanced (of height $O(\log b)$) and $|\mathsf{Nh}(B)| \leq 3$ for each bag B of $\mathsf{R_G}$.

Lemma 7. *The following assertions hold:*

1. *The height of $\widehat{\mathsf{R_G}}$ is $O(\log b)$.*
2. *For each bag $\widehat{B}$ of $\widehat{\mathsf{R_G}}$, we have $|\widehat{B}| \leq 4 \cdot (t+1)$.*

Proof. We prove each item separately by showing that (i) the height of $\mathsf{R_G}$ is $O(\log b)$, and (ii) $|\mathsf{Nh}(B)| \leq 3$.

1. It is clear that the height of $\mathsf{R_G}$ equals the recursion depth of Rank. Because of the alternating flag, the size of each input component is at halved at least every two successive recursive calls. Hence, at recursion level i, for each component $\mathcal{C}$ processed by Rank in that level we have $|\mathcal{C}| \leq b \cdot 2^{-\lfloor i/2 \rfloor}$. The recursion stops when $|\mathcal{C}| = 1$, hence the height of $\mathsf{R_G}$ is $O(\log b)$.
2. Let B be any bag, and B' its parent in $\mathsf{R_G}$. By Fact 3, $|\mathsf{Nh}(B)|$ can increase by at most one from its parent's. Observe that each bag B with $2 \leq |\mathsf{Nh}(B)| \leq 3$ which is assigned a rank with $f = \mathsf{False}$ appears in the path $B_1 \rightsquigarrow B_2$ in $\mathrm{Tree}(G)$ of every two distinct bags $B_1, B_2 \in \mathsf{Nh}(B)$. Hence for each child B' of B in $\mathsf{R_G}$ we have that $B_i \in \mathsf{Nh}(B')$ for at most one $B_i \in \mathsf{Nh}(B)$, and thus $|\mathsf{Nh}(B')| \leq 2$. Since the flag f alternates between every successive recursive calls of Rank, we get that for every pair of parent-child bags B_1 and B_2, it is $|\mathsf{Nh}(B_i)| \leq 2$ for at least one $i \in \{1, 2\}$. The desired result follows.

$\qquad\square$

Algorithm Balance. We are now ready to outline the algorithm Balance, which takes as input a tree-decomposition $\mathrm{Tree}(G)$ of a graph G, and returns a balanced binary tree-decomposition of G. Balance is based on Rank, with additional preprocessing from Section 3 to obtain the separator bags of Rank fast. Balance operates as follows:

1. Preprocess $\mathrm{Tree}(G)$ using the BalSep data-structure from Section 3 so that on recursive call $(\mathcal{C}, f, k)$ of Rank with $f = \mathsf{True}$, a balancing separator B of $\mathcal{C}$ is returned.

2. Preprocess $\mathrm{Tree}(G)$ to answer LCA queries in $O(1)$ time. On recursive call $(\mathcal{C}, f, k)$ of Rank with $f = $ False, obtain LCAs L in $O(1)$ time.

3. Return Replace($\mathsf{R_G}$).

In Figure 3 we give an example of Balance executed on a tree-decomposition $\mathrm{Tree}(G)$. First, $\mathrm{Tree}(G)$ is turned into a binary and balanced tree $\mathsf{R_G}$ and then into a binary and balanced tree $\widehat{\mathsf{R_G}}$. If the numbers are pointers to bags, such that $\mathrm{Tree}(G)$ is a tree-decomposition for G, then $\widehat{\mathsf{R_G}}$ is a binary and balanced tree-decomposition of G.
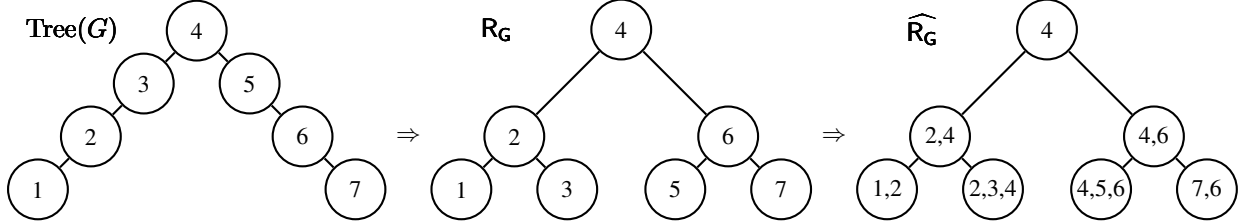
Fig. 3: Given the tree-decomposition $\mathrm{Tree}(G)$ on the left, the graph in the middle is the corresponding $\mathsf{R_G}$ and the one on the right is the corresponding balanced tree-decomposition $\widehat{\mathsf{R_G}} = $ Replace($\mathsf{R_G}$) after replacing each bag B with $\mathsf{NhV}(B)$.

Lemma 8. *Algorithm* Balance *runs in $O(b)$ time and space.*

Proof. Let B be any bag that was assigned a rank when Rank was executed for a component pair $(r_\mathcal{C}, X_\mathcal{C})$. By Fact 2 and Lemma 7 Item 2, it is $|X_\mathcal{C}| \leq |\mathsf{Nh}(B)| \leq 3$. By Theorem 1, $\mathrm{Tree}(G)$ can be preprocessed in $O(b)$ time, such that each balancing separator of component $\mathcal{C}$ is obtained in $O(\log^2 |\mathcal{C}| \cdot \log |X_\mathcal{C}| + |X_\mathcal{C}| \cdot \log |X_\mathcal{C}|) = O(\log^2 |\mathcal{C}|)$ time. Hence when $f = $ True, Rank spends $O(\log^2 |\mathcal{C}|)$ time in $\mathcal{C}$. Moreover, with $O(b)$ additional preprocessing time, when $f = $ False, in $O(1)$ time the desired bag B is obtained via at most three LCA queries of $O(1)$ time each. We can write the following recurrences for the time complexity $\mathcal{T}_f(b)$ of Rank on input of size b and flag f:

$$\mathcal{T}_{\mathsf{True}}(b) \leq \sum_{i=1}^{3} \mathcal{T}_{\mathsf{False}}\left(\gamma_{\mathsf{True}}^i \cdot b\right) + O\left(\log^2 b\right)$$

$$\mathcal{T}_{\mathsf{False}}(b) \leq \sum_{i=1}^{3} \mathcal{T}_{\mathsf{True}}\left(\gamma_{\mathsf{False}}^i \cdot b\right) + O(1)$$

with $\gamma_{\mathsf{True}}^i \leq \frac{1}{2}$ and $\sum_i \gamma_{\mathsf{True}}^i < 1$ and $\sum_i \gamma_{\mathsf{False}}^i < 1$. Initially the algorithm is called with $f = $ True, and the running time $\mathcal{T}(b)$ satisfies the recurrence

$$\mathcal{T}(b) \leq \sum_{i=1}^{9} \mathcal{T}\left(\gamma^i \cdot b\right) + O\left(\log^2 b\right)$$

with $\gamma^i \leq \frac{1}{2}$ and $\sum_i \gamma_i < 1$. It is easy to verify that $\mathcal{T}(b) = O(b)$ as desired. The space complexity follows. $\qquad\square$

