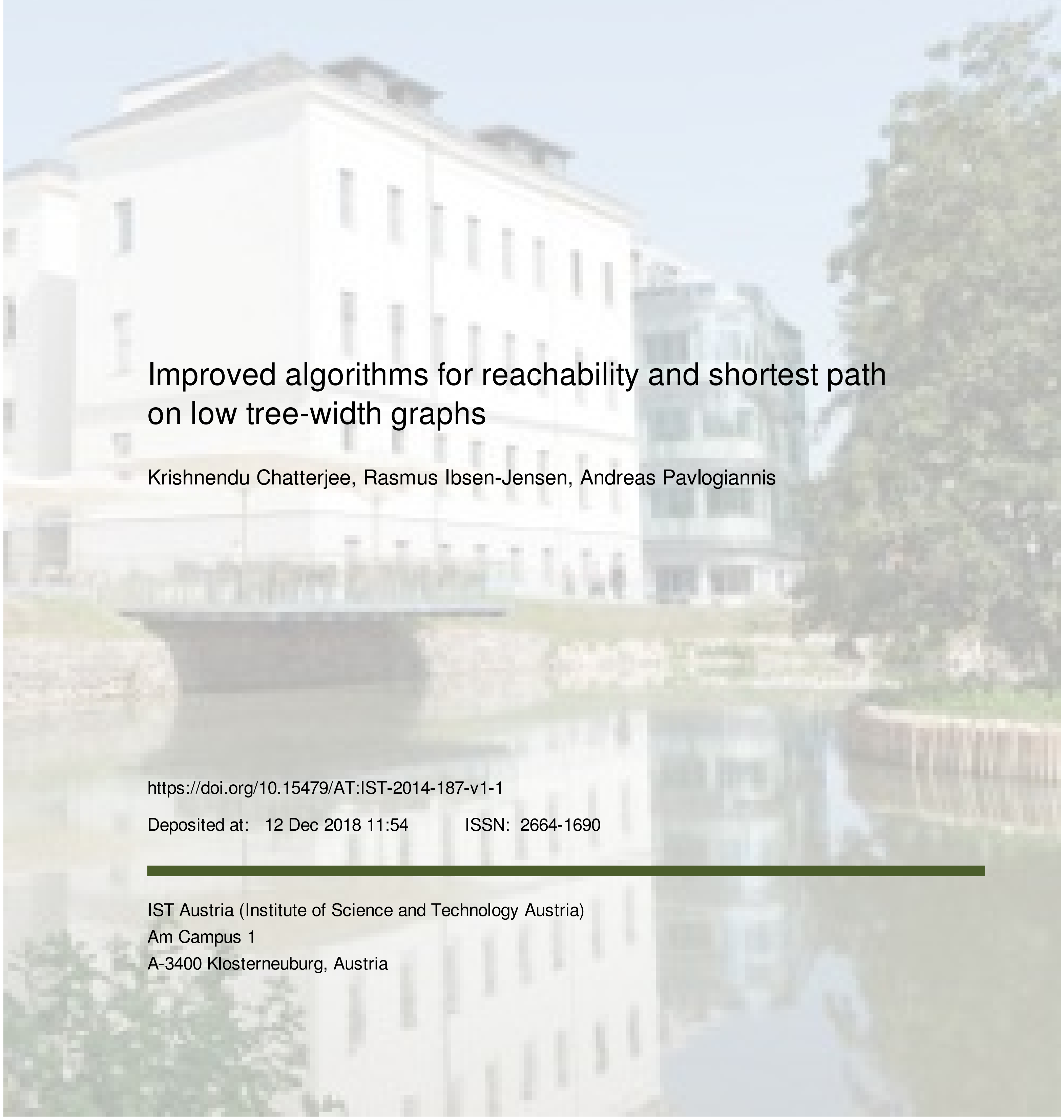

Improved algorithms for reachability and shortest path on low tree-width graphs

Krishnendu Chatterjee, Rasmus Ibsen-Jensen, Andreas Pavlogiannis

https://doi.org/10.15479/AT:IST-2014-187-v1-1
Deposited at: 12 Dec 2018 11:54 ISSN: 2664-1690

Improved Algorithms for Reachability and Shortest Path on Low Tree-width Graphs

Krishnendu Chatterjee and Rasmus Ibsen-Jensen and Andreas Pavlogiannis

Technical Report No. IST-2014-187-v1+1
Deposited at 14 Apr 2014 11:53
http://repository.ist.ac.at/187/1/main$_{full_tech.pdf}$

Improved Algorithms for Reachability and Shortest Path on Low Tree-width Graphs

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

[†] IST Austria

Abstract

We consider the reachability and shortest path problems on low tree-width graphs, with n nodes, m edges, and tree-width t, on a standard RAM with wordsize W. We use $\widetilde{O}$ to hide polynomial factors of the inverse of the Ackermann function. Our main contributions are three fold:

1. For reachability, we present an algorithm that requires $O(n \cdot t^2 \cdot \log(n/t))$ preprocessing time, $O(n \cdot (t \cdot \log(n/t))/W)$ space, and $O(t/W)$ time for pair queries and $O((n \cdot t)/W)$ time for single-source queries. Note that for constant t our algorithm uses $O(n \cdot \log n)$ time for preprocessing; and $O(n/W)$ time for single-source queries, which is faster than depth first search/breath first search (after the preprocessing).
2. We present an algorithm for shortest path that requires $O(n \cdot t^2)$ preprocessing time, $O(n \cdot t)$ space, and $\widetilde{O}(t^2)$ time for pair queries and $O(n \cdot t)$ time single-source queries.
3. We give a space versus query time trade-off algorithm for shortest path that, given any constant $\epsilon > 0$, requires $O(n \cdot t^2)$ preprocessing time, $O(n^\epsilon \cdot t^2)$ space, and $\widetilde{O}(n^{1-\epsilon} \cdot t^2)$ time for pair queries.

Our algorithms improve all existing results, and use very simple data structures.

Keywords: *Graph algorithms; Low tree-width graphs; Reachability and Shortest path.*

1 Introduction

In this paper we present improved algorithms for the reachability and shortest path problems on low tree-width weighted directed graphs, with n nodes, m edges, and tree-width t. We consider the problems on a standard RAM with wordsize W, which is assumed to be less than polynomial in n. For the purpose of least common ancestor queries and construction of pointers, we assume that W is $\Omega(\log n)$.

Reachability/shortest path problems. The *pair* reachability (resp. shortest path) 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 shortest path problem, such as, A^*-algorithm (pair) [18], Dijkstra's algorithm (single-source) [12], Bellman-Ford algorithm (single-source) [4, 15, 22], Floyd-Warshall algorithm (all pairs) [14, 27, 24], and Johnson's algorithm (all pairs) [19] and others for various special cases, there exist in essence only two different algorithmic ideas for reachability: Fast matrix multiplication (all pairs) [13] and depth first search/breath first search (DFS/BFS) (single-source) [11]. We will not explicitly compare our algorithms to the above generic algorithms, other than DFS/BFS, as our algorithms are much faster for low tree-width graphs.

Low tree-width graphs. A very well-known concept in graph theory is the notion of *tree-width* of a graph, which is a measure of how similar a graph is to a tree (a graph has tree-width 1 precisely if it is a tree). The tree-width of a graph is defined based on a *tree-decomposition* of the graph [16], see Section 2 for a formal definition. Beyond the mathematical elegance of the tree-width property for graphs, there are many classes of graphs which arise in practice and have low (even constant) tree-width. An important example is that the control flow graph for goto-free programs for many programming languages are of constant tree-width [26]. Also many chemical compounds have tree-width 3 [28]. For many other applications see the surveys [8, 6]. Given a tree-decomposition of a graph with low tree-width t, many problems on the graph become complexity-wise easier (i.e., many NP-complete problems for arbitrary graphs can be solved in time polynomial in the size of the graph, but exponential in t, given a tree-decomposition [3, 5, 7]). Even for problems that can be solved in polynomial time, faster algorithms can be obtained for low tree-width graphs, for example, for the shortest path problem [10].

Previous results. The algorithmic question of the shortest path (pair, single-source, all pairs) problem for low tree-width graphs has been considered extensively in the literature, and many algorithms have been presented [1, 10, 23]. The previous results are incomparable, in the sense that the best algorithm depends on the tree-width and the number of queries (see rows 1-4 of Table 1 for a summary of the existing results). No previous paper has explicitly considered the reachability problem on low tree-width graphs. This is likely because DFS/BFS achieve $O(m)$ time and $m \leq nt$.

Our results. In this work, we present several new algorithms for the reachability and shortest path problems for low tree-width graphs. Let $\alpha(n)$ be the inverse of the Ackermann function. For simplicity of presentation of our results, we hide polynomial factors of $\alpha(n)$ with the $\widetilde{O}$ notation. Our three main contributions are as follows:

1. *(Reachability).* For reachability, we present an algorithm that requires $O(n \cdot t^2 \cdot \log(n/t))$ preprocessing time, $O(n \cdot t \cdot \log(n/t))$ space, and answers pair queries in time $O(t/W)$ and single-source queries in time $O((n \cdot t)/W)$. Hence the algorithm answers single-source queries *faster* than DFS/BFS for tree-width smaller than the wordsize, after some preprocessing. While our algorithm achieves this using the so-called word-tricks, to the best of our knowledge, DFS/BFS have not been made faster using word-tricks. This result is in row i of Table 1.
2. *(Shortest path).* For shortest path, for every existing algorithm we present an algorithm that strictly improves the previous bounds (Theorems 4 to 6 and also Remark 4). Observe that while we present three algorithms, our algorithm in row ii of Table 1 is better than the existing results (up to some polynomial factors of $\alpha(n)$ and except for the low space result in row 4, for which row iii is better). The bounds are as follows:
   (a) In Theorem 4 (Section 5) we present an algorithm that requires $O(n \cdot t^2 \cdot \log(n/t))$ preprocessing time, $O(n \cdot t \cdot \log(n/t))$ space, and answers pair queries in $O(t)$ time. This strictly improves row 1 of Table 1.
   (b) In Theorem 6 (Section 7) we present an algorithm that requires $O(n \cdot t^2)$ preprocessing time, $O(n \cdot t)$ space, and $\widetilde{O}(t^2)$ pair query time. This corresponds to row ii of Table 1 and improves the results of

rows 1-3 of Table 1. We also have specific algorithms (Theorem 5 and also Remark 4) which improve the remaining existing algorithms, not improved by Theorem 6, without ignoring polynomial factors of $\alpha(n)$.

   (c) For row 4 in Table 1, our algorithm in row iii (details in the following item) is better, except that the assumption on t is slightly stronger (our assumption is is $t^{2+\epsilon} \leq n$ vs. $t^2 \cdot \log^2(n) \leq n$ of [1]): Note that the assumption $t^2 \cdot \log^2(n) \leq n$ is not explicitly mentioned in [1], but they construct components containing at least $t^2 \cdot \log^2(n)$ nodes, which implies that $t^2 \cdot \log^2(n) \leq n$.

3. *(Space and query time trade-off).* Finally, we present a space and query time trade-off algorithm for shortest path in Theorem 7 (Section 8), that given any constant $\epsilon > 0$, requires $O(n^\epsilon \cdot t^2)$ space and answers pair queries in time $\widetilde{O}(n^{1-\epsilon} \cdot t^2)$; and the preprocessing time required is $O(n \cdot t^2)$. While all previous algorithms use linear space in n, we present the first algorithm that uses sub-linear space in n and even ensures sub-linear in n query time. This contribution is shown in row iii of Table 1. We also present an algorithm with $O(\log^2(n) \cdot t^2)$ space that answers pair queries in time $O(n \cdot t^2 \cdot \log^2(n))$ (Remark 6, see row iv of Table 1). Our space and query time trade-off algorithms require oracle access to the graph and its tree-decomposition.

Moreover, all our algorithms use very basic data structures where sets are stored in the form of lists, except that we use a data structure by [25], simplifying [17], to find least common ancestors of nodes in (balanced binary) trees.

Important techniques. Our improvements are achieved by introducing several new techniques.

1. In a tree-decomposition, the bags of the tree, represent sets of nodes of the graph. The key intuition for row i of Table 1 is to modify the tree-decomposition into a balanced tree, such that for all nodes u and v, and for all paths P from u to v, the least common ancestor bag B_L of the bags B_u and B_v containing u and v, respectively, contain a node from P. Then we use bit-arrays to store which node in B_L can u reach and which can reach v.
2. The key intuition for row ii of Table 1:
   (a) One of our key improvements is a simple and fast algorithm for local distance computation (i.e., the distance between nodes that appear in the same bag of the tree-decomposition). The 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 find the local distance for all n nodes. Our algorithms also enable fast computation of single-source shortest path queries.
   (b) Row ii of Table 1 is build on [2] and local distance computation. The technique of [2] given a tree (in this case the tree-decomposition) splits the tree into components (subtrees) and then constructs a component tree out of the root bags of the components. Then the algorithm proceeds recursively on both the subtrees as well as the component tree. A direct application of the above technique leads to a problem where the number of nodes in the roots of the components is upto a factor of t higher than the number of components. Our trick is to make a factor of t less components and then handle the increase in the size of the subtrees using our fast single-source shortest path algorithm (based on local distance computation from above).
3. The simplicity of our local distance computation algorithm allows us to obtain a recursive version, for any $\epsilon > 0$, for the space and query time trade-off algorithm. The idea is that for a subset of bags (where the subset is not necessarily connected) of size close to n^ϵ/t bags, our algorithm can find the local distance in $O(n \cdot t^2)$ time and uses $O(n^\epsilon \cdot t)$ space. The key trick is that nothing is stored in the components, except $O(t^2)$ words for the roots.

Related works. Other than the papers on graphs with low tree-width (mentioned above), another related work is that of the same problem for bounded tree-width graphs but for the external memory model (aka the I/O model) [21]. While the result of [21] is not directly related to our work (since the I/O model is different from the RAM model) the paper presents an algorithm which is similar in idea to the local distance computation. Using the same definition of b, t and n as in Table 1, recalling that $b \geq n/t$, the differences are that our algorithm for local distances (and thus also the single-source shortest path algorithm) uses less space ($O(n \cdot t)$ versus $O(b \cdot t^2)$), and less time ($O(n \cdot t^2)$ versus $O(b \cdot t^3)$), because they run Floyd-Warshall algorithm [14, 27, 24] on each bag in their passes, instead of our more efficient two-step path shortenings.

Organization. In Section 2 we give formal definitions of tree-width and the problems we consider. In Section 3 we present algorithms for a more general class of graphs (in which separators exists). In Section 4 we give our local

Table 1: Algorithms for reachability and shortest path, pair queries and single-source queries, on a weighted directed graph G with n nodes, m edges, and a tree-decomposition of width t and b bags (and $b \geq n/t$). The model of computation is the standard RAM model with wordsize W. We use $\widetilde{O}$ to hide polynomial factors of $\alpha(n)$, which is the inverse Ackermann function. Rows 1-5 are previous results, and rows *i*-*iv* are the results of the paper.

Row	Preprocessing time	Space usage	Pair query time	Single-source query time	Assumptions	From
1	$O(n^2 \cdot t)$	$O(n^2)$	–	–	None	[23] [a]
2	$O(n \cdot t^4)$	$O(n \cdot t^4)$	$\widetilde{O}(t^4)$	$O(n \cdot t^4)$	None	[10]
3	Not given [b]	$O(b \cdot t^2)$	$O(t^2 \cdot \log\log n)$	$O(n \cdot t^2 \log\log n)$ [c]	None	[1]
4	Not given	$O(n)$ [d]	$O(t^3 \log^2(n) \cdot (t^2 + \log(t) \cdot \log\log(n)))$	–	$t^2 \cdot \log^2(n) \leq n$	[1]
5	–	$O(\lceil n/W \rceil)$	–	$O(m)$	None	DFS/BFS [11] [e]
i	$O(n \cdot t^2 \cdot \log(n/t))$	$O\left(n \cdot \left\lceil \frac{t \cdot \log(n/t)}{W} \right\rceil\right)$	$O(\lceil \frac{t}{W} \rceil)$	$O(\lceil \frac{n \cdot t}{W} \rceil)$	None	This paper [e]
ii	$O(n \cdot t^2)$	$O(n \cdot t)$	$\widetilde{O}(t^2)$	$O(n \cdot t)$	$t \leq \frac{n}{\alpha^2(n)}$	This paper
iii	$O(n \cdot t^2)$	$O(n^\epsilon \cdot t^2)$	$\widetilde{O}(n^{1-\epsilon} \cdot t^2)$	– [f]	$\epsilon > 0, t \leq \frac{n}{\alpha^2(n)}$	This paper
iv	–	$O(\log^2(n) \cdot t^2)$	$O(n \cdot t^2 \cdot \log^2(n))$	–	$t \leq \frac{n}{\alpha^2(n)}$	This paper

[a] This algorithm solves the all pairs problem in the given time and space bounds.
[b] The preprocessing consists of computing local distances, but no bound was given. Following the technique of our paper, see Theorem 2, this can be done in $O(n \cdot t^2)$ time and $O(n \cdot t)$ space.
[c] Obtained by multiplying the time for pair query by n.
[d] This is the space usage after preprocessing.
[e] Only for reachability queries.
[f] Not given since the size of the output is larger than the data-structure.

distance algorithm and a fast single-source shortest path algorithm, which are used in the later sections. Using our fast single-source shortest path algorithm together with the algorithm of Section 3, in Section 5 we present an algorithm for row i of Table 1. Then in Section 6 we present an algorithm which we modify in Section 7 to obtain row ii of Table 1. Finally, in Section 8 we present the algorithms for rows iii and iv of Table 1.

2 Definitions

Graphs, reachability, and shortest paths. 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, along with a weight function $\mathsf{wt} : E \to \mathbb{R}$ on the edges of G. We assume that $(u, u) \in E$ and $\mathsf{wt}(u, u) = 0$ for all $u \in V$. Given a set $X \subseteq V$, we denote with $G \restriction X$ the subgraph $(X, E \cap (X \times X))$ of G induced by the set X of nodes. A path $P : u \rightsquigarrow v$ is a sequence of nodes $(x_1, \dots, x_k)$ such that $u = x_1$, $v = x_k$, and for all $1 \leq i \leq k-1$ we have $(x_i, x_{i+1}) \in E$. The length of P is $k - 1$. A path P is *simple* if no node repeats in the path (i.e., it does not contain a cycle). 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 $P : u \rightsquigarrow v$. The weight function is extended to paths, and the weight of a path $P = (x_1, \dots, x_k)$ is $\mathsf{wt}(P) = \sum_{i=1}^{k-1} \mathsf{wt}(x_i, x_{i+1})$ if $|P| \geq 1$ else $\mathsf{wt}(P) = 0$. For $u, v \in V$, the distance $d(u, v) = \min_{P: u \rightsquigarrow v} \mathsf{wt}(P)$ is weight of the minimum weight simple path $P : u \rightsquigarrow v$, and if $(u, v) \notin E^*$, then $d(u, v) = \infty$. Given a path P and a set of nodes A, we denote with $A \cap P$ the set of nodes that appear in both P and A.

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

- Given nodes $u, v \in V$, the *pair reachability query* returns true iff $(u, v) \in E^*$.
- Given a node $u \in V$, the *single-source reachability query* returns the set $\{v : (u, v) \in E^*\}$ of nodes reachable from u.
- Given nodes $u, v \in V$, the *pair shortest path query* returns the distance $d(u, v)$ from u to v.

- Given a node $u \in V$, the *single-source shortest path query* returns the distance $d(u, v)$ from u to v, for all v such that $(u, v) \in E^*$.

Separators and separator families. For a number $k \in \mathbb{N}$, a *k-separator* for a graph $G = (V, E)$ is a pair (X, C), where $X \subseteq V$ with $|X| \leq k$, and $C = \{C_i : C_i \text{ is a CC in the graph } G \upharpoonright (V \setminus X)\}$ is the set of all maximal connected components (CCs) of the graph $G \upharpoonright (V \setminus X)$ (the CCs are SCCs (strongly connected components) in the undirected version of the graph). We sometimes simply call X a separator. For a pair of numbers $k \in \mathbb{N}$ and $0 < \Delta < 1$, a (k, Δ)-separator is a k-separator (X, C) such that for all $C_i \in C$ we have $|C_i| \leq n \cdot \Delta$. A *separator-family* $\mathcal{F}$ is a vertex-subset-closed family of graphs (i.e., if a graph G belongs to the family, then for each subset of nodes $V' \subseteq V$, the graph $G \upharpoonright V'$ also belongs to the family), for which there exist a monotone increasing function $g : \mathbb{N} \to \mathbb{N}$ and $\Delta > 0$ such that for each graph $G \in \mathcal{F}$ of n nodes and m edges, a $(g(n), \Delta)$-separator exists and can be computed in time $O(m \cdot g(n))$.