We are now ready to wrap-up the results of this section. Given a tree-decomposition $\mathrm{Tree}(G)$ of b bags and width t, Balance constructs a tree $\widehat{\mathsf{R_G}}$ which, by Lemma 6 is a tree-decomposition of G. By Lemma 7, $\widehat{\mathsf{R_G}}$ is balanced, and has width at most $4 \cdot t + 3$. Finally, $\widehat{\mathsf{R_G}}$ can be turned from ternary to binary with only a constant increase in its size, while maintaining the properties of being balanced and having width at most $4 \cdot t + 3$. These lead us to the following theorem.

Theorem 2. *Given a graph G, let $\mathcal{T}(G)$ be the time and $\mathcal{S}(G)$ be the space required for constructing a tree-decomposition $T = \text{Tree}(G)$ of b bags and width t. Then a balanced binary tree-decomposition T' of G with $O(b)$ bags and width at most $4 \cdot t + 3$ can be constructed in $O(\mathcal{T}(G) + b)$ time and $O(\mathcal{S}(G) + b)$ space.*

We remark that in the statement of Theorem 2, the tree-decomposition T' is stored implicitly, as a rank tree $\mathsf{R_G}$, where each bag B stores (at most) four pointers to bags in $\text{Tree}(G)$, i.e., the bags corresponding to the bags in $(\{B\} \cup \mathsf{Nh}(B))$. Storing each bag of T' explicitly requires $O(b \cdot t)$ time and and space (rather than $O(b)$).

The new tree-decomposition $\widehat{\mathsf{R_G}}$ can be made nice by a standard process which increases the height by a factor of at most t.

Corollary 1. *Given a graph G, let $\mathcal{T}(G)$ be the time and $\mathcal{S}(G)$ be the space required for constructing a tree-decomposition $T = \text{Tree}(G)$ of b bags and width t. Then a nice binary tree-decomposition T' of G of width at most $4 \cdot t + 3$ and height $O(t \cdot \log b)$ can be constructed in $O(\mathcal{T}(G) + b \cdot t)$ time and $O(\mathcal{S}(G) + b \cdot t)$ space.*

5   Lower Bound on Width of Balanced Tree-decompositons

In this section we will argue there exists a family $\{G_t^n \mid n \geq 3 \cdot t \wedge n \equiv 0 \pmod{t}\}$ of graphs, such that the graph G_t^n has n nodes and treewidth $2 \cdot t - 1$ and any tree-decomposition of G_t^n, with width t' and of height h is such that either $h \geq \frac{n}{2 \cdot t'}$ or $t' \geq 3 \cdot t - 1$. Note that if t is $o(n/\log n)$, then only in the later case can the tree-decomposition be balanced. Thus, the width must increase by a factor when one constructs balanced tree-decompositions.

The graph G_t^n. The graph G_t^n is defined as follows. Let $n' = \frac{n}{t}$. For each $i \in \{1, \ldots, n'\}$, let V_i be a set of t nodes, such that $V_i \cap V_j = \emptyset$ for $i \neq j$ and $V = \bigcup_i V_i$. Also, let $V_0 = V_{n'+1} = \emptyset$. For each $i \in \{1, \ldots, n'\}$, each node $u \in V_i$ has an edge to each other node in $V_{i-1} \cup V_i \cup V_{i+1}$. There is an illustration of G_3^{18} in Figure 4.
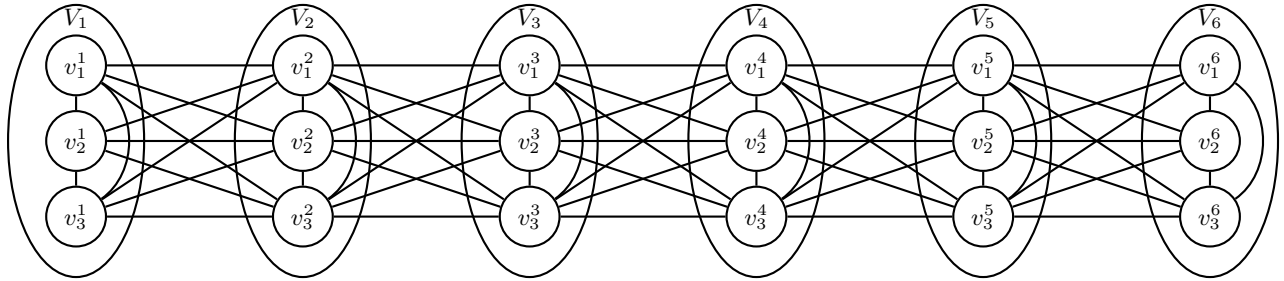

Fig. 4: The graph G_3^{18}.

We will first argue that the treewidth of G_t^n is at most $2t - 1$.

Lemma 9. *For any t and n the treewidth of G_t^n is at most[3] $2 \cdot t - 1$.*

Proof. For each $i \in \{1, \ldots, n' - 1\}$ let bag B_i consist of $V_i \cup V_{i+1}$, and for $i \leq n' - 2$, let it be connected to bag B_{i+1}. It is easy to see that this is a tree-decomposition of G_t^n. $\qquad\square$

Next is a technical lemma that will let us do a case analysis at the end. For the remaining of this section, given a tree of bags T of a graph G, we denote with T_v the set of nodes of G that appear in the bags of T.