Merge operation on a set of subsets. We now define the *merge* set operation on a set of subsets, which is required for the definition of binary-separator hierarchies. Let $C = \{C_1, C_2, \dots, C_j\}$ be a set of pairwise disjoint subset of nodes each of size at most $\overline{n} \cdot \Delta$ such that their union consists of $\overline{n}$ nodes, i.e., each C_i is subset of V with $|C_i| \leq \overline{n} \cdot \Delta$, $|\bigcup_{1 \leq i \leq j} C_i| = \overline{n}$, and $C_i \cap C_k = \emptyset$ for $i \neq k$. The merge operation $\mathsf{Merge}(C)$ takes as argument C such that $|C| \geq 2$ and returns a pair of sets of nodes as follows: (1) If $C = \{C_1, C_2\}$ contains two sets with $|C_1| \leq |C_2|$, then return (C_1, C_2). (2) Otherwise, let C_1 and C_2 be two of the smallest size sets in C and $\widehat{C} = C_1 \cup C_2$, and let $C' = \{C_3, \dots, C_j, \widehat{C}\}$ be the set obtained by adding $\widehat{C}$ and removing C_1 and C_2 from C. Then return $\mathsf{Merge}(C')$.

Proposition 1. *Given a set $C = \{C_1, C_2, \dots, C_j\}$, where (1) $C_i \cap C_k = \emptyset$, for $i \neq k$; (2) $|\bigcup_{1 \leq i \leq j} C_i| = \overline{n}$; and (3) $|C_i| \leq \overline{n} \cdot \Delta$ for all i; let $\mathsf{Merge}(C) = (C_\ell, C_r)$. Then $|C_\ell| \leq |C_r| \leq \max(\Delta, \frac{2}{3}) \cdot \overline{n}$.*

Proof. There are two cases to consider: either $C_r \in C$ or C_r is a union of sets in C. If $C_r \in C$, then it has size at most $\overline{n} \cdot \Delta$. Otherwise, consider the last time that the union of sets were constructed to obtain C_r. At that time there were at least three pairwise disjoint sets, and hence C_r cannot contain more than $\frac{2}{3} \cdot \overline{n}$ elements, since each of the two smallest sets of at least three disjoint sets covering $\overline{n}$ elements cannot be larger than $\frac{\overline{n}}{3}$. $\square$

Binary-separator-hierarchies. For our algorithms, an important concept is that of *binary-separator-hierarchies*[1]. Consider a separator family $\mathcal{F}$ with a monotone function g. For a graph $G \in \mathcal{F}$ and a number $\ell \geq g(n)$, a ℓ-binary-separator-hierarchy is a tree, such that each node (which we call *bag*) consists of a subset of nodes of the graph of size at most ℓ. We give a recursive definition of a ℓ-binary-separator-hierarchy of G. If $n \leq \ell$, the root consists of all the nodes of G. Otherwise, if $n > \ell$, let (X, C) be a $(g(n), \Delta)$-separator of G and let $(C_\ell, C_r) = \mathsf{Merge}(C)$. The root consists of the nodes of X and the two subtrees under the root consist of ℓ-binary-separator-hierarchies defined on the graphs $G \upharpoonright C_\ell$ and $G \upharpoonright C_r$ respectively. For a bag B, we denote with $T(B)$ the subtree of the hierarchy rooted in B. When it is clear from the context, we use $T(B)$ to denote the set of nodes of G that appear in the bags in $T(B)$. We denote with $\mathsf{Lv}(B)$ the level of a bag B, defined as follows: For the root bag of the tree, the level is 0, and for all other bags in the tree, the level of the bag is 1 plus the level of the parent (i.e., distance from the root). Finally, we denote with B_u the unique bag in the separator-hierarchy that contains u, and with $\mathsf{Lv}(u) = \mathsf{Lv}(B_u)$ the level of its bag in the separator-hierarchy.

Tree-decomposition. Given a graph G, a tree-decomposition $\mathrm{Tree}(G) = (V_T, E_T)$ is a tree such that the following conditions hold:

1. $V_T = \{B_0, \dots, B_{n'-1} : \forall i \; B_i \subseteq V\}$ and $\bigcup_{B_i \in V_T} B_i = V$.
2. For all $(u, v) \in E$ there exists $B_i \in V_T$ such that $u, v \in B_i$.
3. 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 $B_i \cap B_j \subseteq B_k$.

[1]Note that a binary-separator-hierarchy is similar to a separator-hierarchy, but since we do not use regular separator-hierarchies we do not define them.

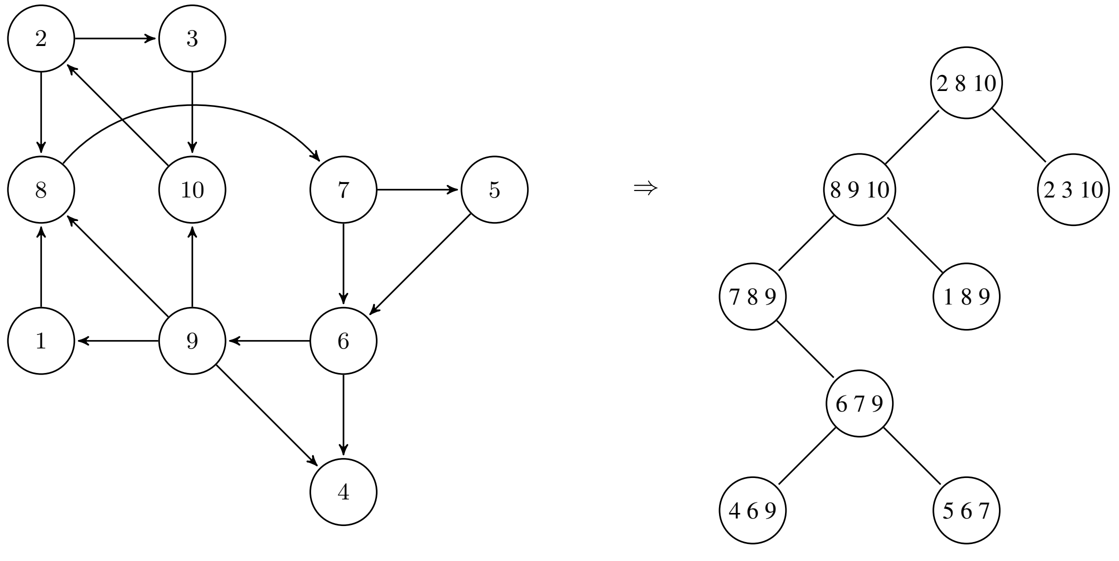

Figure 1: A graph G with tree-width 2 and a corresponding tree- decomposition $\mathrm{Tree}(G)$.

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. Let G be a graph, $T = \mathrm{Tree}(G)$, and B_0 be the root of T. Denote with $\mathsf{Lv}\,(B_i)$ the depth of B_i in T, with $\mathsf{Lv}\,(B_0) = 0$. For $u \in V$, we say that a bag B_i *introduces* u if B_i is the bag with the smallest level among all bags that contain u. By definition, there is exactly one bag introducing each node u. We often write B_u for the bag that introduces u, i.e., $B_u = \arg\min_{B_i \in V_T:\, u \in B_i} \mathsf{Lv}\,(B_i)$, and denote with $\mathsf{Lv}(u) = \mathsf{Lv}\,(B_u)$. See Figure 1 for an example of a graph and a tree-decomposition of it. We later use this example to illustrate our algorithms.

Remark 1. It follows from the above that every bag B is a $|B|$-separator (B, C) of G and the following condition holds: consider any path $B_i \rightsquigarrow B_j$ in $\mathrm{Tree}(G)$ such that B appears in the path, then for all $u \in (B_i)$ and $v \in (B_j)$, the nodes u and v do not appear in the same component in C, i.e., for all $C_i \in C$ either $u \notin C_i$ or $v \notin C_i$.

Semi-nice tree-decomposition. A tree-decomposition $T = \mathrm{Tree}(G)$ is called *semi-nice* if T is a binary tree, and every bag introduces at most one node. For every graph there exists a semi-nice tree-decomposition that achieves the tree-width of G and uses $n' = O(n)$ bags [20]. Given a tree-decomposition $T = \mathrm{Tree}(G)$ of a graph, we consider T as an undirected graph, and a *connected component* T_i is a set of bags in T such for every pair $B, B' \in T_i$ the unique path from B to B' in T only goes through bags in T_i. For every connected component T_i of a semi-nice tree-decomposition T, the tree obtained from T_i is a semi-nice tree-decomposition of the subgraph of G induced by the nodes that appear in the bags of T_i. In the sequel, when we consider tree-decomposition $\mathrm{Tree}(G)$ of a graph we only consider semi-nice tree-decomposition.

Remark 2. Given a semi-nice tree-decomposition $T = \mathrm{Tree}(G)$ that achieves tree-width t, every connected component T_i of T contains at most $|T_i| + t + 1$ nodes of G.

Word tricks and least common ancestor (LCA) queries. Our reachability algorithms (in Sections 3 and 5) use so called "word tricks" heavily. Without the word tricks, it naturally costs a factor of W in the query time and space. We also use constant time LCA queries which cannot be done without word tricks. Without word tricks, the LCA queries can be done with a binary search over the height of the separator hierarchies. Instead of constant time then the LCA queries require $O(\log\log(n/t))$ time (which is added to the query time), since the hierarchy is of height $O(\log(n/t))$; and amortized $O(1)$ additional pointers for each bag of the separator hierarchy.

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

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

where $\log^{(0)*} n = \log n$ and $\log^{(1)*} n = \log^* n$. Furthermore, we use the inverse of the Ackermann function [2]. The Ackermann function is:

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

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

Problem input. In the following sections we consider a weighted directed graph G and require that its semi-nice tree-decomposition $\mathrm{Tree}(G)$ is given. For algorithms to construct semi-nice tree-decomposition (or approximation of tree-decomposition) of graphs see [8, 6, 9]. We present several algorithms for preprocessing G such that pair and single-source reachability and shortest path queries are answered fast.