---

[3] the treewidth of G_t^n is exactly $2t - 1$, but we will only show the upper bound

Lemma 10. *For any t and n consider the graph G_t^n and a tree-decomposition $\text{Tree}(G_t^n) = (V_T, E_T)$ of it of width t' and height h. Either $h \geq \frac{n}{2 \cdot t'}$ or there exists a bag B and numbers i, j where $j - i \geq 2$ and such that $V_i \cup V_j \subseteq B$.*

Proof. For each bag B in V_T with child-bags $B_1, \ldots, B_k$ for some k we can without loss of generality assume that $(T_v(B_i) \setminus B)$ (resp. $V \setminus T_v(B)$, if B is not the root) are non-empty, since we otherwise could simply remove the subtree $T(B_i)$ (resp. $V \setminus T_v(B)$) and still have a tree-decomposition with the same or lower height and width.

There are now two cases. Either (1) there exists a bag B with child-bags $B_1, \ldots, B_k$ for some $k \geq 2$ (or $k \geq 3$, in case B is the root); or (2) not. If not, the tree-decomposition forms a line with length at least $\frac{n}{t'}$, hence no matter how the root of the tree-decomposition is picked, the height is at least $\frac{n}{2 \cdot t'}$.

Otherwise, in case (1), pick three nodes v_1, v_2, v_3, one in each of $(T_v(B_1) \setminus B)$, $(T_v(B_2) \setminus B)$ and $V \setminus T_v(B)$ (or $(T_v(B_3) \setminus B)$ in case B is the root). Let i', j', k' be such that $v_1 \in V_{i'}$, $v_2 \in V_{j'}$ and $v_3 \in V_{k'}$. We consider the case $i' \leq j' \leq k'$ (the others are similar). We have that $j' - i' \geq 2$ (resp. $k' - j' \geq 2$), since otherwise there is an edge between v_1 and v_2 (resp. v_2 and v_3) and hence a path between them that does not intersect with nodes in B, contradicting Lemma 1. Also, we have that there exists an i and a j such that $i' < i < j' < j < k'$ (hence, $j - i \geq 2$) and such that each node in $V_i \cup V_j$ is in B, since otherwise, there is atleast one node in V_i for $i' < i < j'$ (resp. in $j' < i < k'$) which is not in B and thus no nodes in B on some path $P : v_1 \rightsquigarrow v_2$ (resp. $P : v_2 \rightsquigarrow v_3$), again contradicting Lemma 1. This completes the proof $\qquad\square$

The graph $G(i, j, t, n)$. Given numbers i and j and the graph G_t^n, for some t and n, let $G(i, j, t, n)$ be the graph similar to G_t^n, except that it has an edge between each pair of nodes in $V_i \times V_j$. There is an illustration of $G(2, 5, 3, 18)$ in Figure 5.
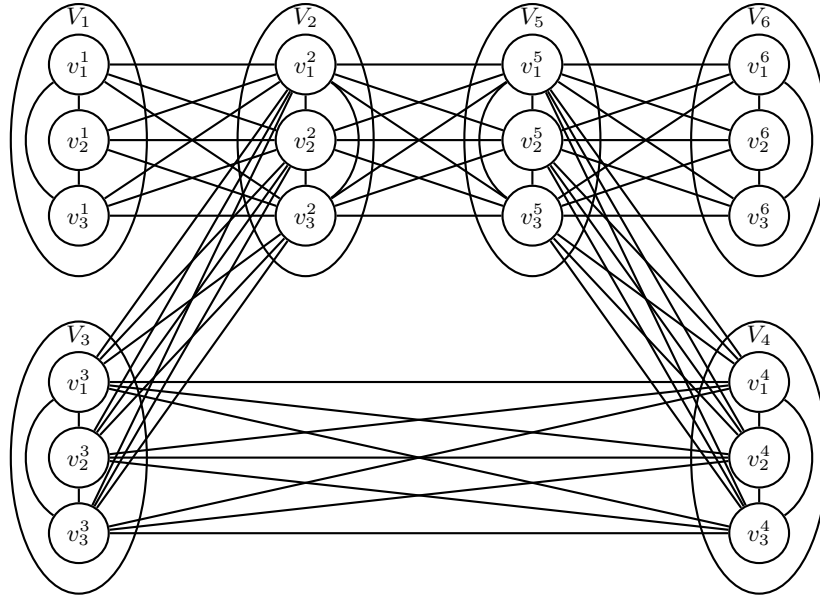

Fig. 5: The graph $G(2, 5, 3, 18)$.

The k-cops and robber game. For some integer k, the k-cops and robber game is a two-player zero-sum game on a graph G, as defined by Seymour and Thomas [29]. Initially, player 1 selects a set of nodes X_0 of size k, where he places a cop on each. Afterwards, player 2 selects a node $r_0 \in (V \setminus X_0)$ and puts the robber on this node. At this point round 1 begins. In round i, for each $i \geq 1$, first player 1 selects a set of nodes X_i of size k and then player 2 selects a

node r_i and a path $P : r_{i-1} \rightsquigarrow r_i$, such that for each node $v \in P$ we have that $v \notin (X_i \cap X_{i-1})$. If player 2 can not do so, player 1 wins, otherwise, the play continues with round $i + 1$. Player 2 wins in case player 1 never does.

As shown by [29], we have that player 1 has a winning strategy if and only if the treewidth of G is at most $k - 1$. Otherwise player 2 has a winning strategy and the treewidth of G is k or more.

We will next argue that the treewidth of $G(i, j, t, n)$ is a factor larger than the one of G_t^n.

Lemma 11. *For any i, j, t, n, where $j - i \geq 2$ the graph $G(i, j, t, n)$ has treewidth at least[4] $3 \cdot t - 1$.*

Proof. We will argue that player 2 has a winning strategy in the $(3t - 1)$-cops and robber game on $G(i, j, t, n)$. The main idea of player 2's winning strategy is to stay inside $V' = \bigcup_{\ell=i}^{j} V_\ell$. Since $j - i \geq 2$ we have that $|V'| \geq 3 \cdot t$. Observe that each node in V' has $3 \cdot t - 1$ adjacent nodes in V'. The strategy is as follows: Let r_0 be some node in $(V' \setminus X_0)$ (such a node exists, since $|X_0| = 3 \cdot t - 1$ and $|V'| \geq 3 \cdot t$). In each round i, if $r_{i-1} \notin X_i$, set $r_i = r_{i-1}$, otherwise, pick a node $r_i \in V'$ adjacent to r_{i-1} such that $r_i \notin X_i$ and let P be the path (r_{i-1}, r_i). Such a node exists, since r_{i-1} has $3 \cdot t - 1$ distinct adjacent nodes in V' (each different from r_{i-1}), $|X_i| = 3 \cdot t - 1$ and $r_{i-1} \in X_i$. In either case, the cops will not catch the robber in that round and thus not ever. $\square$

We will next argue that any tree-decomposition of G_t^n has either large height or large width.