Negative cycles. In Section 3 we consider non-negative weight functions. In Section 4 and onward, we also allow negative weights, and the negative weights are handled as follows. If there is a negative cycle, then our algorithms report the existence of such a cycle, and if there is no negative cycle, then it computes the shortest paths. Algorithm LocDis in Section 4 reports negative cycles in $O(n \cdot t^2)$ time and $O(n \cdot t)$ space, and is the basis of the following algorithms (where we do not explicitly mention about negative cycles) (also see Remark 6).

3 Algorithms for Separator-families

In this section we present algorithms for graphs that belong to a separator-family $\mathcal{F}$ and for the algorithms of this section we consider only non-negative weights. In the later sections where we focus on low-tree width graphs, we will also consider negative weights. Since the main problem of interest is graphs with tree-width t, for some t, and they belong to separator-families where $g(n) = t + 1$, our algorithms do not attempt to find separators of separators. We first describe the preprocessing done by our algorithms (for shortest path and reachability) and then how to answer the queries. Finally we show the correctness and time and space usage of the algorithms.

Intuitive description. Intuitively, the idea is that any path $u \rightsquigarrow v$ must go through some node in a bag which is a common ancestor of the bags B_u and B_v in the binary-separator-hierarchy. We thus simply precompute the answer for all queries from/to each node in each bag B of the binary-separator-hierarchy to/from each node in $T(B)$, for paths in $G \upharpoonright T(B)$. This ensures that for a node w in the highest bag on a path $u \rightsquigarrow v$, the paths $u \rightsquigarrow w$ and $w \rightsquigarrow v$ have been precomputed. To answer a single-source query from u, we simply proceed upward in the separator hierarchy from B_u and for each ancestor B find the set of nodes Z_u in B that u can reach, together with the nodes that each $w \in Z_u$ can reach in $T(B)$. To answer a pair query (u, v) we proceed similarly, except that we only check if there is a node w such that $u \rightsquigarrow w$ and $w \rightsquigarrow v$. More formally, the algorithms are as follows:

Preprocessing. We describe the preprocessing for algorithms Reach (for reachability) and ShPat (for shortest path) that solve the corresponding problems.

1. Construct a $g(n)$-binary-separator-hierarchy S for G.
2. Create pointers from each node $v \in V$ to B_v, and from every bag B in S to the nodes in V that B contains.
3. (Only for Reach): Assign to each node v an index, based on a post-order *depth-first-search (DFS)* of the tree — the indices for nodes in a bag can be assigned in an arbitrary way inside that bag. Also, store a *map* $\mathbb{N} \to V$ from the index to the corresponding node. Store in each bag B the ranges of indices of the nodes that appear in $T(B)$ and in B.
4. Compute the number of nodes in each bag B and write it in B together with the sum of the number of nodes contained in the ancestors of B in S.

5. Preprocess S such that LCA queries can be answered in constant time (e.g. using [25]).

6. Then depending on the problem, do as follows:

   (a) (REACH): For each bag B of S at level i, for each node v in B, use DFS in $G \upharpoonright T(B)$ to find the nodes F'_v (*from* v) in $T(B)$ that v can reach and the nodes T'_v (*to* v) in $T(B)$ that can reach v. Then for each node $u \in T(B)$ let T^i_u contain the nodes $v \in B$ such that $u \in F'_v$ and let F^i_u contain the nodes $v \in B$ such that $u \in T'_v$. Store the sets T^i_u and F^i_u as bit-arrays in u, just after the sets T^{i-1}_u and F^{i-1}_u, (i.e., maybe partly in the same word) and the sets T'_v and F'_v in v.

   (b) (SHPAT): For each bag B of S, for each node v in B, use Dijkstra's algorithm [12] (note that in this section we consider non-negative weights) to find the distance function $\mathsf{f}(v) : T(B) \to \mathbb{R}$, such that $\mathsf{f}(v)(u)$ is the distance from v to u, for $u \in T(B)$, in the subgraph $G \upharpoonright T(B)$. Similarly, find the distance function $\mathsf{t}(v) : T(B) \to \mathbb{R}$, such that $\mathsf{t}(v)(u)$ is the distance from u to v, for $u \in T(B)$, in the subgraph $G \upharpoonright T(B)$. In either case we write $\mathsf{f}(v,u) = \mathsf{f}(v)(u)$ and $\mathsf{t}(v,u) = \mathsf{t}(v)(u)$.

Index interval property. The assignment of indices in Step 3 using DFS ensures that for every subtree T, the indices of the nodes contained in that subtree form an interval (i.e. there exist $a, b \in \mathbb{N}$, such that all indices of nodes in T are in $[a; b]$ and no index of any node outside T falls into that interval).

Pair reachability query. Given a pair $u, v \in V$, let L be the LCA bag of B_u and B_v in S. Let a be the number of nodes in ancestors of L and b be the number of nodes in L (this is stored in L). Compute the number c which is the boolean-AND of the first $a + b$ bits of F_u and T_v. Return true iff $c = 1$ (i.e., there exists a node w whose index is 1 in F_u and T_v, and hence there is a path $u \rightsquigarrow w \rightsquigarrow v$).

Single-source reachability query. Given a node $u \in V$, create an output bit-array of size $\lceil n/W \rceil$ words that consists of all 0's except that the index of u is 1. Then for $i = 0$ to $\mathsf{Lv}(u)$ consider F^i_u and the bag B at level i which is an ancestor of B_u. For each node v whose entry is 1 in F^i_u, boolean-OR the bit array F'_v with the part of the output-bit array whose indices correspond to nodes in F'_v. Note that such a part exists and the boolean-OR operation can be computed in time length divided by the wordsize, because the indicies of nodes in F'_v form an interval, the endpoints of which are stored in B. Afterwards, return the output bit array.

Pair shortest path query. Given a pair $u, v \in V$, start with L being the LCA bag of B_u and B_v in S, and let $c = \infty$. Then apply the following recursive procedure on (L, c): Let $c' = \min_{v' \in L} \mathsf{t}(v', u) + \mathsf{f}(v', v)$. If L is the root return $\min(c', c)$, otherwise let L' be the parent of L and recursively proceed with $(L', \min(c', c))$.

Single-source shortest path query. Given a node $u \in V$, first execute the single-source-reachability query from u, and then run the pair shortest path query, once for each reachable node v, and return the list of the distances.

Example 1. We consider the graph G given in Figure 1. The names of the nodes in G are also the indices they get assigned by the DFS. The algorithm preprocesses G and gets the binary-separator-hierarchy in Figure 2 and for each node store the corresponding bit-arrays shown in Figure 3.

To answer the pair reachability query of whether node 6 can reach node 5, the algorithm proceeds as follows: First find the LCA of the bags that contain node 6 and node 5 in the binary-separator-hierarchy. That is the bag that contains 6 and 7. Observe that there are 3 nodes in the ancestors (i.e. 8, 9, 10) and 2 nodes in (6 7). This is written down in preprocessing. Then boolean-AND the first $3 + 2 = 5$ bits of $F_6 = 11110$ and $T_5 = 111011$. We see that we get 11100. Since this is not all 0, the pair query of reachability from 6 to 5 returns true.

To answer the single-source reachability query for node 6, we consider $F_6 = 11110$. We then, for each 1 take the corresponding F'_v list and boolean-OR them together at the right place. The four 1's in F_6 corresponds to 8, 9, 10, 6, in that order (we start in the root of the binary-separator-hierarchy and proceed downward towards the bag that contains node 6) and $F'_8 = F'_9 = F'_{10} = 1111111111$ and $F'_6 = 1010$. We then boolean-OR F'_8, F'_9, F'_{10} together (over all bits) and then, in the result of 1111111111, we boolean-OR the bits corresponding to indices in $[4; 7]$ together with F'_6. We boolean-OR these bits because the subtree rooted at the bag (6 7) (which contains node 6), consists of 4,5,6,7. Recall that the indices of nodes that appear in some subtree rooted at some bag always form an interval. Our boolean-OR gives us 1111111111, which then means that node 6 can reach all nodes.

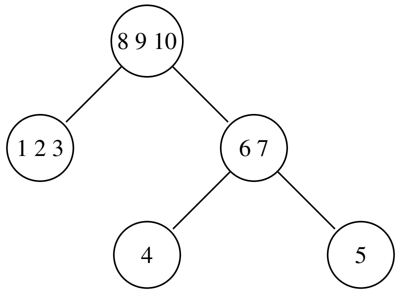

Figure 2: A binary-separator-hierarchy obtained from G, by the algorithm REACH.

v	F'_v	v	T'_v	v	F_v	v	T_v
1	100	1	100	1	111100	1	111100
2	011	2	010	2	111011	2	111001
3	001	3	011	3	111001	3	111011
4	1	4	1	4	000000	4	111111
5	1	5	1	5	111101	5	111011
6	1010	6	0111	6	11110	6	11111
7	1111	7	0001	7	11111	7	11101
8	1111111111	8	1110111111	8	111	8	111
9	1111111111	9	1110111111	9	111	9	111
10	1111111111	10	1110111111	10	111	10	111

Figure 3: Bit-arrays for the nodes of G, by the algorithm REACH.

Lemma 1. *The algorithm* REACH *(resp.* SHPAT*) correctly answers pair and single-source reachability (resp. shortest path) queries.*

Proof. First consider a query (u, v) and any path $P : u \rightsquigarrow v$. We argue that the query returns true (in the case of reachability) or a number not larger than $\mathsf{wt}(P)$ (in the case of shortest path). We first show the following claim:

Claim 1. *Let P be a $u \rightsquigarrow v$ path and let $w \in \mathsf{argmin}_{v' \in P} \mathsf{Lv}(v')$. We have that B_w is an ancestor of $B_{v'}$ in the separator-hierarchy, for all $v' \in P$.*

Proof. Assume towards contradiction that there exists some $v' \in P$ with B_w not being an ancestor of $B_{v'}$. By the choice of w, we have that $B_{v'}$ is not an ancestor of B_w either. Then there exists a common ancestor B of both, for which B_w and $B_{v'}$ appear in the left and right subtree of B. Since B is a separator of nodes in its left and right subtree, there exists some $w' \in B \cap P$ which contradicts the choice of w, because $\mathsf{Lv}(w') < \mathsf{Lv}(w)$. □

If $(u, v) \in E^*$, then the pair query returns true. Since we have that each $P : u \rightsquigarrow v$ consists only of nodes in $T(B_w)$, where B_w is the bag highest in the separator hierarchy according to Claim 1, the algorithm must return true (in the case of reachability) or a number at most $\mathsf{wt}(P)$ (in the case of shortest path). More specifically about reachability: Let $\mathsf{Lv}\,(B_w) = i$ and j be the index of w. Then bit j is 1 in T^i_u iff u can reach w while staying in $T(B_w)$. Similar reasoning holds for F^i_v. Then F^i_u boolean-AND T^i_v cannot be 0, since there is a path going through w which stays in $T(B_w)$. Also, the algorithm computed a boolean-AND of F^i_u and T^i_v as part of the partial boolean-AND of F_u and T_v (since B_w is a common ancestor and thus above the LCA bag of B_u and B_v in the separator hierarchy). Similarly for shortest paths, we can conclude that $\mathsf{t}(u, w) + \mathsf{f}(w, v) \leq \mathsf{wt}(P)$.