Lemma 12. *For any n and t, consider a tree-decomposition $\mathrm{Tree}(G_t^n)$ of G_t^n of width t' and height h. Either $h \geq \frac{n}{2 \cdot t'}$ or $t' \geq 3 \cdot t - 1$.*

Proof. If $h \geq \frac{n}{2t'}$, we are done. Otherwise, let B be a bag and i, j numbers where $j - i \geq 2$ and such that $V_i \cup V_j \subseteq B$. Such a bag and numbers exists by Lemma 10. But then, $\mathrm{Tree}(G_t^n)$ is also a tree-decomposition of $G(i, j, t, n)$, since each edge between V_i and V_j is in B. We therefore get, by Lemma 11 that the width of $\mathrm{Tree}(G_t^n)$ is at least $3 \cdot t - 1$. $\square$

Theorem 3. *For numbers n and $t = o(n/\log n)$, the graph G_t^n has treewidth at most $2 \cdot t - 1$, but each balanced tree-decomposition of G_t^n has width at least $3 \cdot t - 1$.*

Proof. The bound on t implies that any tree-decomposition of width $t' < 3 \cdot t - 1$ and height h, such that $h \geq \frac{n}{2 \cdot t'}$ cannot be balanced. The statement then follows from Lemmas 9 and 12. $\square$

6 Local Reachability

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. Here we present an algorithm for computing "local reachability" in each bag, in particular, for each bag B and nodes $u, v \in B$, compute whether $(u, v) \in E^*$. Our algorithm requires $O(n \cdot t^2)$ time and $O(n \cdot t)$ space.

Use of the set-list data structure. We use various operations on a set data structure A that contains nodes from a small subset $V_A \subseteq V$ (i.e., $A \subseteq V_A \subseteq V$) of size bounded by $t + 1$, for t being the treewidth of G. Each set data structure A is represented as a pair of lists (L_1, L_2) of size $t + 1$ each. The list L_1 stores V_A in some predefined order on V, and the list L_2 is a binary list that indicates the elements of V_A that are in A. The initialization of A takes $O(t \cdot \log t)$ time, simply by sorting V_A in L_1, and initializing L_2 with ones in the indexes corresponding to elements in A. Intersecting two sets A_1, A_2, and inserting in A_1 all elements of $V_{A_1} \cap A_2$ takes $O(t)$ time, by simultaneously traversing the corresponding L_1 lists of the sets in-order.

---

[4] the treewidth of $G(i, j, t, n)$ is exactly $3t - 1$, but we will only show the lower bound

Forward and backward edges. Given a graph $G = (V, E)$ and its tree-decomposition $\mathrm{Tree}(G)$, we represent the edges of G as two sets for each node u, using the set-list data structure:

$$\mathrm{FWD}(u) = \{v : (u, v) \in E \text{ and } v \in B_u\}; \quad \mathrm{BWD}(u) = \{v : (v, u) \in E \text{ and } v \in B_u\}$$

Clearly, for all $u \in V$, we have $|\mathrm{FWD}(u)| \leq t + 1$ and $|\mathrm{BWD}(u)| \leq t + 1$. The following lemma states that the sets $\mathrm{FWD}(u)$ and $\mathrm{BWD}(u)$ store all edges in E. As a corollary, there are at most $2 \cdot n \cdot t$ edges in a graph G with treewidth t. It is well-known that a slightly stronger statement can be shown (i.e. the number of edges is $O(n \cdot t)$, but the hidden constant is below 2), but this statement suffices for our applications.

Lemma 13. *For all* $(u, v) \in E$, *we have* $v \in \mathrm{FWD}(u)$ *or* $u \in \mathrm{BWD}(v)$.

Proof. Consider some $(u, v) \in E$, such that $\mathsf{Lv}(v) \leq \mathsf{Lv}(u)$. By the definition of tree-decomposition, there exists some $B_i \in V_T$ such that $u, v \in B_i$. Then v appears in all bags B_j in the unique acyclic path $P : B_i \rightsquigarrow B_v$, and since $\mathsf{Lv}(v) \leq \mathsf{Lv}(u)$, the bag B_u appears in P. Hence $v \in B_u$ and $v \in \mathrm{FWD}(u)$. Similarly, if $\mathsf{Lv}(v) \geq \mathsf{Lv}(u)$, it follows that $u \in \mathrm{BWD}(v)$. $\square$

Local reachability. We first extend the definition of forward and backward edges to reachability, and then define the *local reachability* relation. 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 sets*

$$\mathrm{FWD}^*(u) = \{v : (u, v) \in E^* \text{ and } v \in B_u\}; \quad \mathrm{BWD}^*(u) = \{v : (v, u) \in E^* \text{ and } v \in B_u\}$$

i.e., $\mathrm{FWD}^*(u)$ (resp. $\mathrm{BWD}^*(u)$) is the set of nodes v that can be reached (resp. reach) u. Given a bag B, the local reachability relation is defined as

$$\mathsf{LR}(B) = \{(u, v) : u, v \in B \text{ and } v \in \mathrm{FWD}^*(u) \text{ or } u \in \mathrm{BWD}^*(u)\}.$$

Clearly, for all $u \in V$, we have $|\mathrm{FWD}^*(u)| \leq t + 1$ and $|\mathrm{BWD}^*(u)| \leq t + 1$. Given the sets $\mathrm{FWD}^*(u)$ and $\mathrm{BWD}^*(u)$ for all $u \in V$, the relation $\mathsf{LR}(B)$ can be constructed in $O(t^2)$ time, for each $B \in V_T$ (note that actually storing $\mathsf{LR}(B)$ explicitly for all $B \in V_T$ in total requires $\Omega(n \cdot t^2)$ space, which is beyond our space requirements). Similarly to Lemma 13 it can be shown that for every bag B and all pairs of nodes $u, v \in B$, $(u, v) \in E^*$ iff $u \in \mathrm{BWD}^*(v)$ or $v \in \mathrm{FWD}^*(u)$.