If the pair query returns true, then $(u, v) \in E^*$. Consider a query (u, v) that returns true (resp. a number c). We argue that there is a path between u and v (resp. and of weight c). We only argue about reachability, and the argument is similar for shortest path. Since the query returned true, there is some i such that F_u^i boolean-AND T_v^i is not 0 and such that $i \leq \mathsf{Lv}(L)$, for L being the LCA of B_u and B_v. Let w be some node with index equal to a non-zero bit of the boolean-AND. We see that $u \in T'_w$ and $v \in F'_w$ (because otherwise the corresponding bit could not be 1 in both T_u^i and F_v^i). Hence there is a path from u to w to v, contained in $T(B_w)$.

Single-source reachability queries. Consider a single-source reachability query from a node u. We argue that the j-th bit in the output array is 1 iff u can reach the node w with index j. First, if the j-th bit is 1, then at some point of the preprocessing, there must have been some bag B_v in level i such that (1) B_v is an ancestor of B_u; and (2) u can reach v (by construction of F_u^i); and (3) v can reach w (by construction of F'_v). Hence, by transitivity $(u, w) \in E^*$. Conversely, if $(u, w) \in E^*$, consider some path $P : u \rightsquigarrow w$, and let $v = \mathsf{argmin}_{v' \in P} \mathsf{Lv}(v')$. Then, clearly u and w belong to $T(B_v)$ and thus F_u^i contains w, for $i = \mathsf{Lv}(B_w)$, and F'_w contains v.

Single-source shortest path queries. The correctness of single-source shortest path follows directly from the correctness of the pair and of the reachability version. □

Theorem 1. *Consider a graph G with n nodes and m edges, in a separator family $\mathcal{F}$, with function $g(n)$. Let W be the wordsize of the RAM and define the function h as $h(n) = g(n)$ if $g(n)$ is polynomial in n and $h(n) = g(n) \cdot \log(n/g(n))$ otherwise. The algorithm* REACH *(resp.* SHPAT*) correctly answers pair and single-source reachability (resp. shortest path) queries and requires*

- $O(m \cdot h(n))$ *(resp. $O((m + n \cdot \log n) \cdot h(n))$) preprocessing time;*
- $O(n \cdot \lceil h(n)/W \rceil)$ *(resp. $O(n \cdot h(n))$) space;*
- $O(\lceil h(n)/W \rceil)$ *(resp. $O(h(n))$) pair reachability (resp. shortest path) query time; and*
- $O(n \cdot \lceil g(n)/W \rceil)$ *(resp. $O(n \cdot g(n))$) single-source reachability (resp. shortest path) query time.*

Proof. The correctness of the queries follows from Lemma 1. For each i, let Φ^i be an upper bound on the number of nodes in $T(B)$ for any bag B at level i. We have that $\Phi^i \leq n \cdot (\max(\Delta, \frac{2}{3}))^i$ by applying Proposition 1 recursively.

We first prove the following claim:

Claim 2. *Summing $g(\Phi^i)$ over all $\ell = O(\log(n/g(n)))$ levels, we have that $\sum_{i=0}^{\ell} g(\Phi^i) = O(h(n))$.*

Proof. If $g(n)$ is a polynomial in n we have that $\sum_{i=0}^{\ell} g(\Phi^i) \leq \sum_{i=0}^{\ell} (\Delta')^i \cdot g(n) = O(g(n)) = O(h(n))$, for $0 < \Delta' < 1$, since Φ^i forms an increasing geometric sequence and the last element in the sequence is n. Otherwise, if g is not polynomial, we get from monotonicity that $\sum_{i=0}^{\ell} g(\Phi^i) \leq g(n) \cdot \log(n/g(n)) = h(n)$. □

Preprocessing time. We consider each step on its own:

1. *Time usage of Step (1):* The construction of internal bags at level i in the separator-hierarchy takes $O(m \cdot g(\Phi^i))$ time. This is because each edge is in at most one subtree at level i and each subtree consists of at most Φ^i nodes. Summing over all levels we get, using Claim 2, that we use $O(m \cdot h(n))$ time.
2. *Time usage of Step (2):* Step (2) can clearly be done in $O(n)$ time.

3, 4. *Time usage of Steps (3) and (4):* Step (3) can be done in $O(n)$ time, since the time for the DFS in the separator hierarchy is $O(n)$, and there is one array entry per node in the whole tree. Step (4) can be incorporated in the DFS of Step (3), with constant time spent in each bag.

5 *Time usage of Step (5):* Using algorithms by [25], this can be done in linear time in the size of S (which is at most the size of G).

6 *Time usage of Step (6):* Let $\theta = O(m)$ be the cost of DFS in the case of preprocessing for reachability, and $\theta = O(m + n \cdot \log n)$ the cost of Dijkstra's algorithm [12] in the case of shortest paths. For each i, every node and edge appears in at most one subtree rooted at level i, hence the total cost for either operation in that level is $O(g(\Phi^i) \cdot \theta)$. Using Claim 2 we therefore get a total running time of $O(\theta \cdot h(n))$ over all $O(\log(n/g(n)))$ levels.

We see that each of the six steps requires $O(\theta \cdot h(n))$ time. Thus the preprocessing time is as desired.

Space usage for reachability. The binary-separator-hierarchy is a tree and has at most n bags (because each node of G appears in a unique bag in the hierarchy). Each node v has one pointer to the bag B_v, one word, the index and a pointer from the map (an array over the indices), and the bit-arrays F_v and T_v (and possibly F'_v and T'_v, if B_v is an internal bag in the hierarchy). Each bit-array consists of $O(h(n))$ entries by Claim 2, because we use $g(n_i)$ bits for level i, where n_i is the number of nodes in the separator at level i which is an ancestor of B_v. We can divide the size of the bit-arrays with W, by packing the bits into words. We see that we use at most $O(n \cdot \lceil h(n)/W \rceil + n) = O(n \cdot \lceil h(n)/W \rceil)$ space.

Space usage for shortest path. The space usage for shortest path is similar to the one for reachability, except that we cannot divide by W and hence, get that we use $O(n \cdot h(n))$ space.

Query time: pair reachability. On a query (u, v), we first use the LCA query structure to find the LCA bag L of B_u and B_v, and do some boolean operations on the words in L. Afterwards, the algorithm computes a boolean-AND over a prefix of T_u and F_v and returns if that is not all 0. The length of T_u and F_v is $O(h(n))$ each using Claim 2 and thus it is also a bound on the length of the prefix. We can do the operation in time $O(\lceil h(n)/W \rceil)$ (because both boolean-AND and checking if a number is 0 can be done in constant time per word).

Query time: single-source reachability. Consider some level i and let Φ^i be an upper bound on the number of nodes in $T(B)$, where B is the ancestor of B_u in that level. We split into two cases. Either $\Phi^i \geq W$ or not. If not, we can do the boolean-OR in time $O(g(n))$, otherwise we can do it in time $O(\Phi^i \cdot g(n)/W)$. Thus we get a total time of at most $\sum_i^\ell O(\Phi^i \cdot g(n)/W + g(n)) = (O(n \cdot g(n)/W + \ell \cdot g(n)) = O(n \cdot g(n)/W)$, similar to Claim 2, as desired.

Query time: pair shortest path. On a query (u, v), we first use the LCA query structure to find the LCA bag L of B_u and B_v, and let $\mathsf{Lv}(L) = i$. We then for each node v' in L find $\mathsf{t}(v', u)$ and $\mathsf{f}(v', v)$ and the sum $\mathsf{t}(v', u) + \mathsf{f}(v', v)$. For each node v', this can be done in constant time. We then do the same all the way up the tree. In level i there are at most $g(\max(\Delta, \frac{2}{3})^i \cdot n)$ nodes in the separator of that level and hence, using an argument similar to Step (1) of preprocessing time, we get that we use at most $O(h(n))$ time following Claim 2.

Query time: single-source shortest path. For any node u, and for each i, we have at most Φ^i nodes v, such that v has a LCA bag with u below level i. Hence, we get a total time of

$$\sum_{i=0}^{\ell} g(\Phi^i) \cdot \Phi^i \leq g(n) \cdot n \cdot \sum_{i=0}^{\ell} \frac{\Phi^i}{n} = O(n \cdot g(n)) \ .$$

The inequality is obtained as follows: since $\Phi^i \leq (\Delta')^i \cdot n$, for some constant $0 < \Delta' < 1$ for all i, the geometric sum $\sum_{i=0}^{\ell} \frac{\Phi^i}{n}$ is bounded by a constant; and moreover, since $\Phi^i \leq n$ by the monotonicity of $g()$, we have $g(\Phi^i) \leq g(n)$. The desired result follows. $\square$

It is straightforward to find $(t+1, \frac{2}{3})$-separators in a semi-nice tree-decomposition using DFS (see the $f(k)$-partitioning in Section 6 for an explicit construction). From Theorem 1 we obtain the following corollary.