Subsuming tree-decomposition. Our algorithm LocReach for local reachability computation is more elegantly stated on a nice tree-decomposition. In order to apply LocReach on any tree-decomposition T, we first construct a nice tree-decomposition T' out of T, and then execute LocReach on T'. Here we describe a slightly technical construction of such a T' such that (i) T' uses asymptotically the same space as T, and (ii) the local forward and backward sets of T and T' are equal.

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 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 $B = B'$.

Claim 4. For every tree-decomposition $T = (V_T, E_T)$ with b bags and width t there exists a tree-decomposition $T' = (V_T', E_T')$ of $O(b)$ 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.

15

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 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 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 definitions 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'. $\qquad\square$

Algorithm LocReach. Given a graph G and a tree-decomposition $T = \text{Tree}(G)$ with $O(n)$ bags and width t, we present an algorithm for computing the local forward and backward sets. First, construct a nice tree-decomposition $T' = (V'_T, E'_T)$ of $O(n \cdot t)$ bags which subsumes T, using the construction of Claim 4. The computation is then performed as a two-way pass on T'. For each node $u \in V$ maintain two sets $\text{Fwd}'(u)$ and $\text{Bwd}'(u)$ using the set-list data structure, from the universe B_u. Initially set $\text{Fwd}'(u) = \text{Fwd}(u)$ and $\text{Bwd}'(u) = \text{Bwd}(u)$ for all $u \in V$. Given a bag B, define

$$\text{LR}'(B) = \{(u, v) : u, v \in B \text{ and } v \in \text{Fwd}'(u) \text{ or } u \in \text{Bwd}'(v)\}.$$

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 do as follows. For every pair of nodes $u, v \in B$ for which $(u, x), (x, v) \in \text{LR}'(B_x)$, if $\text{Lv}(u) \geq \text{Lv}(v)$ insert v in $\text{Fwd}'(u)$, otherwise insert u in $\text{Bwd}'(v)$.

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 do as follows. For every pair of nodes $u, v \in B$ for which $(u, v) \in \text{LR}'(B_x)$, if $v \in \text{Bwd}'(x)$ insert u in $\text{Bwd}'(x)$, and if $u \in \text{Fwd}'(x)$ insert v in $\text{Fwd}'(x)$.

In the following we establish that at the end of the second pass it holds that $\text{Fwd}'(u) = \text{Fwd}^*(u)$ and $\text{Bwd}'(u) = \text{Bwd}^*(u)$ for each $u \in V$. 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 of B_{x_i} is in $T(B)$. The following lemma captures the main intuition behind U-shaped paths.

Lemma 14. *Given a bag B and nodes $u, v \in B$ such that exists an (acyclic) path $P : u \rightsquigarrow v$ which is U-shaped in B, either $|P| = 1$ or $P = (u, y_1, \ldots, y_k, v)$ and P is U-shaped in B_x, where $x = \arg\min_i \text{Lv}(y_i)$.*

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 by Lemma 1 the subpath $y \rightsquigarrow x$ (or $x \rightsquigarrow y$) of P_1 would go through nodes of L of smaller level than $\text{Lv}(x)$, 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 $\text{Lv}(y) < \text{Lv}(x)$. 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 it follows that P is U-shaped in B_x.
□

Lemma 15. *For each node $u \in V$, the algorithm* LocReach *correctly computes the sets* $\text{Fwd}^*(u)$ *and* $\text{Bwd}^*(u)$.

Proof. It is clear that $\text{Fwd}'(u) \subseteq \text{Fwd}^*(u)$ and $\text{Bwd}'(u) \subseteq \text{Bwd}^*(u)$, that is, for every node v inserted in $\text{Fwd}'(u)$ (resp. $\text{Bwd}'(u)$) by the algorithm, we have $(u, v) \in E^*$ (resp. $(v, u) \in E^*$). The proof focuses on showing $\text{Fwd}^*(u) \subseteq \text{Fwd}'(u)$ and $\text{Bwd}^*(u) \subseteq \text{Bwd}'(u)$. Note that by the initialization of $\text{Fwd}'(u)$ and $\text{Bwd}'(u)$ we have $\text{Fwd}(u) \subseteq \text{Fwd}'(u)$ and $\text{Bwd}(u) \subseteq \text{Bwd}'(u)$. We first claim that after the first pass processes a bag B, for all $u, v \in B$ for which there exists a (acyclic) path $P : u \rightsquigarrow v$ that is U-shaped in B, we have $(u, v) \in \text{LR}'(B)$. 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 $|B| = 1$.
2. If B is not a leaf, by Lemma 14 either $|P| = 1$, or $P = (u, y_1, \ldots, y_k, v)$ and P is U-shaped in B_x, where $x = \arg\min_i \text{Lv}(y_i)$. If $|P| = 1$, the claim follows from the initialization of Fwd' and Bwd'. Otherwise, if $B_x \neq B$, the proof follows from the induction hypothesis, as 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 each such P_i is U-shaped in B, and by the same reasoning (i.e., by either $|P_i| = 1$ or the induction hypothesis) we get that $(u, x), (x, v) \in \text{LR}'(B)$. Hence, after LocReach processes B, it will hold that either $u \in \text{Bwd}'(v)$ or $v\text{Fwd}'(u)$, and thus $(u, v) \in \text{LR}'(B)$.

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 $\text{Fwd}^*(x) \subseteq \text{Fwd}'(x)$ and $\text{Bwd}^*(x) \subseteq \text{Bwd}'(x)$. 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 $|B_x| = 1$.
2. We now proceed inductively to some internal bag B_x examined by the algorithm in the second pass. We only focus on $\text{Fwd}'(x)$ (the argument is similar for $\text{Bwd}'(x)$). Consider any node v such that there exists a (acyclic) 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, thus by the first pass $(x, u) \in \text{LR}'(B_x)$. By condition T3 of the tree-decomposition, B_v is an ancestor of B_x, and hence the induction hypothesis applies to conclude that $v \in \text{Fwd}'(u)$ or $u \in \text{Bwd}'(v)$, and thus $(u, v) \in \text{LR}'(B_x)$. Hence, after the second pass processes B_x, we have $v \in \text{Fwd}''(x)$, as desired.