Corollary 1. *Given a (weighted) graph $G = (V, E)$ of n nodes and m edges and a semi-nice tree-decomposition $\mathrm{Tree}(G)$ of G of width t that consists of $O(n \cdot t^2)$ bags, let $\widehat{t} = t \cdot \log(n/t)$. Let W be the wordsize of the RAM. There exist algorithms for reachability (resp. shortest path) that require*

- *$O(n \cdot \widehat{t} \cdot t)$ (resp. $O(n \cdot \widehat{t} \cdot (t + \log n))$) preprocessing time;*
- *$O(n \cdot \lceil \widehat{t}/W \rceil)$ (resp. $O(n \cdot \widehat{t})$) space;*

- *$O(\lceil \widehat{t}/W \rceil)$ (resp. $O(\widehat{t})$) pair reachability (resp. shortest path) query time; and*
- *$O(\lceil n \cdot t/W \rceil)$ (resp. $O(n \cdot t)$) single-source reachability (resp. shortest path) query time.*

Proof. Any graph with tree-width t has at most $O(n \cdot t)$ edges (this is well-known, but it also follows from Lemma 2). Furthermore, a bag B can be found in linear time in the size of G, such that the CCs in $G \upharpoonright (V \setminus B)$ each contains at most $\frac{2 \cdot n}{3}$ nodes. Note that the nodes of B then form a $(t+1, \frac{2}{3})$-separator. We construct a tree-decomposition for each of the children in total time linear in the size of the tree-decomposition, by simply removing B and the nodes of B from the tree-decomposition. This forms a forest, where each child bag is the root of one of the trees. We are therefore able to proceed recursively, and Theorem 1 applies to obtain the desired bounds. ☐

4 Local Distance and Single-source Shortest Path for Low Tree-width

In this section we present two results: computing local distances and improved preprocessing time for single-source shortest path for low tree-width graphs. The second result follows easily from the first. Moreover, we present the results for weight function with negative weights, and the result of this section will be used in the later sections.

4.1 Local distance computation

Consider a graph $G = (V, E)$, with tree-decomposition $(V_T, E_T) = \mathrm{Tree}(G)$, of width t. Here we present an algorithm to compute“local” distances, in the sense that we compute $d(u, v)$, for all u, v such that $u, v \in B$ for some bag $B \in V_T$. Our algorithm requires $O(n \cdot t^2)$ time and $O(n \cdot t)$ space. A similar result was shown in [10, Lemma 3.2], except that our algorithm is a factor of t^2 faster.

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 tree-width 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 indices 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. In some cases, L_2 will not be a binary list, but a list over $\mathbb{R} \cup \{\infty\}$, that associates each element of V_A with a real number (or infinity).

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

$$\textsc{Fwd}(u) = \{(v, \mathsf{wt}(u,v)) : \ (u,v) \in E \text{ and } v \in B_u\}\,;$$
$$\textsc{Bwd}(u) = \{(v, \mathsf{wt}(v,u)) : \ (v,u) \in E \text{ and } v \in B_u\}\,.$$

Clearly, for all $u \in V$, we have $|\textsc{Fwd}(u)| \leq t+1$ and $|\textsc{Bwd}(u)| \leq t+1$. The following lemma states that the sets $\textsc{Fwd}(u)$ and $\textsc{Bwd}(u)$ store all edges in E. As a corollary, there are at most $2 \cdot n \cdot t$ edges in a graph G with tree-width 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 constant hidden is below 2), but this statement suffices for our applications.

Lemma 2. *For all $(u,v) \in E$, we have $(v, \mathsf{wt}(u,v)) \in \textsc{Fwd}(u)$ or $(u, \mathsf{wt}(u,v)) \in \textsc{Bwd}(v)$.*

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 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 $(v, \mathsf{wt}(u,v)) \in \textsc{Fwd}(u)$. Similarly, if $\mathsf{Lv}(v) \geq \mathsf{Lv}(u)$, it follows that $(u, \mathsf{wt}(u,v)) \in \textsc{Bwd}(v)$. ☐

Local distances. We first extend the definition of forward and backward edges to distances, and then define *local distances*. Given a tree-decomposition $T = \text{Tree}(G)$ of a graph G and a node $u \in V$, we define the local forward set $\textsc{Fwd}^*(u) = \{(v, d(u, v)) : v \in B_u\}$, and the local backward set $\textsc{Bwd}^*(u) = \{(v, d(v, u)) : v \in B_u\}$, i.e., $\textsc{Fwd}^*(u)$ (resp. $\textsc{Bwd}^*(u)$) is the set of forward (resp. backward) distances to and from nodes that appear in the bag that introduces u. Given a bag B, we define the local distance relation as:

$$\mathsf{LD}\,(B) = \{(u, v, z) : u, v \in B \text{ and } (v, z) \in \textsc{Fwd}^*(u) \text{ or } (u, z) \in \textsc{Bwd}^*(u)\}.$$

Clearly, for all $u \in V$, we have $|\textsc{Fwd}^*(u)| \leq t + 1$ and $|\textsc{Bwd}^*(u)| \leq t + 1$. Given the sets $\textsc{Fwd}^*(u)$ and $\textsc{Bwd}^*(u)$ for all $u \in V$, the relation $\mathsf{LD}\,(B)$ can be constructed in $O(t^2)$ time, for all $B \in V_T$ (note that actually storing $\mathsf{LD}(B)$ explicitly for all $B \in V_T$ in total requires $\Omega(n \cdot t^2)$ space, which is beyond our space requirements). The following lemma states that the sets $\textsc{Fwd}^*$ and $\textsc{Bwd}^*$ store all distances between nodes that appear in the same bag.

Lemma 3. *For all $B \in V_T$ and all $u, v \in B$, we have $(u, v, d(u, v)) \in \mathsf{LD}\,(B)$.*

Proof. Similar to Lemma 2. □

Algorithm $\textsc{LocDis}$**.** Given $T = \text{Tree}(G)$ for some graph G, we present the algorithm $\textsc{LocDis}$ (for local distances) for computing the local forward and backward sets. The computation is performed as a two-way pass. For each node $u \in V$ maintain two sets $\textsc{Fwd}'(u)$ and $\textsc{Bwd}'(u)$ of pairs over $B_u \times (\mathbb{R} \cup \{\infty\})$, using the set-list data structure. For each bag B and $u, v \in B$, we write $d'(u, v)$ for the value z such that $(v, z) \in \textsc{Fwd}'(u)$ or $(u, z) \in \textsc{Bwd}'(v)$. Initially set $\textsc{Fwd}'(u) = \textsc{Fwd}(u)$ and $\textsc{Bwd}'(u) = \textsc{Bwd}(u)$ for all $u \in V$. Then, perform the following passes.

1. *First pass.* Traverse T level by level starting from the leaves (bottom up), and for each encountered bag B_x that introduces some $x \in V$, execute the following steps. For every pair of nodes $u, v \in B_x$ with $\mathsf{Lv}(u) > \mathsf{Lv}(v)$, update the value of v in $\textsc{Fwd}'(u)$ with $z = d'\,(u, x) + d'\,(x, v)$, if z is smaller than the current value of v in $\textsc{Fwd}'(u)$. Similarly for v in $\textsc{Bwd}'(u)$ and $z = d'\,(v, x) + d'\,(x, u)$, if z is smaller than the current value of v in $\textsc{Bwd}'(u)$.

2. *Second pass.* Traverse T level by level starting from the root (top down), and for each encountered introduce bag B_x that introduces some $x \in V$, execute the following steps. For each $v \in B_x$, update the value of v in $\textsc{Fwd}'(x)$ with $z = \min_{u \in B_x} (d'(x, u) + d'(u, v))$ if $z < d'(x, v)$. Similarly for v in $\textsc{Bwd}'(x)$ and $z = \min_{u \in B_x} (d'(v, u) + d'(u, x))$ if $z < d'(v, x)$.

As the following lemma shows, at the end of the second pass it holds that $\textsc{Fwd}'(u) = \textsc{Fwd}^*(u)$ and $\textsc{Bwd}'(u) = \textsc{Bwd}^*(u)$ for each $u \in V$.

Lemma 4. *For each node $u \in V$, the algorithm* $\textsc{LocDis}$ *correctly computes the sets* $\textsc{Fwd}^*(u)$ *and* $\textsc{Bwd}^*(u)$.

Proof. It is clear that whenever the algorithm processes a bag B and updates the set $\textsc{Fwd}'(u)$ or $\textsc{Bwd}'(u)$ of some $u \in B$ with a pair (v, z), then $v \in B$ and z is the weight of a $u \rightsquigarrow v$ path. Given a path $P : x_1, \ldots, x_k$, we say that P is U-shaped if $\mathsf{Lv}(x_i) \geq \mathsf{Lv}(x_1), \mathsf{Lv}(x_k)$ for all $1 < i < k$.

After the first pass processes a bag B_x that introduces some node x, for all $u, v \in B_x$, it holds that $d'(u, v) \leq \mathsf{wt}(P)$, where P is any U-shaped path such that for every intermediate node y it holds $\mathsf{Lv}(y) \geq \mathsf{Lv}(x)$ (i.e., B_y is a descendant of B_x). The claim follows by an easy induction: (1) It is trivially true for B_x being a leaf of T, and (2) if B_x is not a leaf, then every $P_1 : u \rightsquigarrow x$ and $P_2 : x \rightsquigarrow v$ that decompose P are
$Ushape$-shaped paths, and for their intermediate nodes y it holds $\mathsf{Lv}(y) \geq \mathsf{Lv}(x')$, where $B_{x'}$ is a child of B_x. Then the induction hypothesis applies, and $d'(u, x) \leq \mathsf{wt}(P_1)$, $d'(x, v) \leq \mathsf{wt}(P_2)$, and hence after B_x is processed, $d'(u, v) \leq \mathsf{wt}(P)$.

After the second pass processes a bag B_x that introduces some node x, it holds that $(x, d(x, v)) \in \textsc{Fwd}'(x)$ and $(v, d(v, x)) \in \textsc{Bwd}'(x)$ for all $v \in B_x$. The statement holds trivially for the root, since $|B_0| = 1$. We now proceed inductively to some bag B_x examined by the algorithm in the second pass. We only consider $\textsc{Fwd}'(x)$ (the argument is similar for $\textsc{Bwd}'(x)$). The claim is immediate if $d(x, v) = \mathsf{wt}(x, v)$, and follows from the first pass if $d(x, v) = \mathsf{wt}(P)$ for a U-shaped path $P : x \rightsquigarrow v$. The only case left is that $d(x, v) = \mathsf{wt}(P)$ for a path $P : x \rightsquigarrow u \rightsquigarrow v$, where

$u \in B_x$ is the first node in P with $\mathsf{Lv}(u) < \mathsf{Lv}(x)$. Then $P' : x \rightsquigarrow u$ is necessarily a U-shaped path such that for every intermediate node y we have $\mathsf{Lv}(y) \geq \mathsf{Lv}(x)$. It follows from the first pass on B_x that $d'(x,u) \leq \mathsf{wt}(P')$. Additionally, both $\mathsf{Lv}(u), \mathsf{Lv}(v) < \mathsf{Lv}(x)$, hence by the induction hypothesis $d'(u,v) = d(u,v)$, and thus $d'(x,v) = d'(x,u) + d'(u,v) = \mathsf{wt}(P') + d(u,v) \leq d(x,v)$.

Figure 4 depicts the two passes. It follows that 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. □

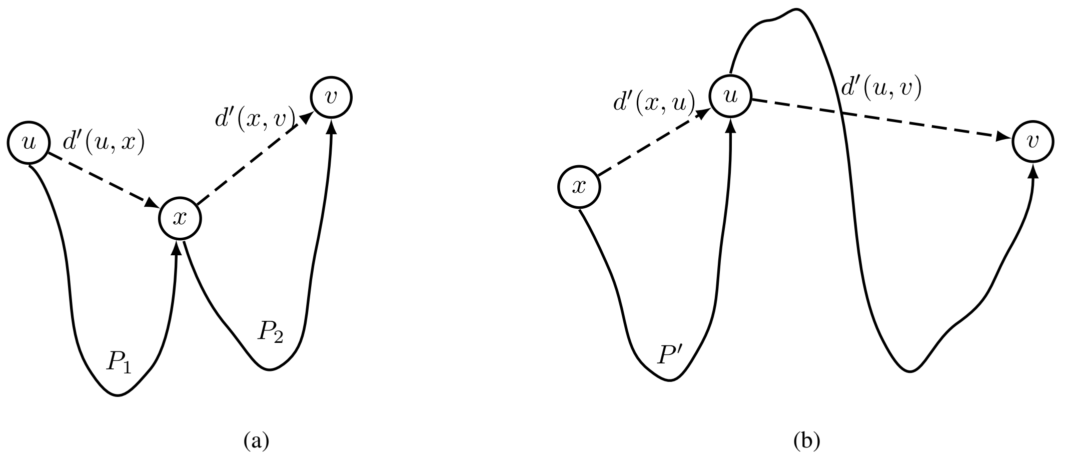

Figure 4: (a) In the first pass, when LOCDIS examines bag B_x, it is $d'(u,x) \leq \mathsf{wt}(P_1)$ and $d'(x,v) \leq \mathsf{wt}(P_2)$ for all U-shaped paths $P_1 : u \rightsquigarrow x$ and $P_2 : x \rightsquigarrow v$. After B_x is processed, $d'(u,v) \leq \mathsf{wt}(P_1) + \mathsf{wt}(P_2)$. (b) In the second pass, when B_x is examined, every path $P : x \rightsquigarrow v$ can be decomposed to $x \rightsquigarrow u \rightsquigarrow v$ where u is the first node in P introduced above B_x. Then $P' : x \rightsquigarrow u$ is U-shaped and $d'(x,u) \leq \mathsf{wt}(P')$ from the first pass, whereas $d'(u,v) = d(u,v)$ by the induction hypothesis. After B_x has been processed, $d'(x,v) = d'(x,u) + d'(u,v) \leq d(x,u)$.

Lemma 5. *Algorithm* LOCDIS *requires $O(n \cdot t^2)$ time and $O(n \cdot t)$ space.*

Proof. The algorithm LOCDIS examines each bag B_x once in each pass. It is straightforward that the time spent in each B_x is $O(t^2)$ for computation of the z-values, and $O(t^2)$ for updating the sets FWD' and BWD'. Hence, the total time of the local distance computation is $O(n \cdot t^2)$. The space bound follows from Lemma 3. □

Remark 3. The algorithm LOCDIS reports any negative cycle C, by discovering that $d'(u,u) < 0$ for some u in C. In the following sections we consider that there are no negative cycles in G (also see Remark 6).

Theorem 2. *Given a (weighted) graph $G = (V,E)$ and a semi-nice tree-decomposition $\mathrm{Tree}(G)$ of G of width t, the algorithm* LOCDIS *either reports a negative cycle, or correctly computes the local distance sets, and runs in $O(n \cdot t^2)$ time and $O(n \cdot t)$ space.*

4.2 Improved preprocessing time for single-source shortest path

Using the algorithm LOCDIS for local distance computation, we present an algorithm for single-source shortest path queries with a preprocessing time of $O(n \cdot t^2)$ and query time and space usage each being $O(n \cdot t)$. A similar result was shown by [10, Theorem 3.1], except that our algorithm for preprocessing time is a factor of t^2 faster and for query time is a factor of t^3 faster.

We first state two lemmas that will allow us to argue for correctness of the algorithms presented in the current and later sections.

Lemma 6. *Consider a weighted graph $G = (V, E)$ and a tree-decomposition* $\mathrm{Tree}(G)$*. Let $u, v \in V$, and $P' : B_1, B_2, \ldots, B_j$ be the unique path in T such that $u \in B_1$ and $v \in B_j$. 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. 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 $\mathrm{Tree}'(G)$, which is similar to $\mathrm{Tree}(G)$ 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 $\mathrm{Tree}'(G)$ forms a tree-decomposition of G, from the definition. By Remark 1, each bag B' in the unique path $P'' : B_1, \ldots, B_i, B, B_{i+1}, \ldots, B_j$ in $\mathrm{Tree}'(G)$ separates u from v in G. Hence, each path $u \rightsquigarrow v$ must go through some node in B. □

Lemma 7. *Consider a weighted graph $G = (V, E)$ and a tree-decomposition* $\mathrm{Tree}(G)$*. Let $u, v \in V$, and $P' : B_1, B_2, \ldots, B_j$ be the unique path in T such that $u \in B_1$ and $v \in B_j$. Let $A = \{u\} \times B_2 \times \cdots \times B_{j-1} \times \{v\}$. Then $d(u, v) = \min_{(x_1, \ldots, x_{j+1}) \in A} \sum_{i=1}^{j} z_i$ where z_i is such that $(x_i, x_{i+1}, z_i) \in \mathsf{LD}(B_i)$.*

Proof. Consider a witness path $P : u \rightsquigarrow v$ such that $\mathsf{wt}(P) = d(u, v)$. Using Lemma 6, we know that there exists some node $x_i \in (B_i \cap B_{i+1} \cap P)$, for each $i \in \{1, \ldots, j\}$. Then, $d(u, v) = \sum_{i=1}^{j} z_i$, where $z_i = d(x_i, x_{i+1})$ and since both x_i and x_{i+1} are contained in B_i, we have that $(x_i, x_{i+1}, z_i) \in \mathsf{LD}(B_i)$. □

In words, Lemma 7 states that for $u \in B_1$, $v \in B_j$, the distance $d(u, v)$ can be written as the minimum sum of distances $d(x_i, x_{i+1})$ between pairs of nodes (x_i, x_{i+1}) that appear in bags B_i that constitute the unique $B_1 \rightsquigarrow B_j$ path in T (see Figure 5 for an illustration).

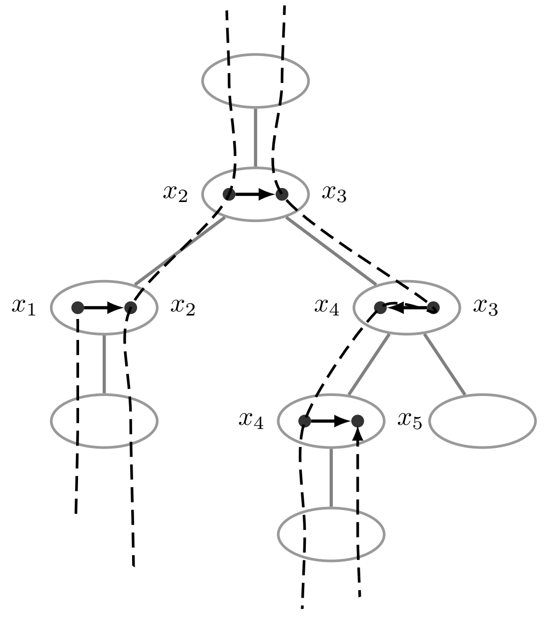

Figure 5: The weight of the minimum weight path $u = x_1 \rightsquigarrow x_5 = v$ (dashed line) can be written as the sum of local distances between nodes that appear in the path that connects their bags.

Our algorithm, namely, ImpSiSoShPa (for improved single-source shortest path) is as follows.

Preprocessing. Apply the local distance algorithm to compute the sets $\mathrm{Fwd}^*(u)$ and $\mathrm{Bwd}^*(u)$ for all $u \in V$.

Query. The query from a node u consists of accessing the bags of $\mathrm{Tree}(G)$ via DFS, starting in the bag B_u. The algorithm maintains a function $d'(u, v)$ for all $v \in V$, initialized with $d(u, v)$ for all $v \in B_x$, and ∞ for all other v.

Upon examining a bag B_v for some $v \in V$ for which $d'(u,v) = \infty$), it updates $d'(u,v)$ with $z = \min_{x \in B_v}(d'(u,x) + d(x,v))$. Finally, it returns $d'(u,v)$ for all $v \in V$.

Theorem 3. *Let a weighted graph $G = (V,E)$ and a semi-nice tree-decomposition $\mathrm{Tree}(G)$ of G of width t that consists of $O(n)$ bags be given. The algorithm* IMPSISOSHPA *correctly answers single-source shortest path queries on G and requires*

- *$O(n \cdot t^2)$ preprocessing time;*
- *$O(n \cdot t)$ space; and*
- *$O(n \cdot t)$ query time.*

Proof. The correctness follows easily from Lemmas 4 and 7 and an induction on the DFS. Lemma 5 guarantees the preprocessing time and space bounds. The query time follows by an $O(n)$ time bound in traversing $\mathrm{Tree}(G)$, and $O(t)$ time spent in each bag B_v. □

5 Improved Query Time for Low Tree-width

We now present improved algorithms for the problem in Corollary 1. The improvement consists of removing the log-factor from the pair query time, while keeping the remaining complexities the same. The resulting algorithms are called REACHTREE and SHPATREE instead of REACH and SHPAT respectively.