Figure 6 depicts the two passes. At the end of the computation, for all $x \in V$ we have $\text{Fwd}'(x) = \text{Fwd}^*(x)$ and $\text{Bwd}'(x) = \text{Bwd}^*(x)$, as desired.
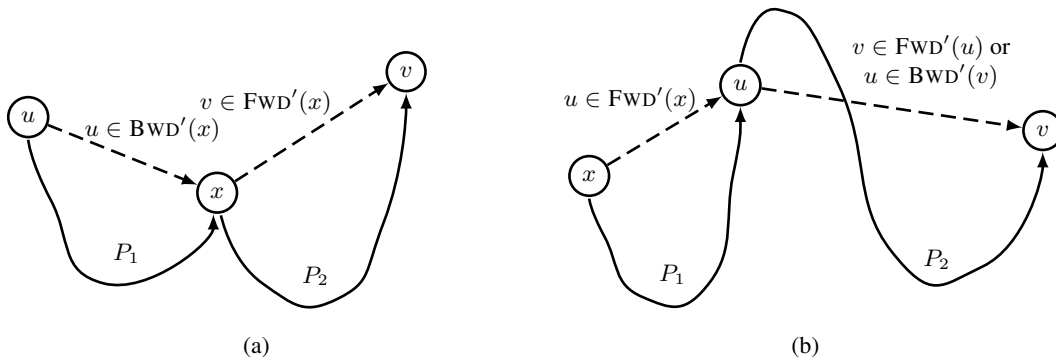□

Fig. 6: Illustration of the two passes for the local reachability computation

Lemma 16. *Algorithm* LOCREACH *requires* $O(n \cdot t^2)$ *time and* $O(n \cdot t)$ *space.*

Proof. By Claim 4, the construction of T' is done in $O(n \cdot t \cdot \log t)$ time and $O(n \cdot t)$ space. The algorithm LOCREACH 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, LOCREACH spends $O(t^2)$ time to iterate over all pairs $u, v \in B_x$, and $O(t)$ time to update each of the $O(t)$ FWD$'$ and BWD$'$ sets, hence it spends $O(t^2)$ time in total in B_x. There are n such bags B_x that are the root bags of a node x, hence the total time of LOCREACH 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'. $\qquad\square$

Theorem 4. *Given a graph* $G = (V, E)$ *and a tree-decomposition* $\mathrm{Tree}(G)$ *of* G *of width* t *and* $O(n)$ *bags, the algorithm* LOCREACH *correctly computes the local forward and backward sets, and uses* $O(n \cdot t^2)$ *time and* $O(n \cdot t)$ *space.*

Remark 1. Given a weight function $w : E \to \mathbb{Z}$, the two passes of LOCREACH can be easily modified to compute the *local distances* in each bag (i.e., for any pair of nodes u, v in a bag, the weight of the minimum weight $u \rightsquigarrow v$ path), with no time or space overhead. Informally, every time a node v is inserted to the FWD$^*(u)$ (or u to BWD$^*(v)$), we also insert a number r which corresponds to the weight of the minimum weight path between u and v, among all paths examined so far. Lemma 15 can be used to show that eventually all acyclic paths between u and v are considered, thus discovering the distance from u to v (or reporting that a negative cycle exists). The focus of the present work is on reachability, and these claims will not be presented formally.

7 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.

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. 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 indexes of nodes whose root bag is in $T(B)$ form a contiguous interval. Then, 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 indexes 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 parents. 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 relative indexes of nodes v in the ancestor bag B_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.

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 preprocesses it in order to answer 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. Obtain a binary, balanced tree-decomposition $T = \mathrm{Tree}(G)$ of G with $O(n)$ bags and width t (from Theorem 2), and preprocess T to answer LCA queries in $O(1)$ time (since T is balanced, this is standard).
2. Compute the local forward and backward sets of each node $u \in V$ (from Theorem 4).
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 indexes to those nodes in some arbitrary order. Additionally, store the number s_u of nodes whose root bags are in $T(B)$ with 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. Traverse T bottom-up, and for every bag B do as follows. For every pair of nodes $(u, v) \in \mathsf{LR}(B)$ such that B is the root bag of v and $i_u < i_v$, insert F_v to the segment $[i_v - i_u, i_v - i_u + s_v]$ of F_u (hence, the nodes reachable from v now become reachable from u, through v).
5. For every node u initialize two sequences of bit sets $(\mathsf{T}_u^i)_{0 \le i \le \mathsf{Lv}(u)}$, $(\mathsf{F}_u^i)_{0 \le i \le \mathsf{Lv}(u)}$, each of size b_{B_i}, where B_i is the ancestor of B_u at level i, and pack them into consecutive words.
6. Traverse T top-down, and maintain two sequences of bit sets $(\overline{\mathsf{T}}_x^i)_{0 \le i \le \mathsf{Lv}(B)}$ and $(\overline{\mathsf{F}}_x^i)_{0 \le i \le \mathsf{Lv}(B)}$ for every node x in the current bag B, where the size of $\overline{\mathsf{T}}_x^i$ and $\overline{\mathsf{F}}_x^i$ is the size b_{B_i} of the ancestor of B in level i. Initially, B is the root of T, 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 in $\mathrm{FWD}^*(x)$ (resp. $\mathrm{BWD}^*(x)$). For each encountered bag B, do as follows:
   (a) Delete all set sequences $(\overline{\mathsf{T}}_x^i)_i$ and $(\overline{\mathsf{F}}_x^i)_i$ for each $x \notin B$.
   (b) For each remaining set sequence of a node x, create a set $\overline{\overline{\mathsf{T}}}_x$ (resp. $\overline{\overline{\mathsf{F}}}_x$) of b_B 0s, and for every $w \in B$ such that $(x, w) \in \mathsf{LR}(B)$ (resp. $(w, x) \in \mathsf{LR}(B)$), set the l_w^B-th bit of $\overline{\overline{\mathsf{F}}}_x$ (resp. $\overline{\overline{\mathsf{T}}}_x$) to 1. Append the set $\overline{\overline{\mathsf{T}}}_x$ (resp. $\overline{\overline{\mathsf{F}}}_x$) to $(\overline{\mathsf{T}}_x^i)_i$ (resp. $(\overline{\mathsf{F}}_x^i)_i$).
   (c) For each $u \in B$ whose root bag is B initialize set sequences $(\overline{\mathsf{F}}_u^i)_i$ and $(\overline{\mathsf{T}}_u^i)_i$ of $a_B + b_B$ 0s 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 $w \in \mathrm{FWD}^*(u)$ (resp. $w \in \mathrm{BWD}^*(u)$), insert $(\overline{\mathsf{F}}_w^i)_i$ (resp. $(\overline{\mathsf{T}}_w^i)_i$) to $(\overline{\mathsf{F}}_u^i)_i$ (resp. $(\overline{\mathsf{T}}_u^i)_i$).
   (d) Finally, set $(\mathsf{F}_u^i)_i$ (resp. $(\mathsf{T}_u^i)_i$) equal to $(\overline{\mathsf{F}}_u^i)_i$ (resp. $(\overline{\mathsf{T}}_u^i)_i$).

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 all such i-th positions will be 1, as long as the path $P : u \rightsquigarrow v$ only visits nodes with certain indexes.

Lemma 17. *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 $P_2 : B \rightsquigarrow B_v$.

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 are done. Otherwise, B' is the root bag of y, and by the local distance computation, it is $(x, y) \in \mathsf{LR}(B')$. Additionally, by construction $i_x \le i_y$ and $s_x \ge s_y + i_y - i_x$, thus by the induction hypothesis, $i_x \le i_v \le i_x + s_x$. Then 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 $i_x \le i_v$ and the $i_v - i_x$-th bit of F_x is 1. If $x = u$ we are done. Otherwise, B_u is also the root bag of x, and 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 18. *At the end of preprocessing, for every node u and $v \in B_i$ where B_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 $l_v^{B_i}$-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 $B_i \rightsquigarrow B$, similarly to Lemma 17. □

Lemma 19. *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 tree-decomposition of G with $O(n)$ bags and width 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. We establish the complexity of each preprocessing step separately.

1. By Theorem 2, this step requires $O(\mathcal{T}(G)+n)$ time and $O(\mathcal{S}(G)+n)$ space for obtaining a binary, balanced tree-decomposition of $b = O(n)$ bags, $O(t)$ width, and height $h = O(\log b) = O(\log n)$. 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 4, this step requires $O(n \cdot t^2)$ time and $O(n \cdot t)$ space.
3. Every bag B is visited once, and all operations on B take constant time, hence this step requires $O(b \cdot t) = O(n \cdot t)$ time.
4. 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:

$$
\sum_{u \in V} \left\lceil \frac{s_u}{W} \right\rceil = \sum_{i=0}^{h} \sum_{u:\mathsf{Lv}(u)=i} \left\lceil \frac{s_u}{W} \right\rceil \leq \sum_{i=0}^{h} \sum_{u:\mathsf{Lv}(u)=i} \left(\frac{s_u}{W} + 1 \right)
$$

$$
= \frac{1}{W} \cdot \sum_{i=0}^{h} \sum_{u:\mathsf{Lv}(u)=i} s_u + \sum_{i=0}^{h} \sum_{u:\mathsf{Lv}(u)=i} 1 \leq \frac{1}{W} \cdot \sum_{i=0}^{h} n \cdot t + n = O(n \cdot t)
$$

   since $h = O(\log n)$ and $W = \Theta(\log n)$. Note that we have $\sum_{u:\mathsf{Lv}(u)=i} s_u \leq n \cdot t$ because $|\bigcup_u \mathsf{F}_u| \leq n$ and every element of $\bigcup_u \mathsf{F}_u$ belongs to at most t 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 $O(n \cdot t)$ for the set operations, by amortizing a constant number of operations per word used.
5. 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:

$$
\sum_{u \in V} 2 \cdot \left\lceil \frac{a_{B_u} + b_{B_u}}{W} \right\rceil \leq 2 \cdot n \cdot \left\lceil \frac{t \cdot h}{W} \right\rceil = O(n \cdot t)
$$

   since $a_{B_u} + b_{B_u} \leq t \cdot h$, $h = O(\log n)$ and $W = \Theta(\log n)$.
6. The space required is the space for storing the set sequences $(\overline{\mathsf{T}}_v^i)$ and $(\overline{\mathsf{F}}_v^i)$, 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 $(\overline{\mathsf{T}}_u^i)$ and $(\overline{\mathsf{F}}_u^i)$ 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 a non-0 entry.

Single-source query. Given a single-source query u, create a bit set A of size n, initially all 0s. Then start from B_u, and for every ancestor B_i of B_u at level i, for every node $v \in B_i$ whose root bag is B_i, if the l_v^B-th bit of F_u^i is 1, insert F_v to the segment $[i_v, i_v + s_v]$ of A. Report that the set of nodes v reached from u is those for which the i_v-th bit of A is 1.

Lemma 20. *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* $O\left(\frac{n \cdot t \cdot \log t}{\log n}\right)$ *time respectively.*

Proof. The correctness of the pair query comes immediately from Lemma 18 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. Let v be any node such that there is a path $P : u \rightsquigarrow v$, and let x be the node with the smallest index in P. It follows from Lemma 1 that B_x is an ancestor of B_u, and by Lemma 17, the $i_v - i_x$ bit in F_x will be 1. Hence when B_x is examined by the query phase of Reachability, F_x will be inserted in position i_x of A, and the i_v bit of A will be 1. This concludes the correctness of the single-source reachability query. Regarding the time complexity, there are at most t nodes u whose root bag is the current examined bag B_i at level i, and the size of each F_u is $\min\left(\frac{b \cdot t}{2^i}, n\right)$, hence:

$$\sum_{i=0}^{h} t \cdot \left\lceil \min\left(\frac{b \cdot t}{2^i}, n\right) \cdot \frac{1}{W} \right\rceil = t \cdot \left(\sum_{i=0}^{\log t} \left\lceil \frac{n}{W} \right\rceil + \sum_{i=\log t}^{h} \left\lceil \frac{b \cdot t}{2^i \cdot W} \right\rceil \right) = t \cdot \left(\frac{n \cdot \log t}{W} + \frac{b}{W} + h \right) = O\left(\frac{n \cdot t \cdot \log t}{\log n} \right)$$

since $h = O(\log n)$, $b = O(n)$ and $W = \Theta(\log n)$. □

Theorem 5. *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 tree-decomposition* $\mathrm{Tree}(G)$ *of $O(n)$ bags and width t on a standard RAM with wordsize $W = \Theta(\log n)$. The data-structure* Reachability *requires $O(\mathcal{T}(G) + n \cdot t^2)$ preprocessing time, $O(\mathcal{S}(G) + n \cdot t)$ preprocessing space, and correctly answers (i) pair reachability queries in $O\left(\left\lceil \frac{t}{\log n} \right\rceil \right)$ time, (ii) single-source reachability queries in $O\left(\frac{n \cdot t \cdot \log t}{\log n}\right)$ time.*

Remark 2. A single-source query can alternatively be answered by breaking it down to n pair queries, which requires $O\left(n \cdot \left\lceil \frac{t}{\log n} \right\rceil \right)$ time. Then the time required for a single-source query can be written as $O\left(\min\left(\frac{n \cdot t \cdot \log t}{\log n}, n \cdot \left\lceil \frac{t}{\log n} \right\rceil \right)\right)$.

Finally, for constant treewidth graphs we have that $\mathcal{T}(G) = O(n)$ and $\mathcal{S}(G) = O(n)$ [10], and thus along with Theorem 5 we obtain the following corollary.

Corollary 2. *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, (ii) single-source reachability queries in $O\left(\frac{n}{\log n}\right)$ time.*

References

1. T. Akiba, C. Sommer, and K. Kawarabayashi. Shortest-Path Queries for Complex Networks: Exploiting Low Tree-width Outside the Core. In *15th International Conference on Extending Database Technology (EDBT)*, pages 144–155, 2012.
2. 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.
3. R. Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.
4. O. Berkman and U. Vishkin. Finding level-ancestors in trees. *Journal of Computer and System Sciences*, 48(2), 1994.
5. M. Bern, E. Lawler, and A. Wong. Linear-time computation of optimal subgraphs of decomposable graphs. *Journal of Algorithms*, 8(2):216 – 235, 1987.

6. S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, 2006.

7. H. Bodlaender. Discovering treewidth. In *SOFSEM 2005: Theory and Practice of Computer Science*, volume 3381 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin Heidelberg, 2005.

8. H. L. Bodlaender. Dynamic programming on graphs with bounded treewidth. In *Automata, Languages and Programming*, volume 317 of *Lecture Notes in Computer Science*, pages 105–118. Springer Berlin Heidelberg, 1988.

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

10. H. L. Bodlaender, P. G. Drange, M. S. Dregi, F. V. Fomin, D. Lokshtanov, and M. Pilipczuk. An $O(c^k n)$ 5-Approximation Algorithm for Treewidth. *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, 0:499–508, 2013.

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

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

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

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

15. M. Elberfeld, A. Jakoby, and T. Tantau. Logspace versions of the theorems of Bodlaender and Courcelle. In *Foundations of Computer Science (FOCS), 2010 51st Annual IEEE Symposium on*, 2010.

16. M. Elberfeld, A. Jakoby, and T. Tantau. Algorithmic meta theorems for circuit classes of constant and logarithmic depth. In *29th International Symposium on Theoretical Aspects of Computer Science, STACS 2012*, 2012.

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

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

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

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

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

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

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

24. 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.

25. L. R. Planken, M. M. de Weerdt, and R. P. van der Krogt. Computing all-pairs shortest paths by leveraging low treewidth. In *Proceedings of the Twenty-first International Conference on Automated Planning and Scheduling (ICAPS-11)*, pages 170–177. AAAI Press, May 2011. Honourable mention for best student paper.

26. B. A. Reed. Finding approximate separators and computing tree width quickly. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Theory of Computing*, STOC '92, 1992.

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

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

29. P. Seymour and R. Thomas. Graph searching and a min-max theorem for tree-width. *Journal of Combinatorial Theory, Series B*, 58(1):22 – 33, 1993.

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

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

32. A. Yamaguchi, K. F. Aoki, and H. Mamitsuka. Graph complexity of chemical compounds in biological pathways, 2003.

A   Experimental results

We have applied our algorithm Reachability to a number of benchmarks of the DaCapo benchmark suit [6] that consist of real-world Java programs. Our results clearly demonstrate that our preprocessing is faster than the complete preprocessing (all-pairs reachability) and we obtain a significant speedup of our single-source query over the BFS, even by factors of 15-30 times faster. Each benchmark gives a collection of graphs (control-flow graph for methods of the program). We report the average size and the average treewidth of the graphs (with at least five hundred nodes) for each benchmark, and report the average running time over all pair queries, and all single-source queries (i.e., the average is over all possible queries). Note that the advantages of Reachability are demonstrated on relatively small graphs (n is small), which indicates that the hidden constants in the O-notations are small. The $O(\log n)$ improvement over DFS/BFS will be more pronounced for larger n. Our results are reported in Table 2. We thank Prateesh Goyal for a help with the implementation of our algorithm.

| | | | Preprocessing | | Query | | | |
| | | | | | Single | | Pair | |
	n	t	Our	Complete Preprocess	Our	No Preprocess	Our	No Preprocess
antlr	698	1.0	89027	136145	15.3	166.3	0.15	14.34
bloat	696	2.3	27597	54335	3.9	72.5	0.10	14.34
chart	1159	1.5	28887	90709	2.3	80.9	0.13	22.32
eclipse	656	1.6	44930	138905	6.7	239.1	0.19	15.76
fop	1209	1.7	36284	91795	2.9	60.6	0.12	43.0
hsqldb	698	1.0	73076	180333	13.0	219.0	0.14	13.89
jython	748	1.5	52176	68687	7.2	85.7	0.11	12.84
luindex	885	1.3	46212	142005	5.6	202.7	0.16	26.44
lusearch	885	1.3	63809	189251	12.8	211.4	0.13	26.01
pmd	644	1.4	37686	52527	2.5	83.9	0.13	12.5
xalan	698	1.0	70967	138420	8.0	235.0	0.19	14.28
Jflex	1091	1.6	60468	91742	3.1	50.8	0.11	20.46
muffin	1022	1.7	34733	66708	2.6	52.7	0.10	18.57
javac	711	1.8	43089	59793	4.8	75.2	0.11	11.86
polyglot	698	1.0	81762	150799	12.2	184.5	0.14	14.14

Table 2: Mean times in microseconds.