Modification to preprocessing, intuitive description. The idea is that every $u \rightsquigarrow v$ path goes through the LCA L of B_u and B_v. Hence in the query phase, we only want to make queries in L. To do so, we modify the bags of the binary-separator-hierarchy so that each bag B forms a separator for the whole graph (and not just for the sub-graph $T(B)$). Previously, for a path $P : u \rightsquigarrow v$, we only stored the information in the bag highest up in the separator-hierarchy which contained a node from P. Now we want each node in P to have that information (if they belong to a bag higher in the hierarchy than the bags B_u and B_v). Note that this is already the case for the root. To ensure it for each other bag, we can proceed inductively downward in the tree. The idea is that to get the information for a fixed bag B, we consider the set of nodes adjacent $T(B)$. All such nodes are in a separator higher in the hierarchy and are thus already updated by induction, and for each such node z we can then test them for paths from u to/from z to/from v, where $u \in T(B)$ and $v \in B$.

Modification to preprocessing, formal description. The changes compared to the algorithms REACH and SHPAT of Corollary 1 are as follows for the preprocessing:

1. Instead of creating a $(t+1)$-binary-separator-hierarchy (as done in Corollary 1), we create a $4 \cdot (t+1)$-binary-separator-hierarchy: the difference is that instead of stopping at size $t+1$ for leaves, we stop at size $4 \cdot (t+1)$. This results in the size of the leaves in the hierarchy being upper bounded by $4 \cdot (t+1)$ instead of $t+1$, while the size of internal bags remains unchanged. For a bag B of the binary-separator-hierarchy, let $U(B) = N(T(B)) \setminus T(B)$ (where for a set $V' \subseteq V$ the set $N(V')$ is the neighbors of V' i.e. $\bigcup_{v \in V'} \{u : (u,v) \in E \text{ or } (v,u) \in E\}$). Traverse the separator-hierarchy by DFS, and for each bag B make the following modifications.
   - **Modification to preprocessing specific to the algorithm** REACHTREE. First, preprocess G using REACH. For each node $u \in U(B)$, node $v \in B$, and $w \in T(B)$, add w to F'_v (i.e. set bit number $i - a$ of F'_v to 1, where i is the index of w and a is such that the nodes in $T(B)$ form the interval $[a; b]$ for some b), if there is a path from v to u to w (obtained by checking if $v \in T'_u$ and $w \in F'_u$) and, similarly, add w to T'_v if there is a path from w to u to v.
   - **Modification to preprocessing specific to the algorithm** SHPATREE. First, preprocess G using SHPAT, using the algorithm IMPSISOSHPA defined in Theorem 3 instead of Dijkstra's algorithm for single-source shortest path computation to determine the functions f and t. For each node $v \in B$ and $w \in T(B)$, let $\bar{\mathsf{f}}(v,w) = \min_{u \in U(B)}(\mathsf{f}(v,w), \bar{\mathsf{t}}(u,v) + \bar{\mathsf{f}}(u,w))$ and $\bar{\mathsf{t}}(v,w) = \min_{u \in U(B)}(\mathsf{t}(v,w), \bar{\mathsf{f}}(u,v) + \bar{\mathsf{t}}(u,w))$.

Note that since the nodes in $U(B)$ belong to separators higher in S, the values $\bar{\mathsf{t}}(u, x)$ and $\bar{\mathsf{f}}(u, x)$ have been computed before B is examined. At the end of the DFS, replace t by $\bar{\mathsf{t}}$ and f by $\bar{\mathsf{f}}$.

2. Observe that each non-leaf bag B in the binary-separator-hierarchy S corresponds to some bag B' in the original tree-decomposition, except that B might contain a subset of the nodes of B' (i.e. B consists of the subset of B' which is not contained in some ancestor of B). Our modification then introduces to B all nodes from B'. Note that the preceding (e.g. the definition of $U(B)$) are defined in terms of B before this modification.

   - **Modification to preprocessing specific to the algorithm** REACHTREE**.** After introducing the additional nodes of B' to B with $i = \mathsf{Lv}(B)$, construct the sets F^i_u and T^i_u from $\{T'_v \mid v \in B\}$ and $\{F'_v \mid v \in B\}$ respectively, as in the algorithm REACH. More precisely, do as follows: Pick an arbitrary ordering σ on the nodes of B, such that $\sigma(v)$ is the number the node $v \in B$ has been assigned by the ordering. Then, for all $w \in T(B)$ and $v \in B$, set the $\sigma(v)$-th bit of F^i_w to true iff $w \in T'_v$ and similarly, set the $\sigma(v)$-th bit of T^i_w to true iff $w \in F'_v$.

Modified reachability query. A reachability query (u, v) is then handled as follows: find the LCA L of B_u and B_v, and let a be the number of nodes contained in the ancestors bags of L and b the number of nodes in L (we store these numbers in bag L). Compute the boolean-AND of the bits between bit $a+1$ and $a+b$ of F_u and T_v and return whether the answer is not all 0.

Modified shortest path query. A shortest path query (u, v) is then handled as follows: find the LCA L of B_u and B_v, and return $\min_{v' \in L} (\mathsf{t}(v', u) + \mathsf{f}(v', v))$.

Example 2. We use the same example as in Example 1, to make the difference between algorithm REACH and algorithm REACHTREE clear. Hence, we again consider the graph G given in Figure 1. The names of the nodes in G are (still) the indices they get assigned by the DFS. The algorithm preprocesses G and gets the binary-separator-hierarchy in Figure 6 and for each node stores the corresponding bit-arrays shown in Figure 7. The two figures are each different from the ones in Example 1.

To answer the pair reachability query of whether node 6 can reach node 5, the algorithm proceeds as follows: First find the LCA of the bags that contain node 6 and node 5 in the binary-separator-hierarchy. That is the bag (6 7 9). Observe that there are 3 nodes in the ancestors (i.e. 8, 9, 10) and 3 nodes in (6 7 9). This is written down in preprocessing. Then boolean-AND the bits from 4 (i.e. the 3 in the ancestor plus 1) to $3 + 3 = 6$ (i.e., the sum of the nodes in the ancestors and in (6 7 9)) of $F_6 = 111111$ and $T_5 = 111111$. We see that we get 111. Since this is not all 0, the pair query $(6, 5)$ returns true. Note the difference to the query in Example 1: we boolean-OR 3 bits instead of 6. This could be more pronounced in a larger example.

To answer the single-source reachability query for node 6, we consider $F_6 = 111111$. For each 1, we take the corresponding F'_v list and boolean-OR them together at the right place. In other words, the 6 ones correspond to nodes 8, 9, 10, 6, 7 (also 9 again) in that order (we start in the root of the binary-separator-hierarchy and proceed downward towards the bag that contains node 6) and $F'_8 = F'_9 = F'_{10} = 1111111111$ and $F'_6 = F'_7 = 1111$. We then boolean-OR F'_8, F'_9, F'_{10} together (over all bits) and then, in the result of 1111111111, we boolean-OR the bits corresponding to indices in $[4; 7]$ together with F'_6 and F'_7: this is because the subtree rooted at the bag (6 7), that contains node 6, consists of 4,5,6,7, and note that because of our way to assign indices the nodes of a subtree always form some interval of indices. Thus we get the result is 1111111111, which then means that node 6 can reach all nodes. Note that this is in essence the same as in Example 1.

Lemma 8. *The algorithms* REACHTREE *and* SHPATREE *correctly answer reachability and shortest path queries respectively.*

Proof. Consider a query from u to v and let L be the LCA of B_u and B_v, corresponding to some L' of the tree-decomposition. It follows from Remark 1 that all paths $u \rightsquigarrow v$ must go through some node $w \in L'$, and because of the modification, we have $w \in L$. We argue that the information $u \rightsquigarrow w$ and $w \rightsquigarrow v$ is captured in L. We only explicitly consider REACHTREE as the argument for SHPATREE is similar. In the unmodified algorithm REACH, we only have information about a path in the node of the highest level which contains a node on the path. We argue that for all nodes $w \in B$ for some B being L or an ancestor of L in the separator-hierarchy, if there is a path of the form $u \rightsquigarrow w \rightsquigarrow v$

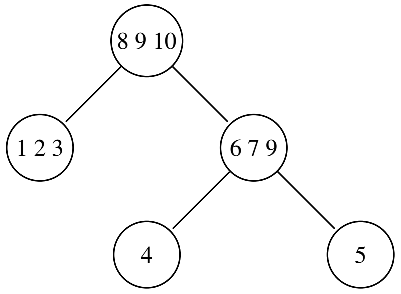

Figure 6: A binary-separator-hierarchy obtained from G, by the algorithm ReachTree.

v	F'_v
1	111
2	111
3	111
4	1
5	1
6	1111
7	1111
8	1111111111
9	1111111111
10	1111111111

v	T'_v
1	111
2	111
3	111
4	1
5	1
6	0111
7	0111
8	1110111111
9	1110111111
10	1110111111

v	F_v
1	111111
2	111111
3	111111
4	0000001
5	1111111
6	111111
7	111111
8	111
9	111
10	111

v	T_v
1	111111
2	111111
3	111111
4	1111111
5	1111111
6	111111
7	111111
8	111
9	111
10	111

Figure 7: Bit-arrays for the nodes of G, by the algorithm ReachTree.

in G, then $u \in T'_w$ and $v \in F'_w$. Note that we only need to argue this for the case where $w \in L$, but we will show the more general statement to allow us to use induction in the level i of B.

Base case, B is the root (at level 0): The statement is true for the root already before the modification, since clearly no path can go through a bag above the root.

Induction case, B is at level $i > 0$: Consider the subtree $T(B)$, and let B' be the corresponding bag before the modification. Observe first that by induction, we already have the information in every node in $(B \setminus B')$ (because such nodes also appear in a bag higher in the separator-hierarchy) and thus only need to consider a node w introduced in B. Any path from $u \in T(B)$ to w must either be contained inside $T(B)$ or go from u to some node u' adjacent to $T(B')$ and then from u' to w. The former type of path was already handled at this level earlier (and thus still is) and the latter must go through some node in $U(B')$. Since each node in $U(B')$ is contained in $T(B'')$, where B'' is an ancestor of B in the separator hierarchy (and hence has all information about paths to/from them and from/to nodes in $T(B'')$, by induction), we query each node $u' \in U(B')$ for paths of the form from u to u' to w (by checking if $u \in T'_u$ and $w \in F'_u$) and from w to u' to u. Afterwards, we have considered all paths going through some node in B, and can then continue to the children. The desired result follows. □

Lemma 9. *Let a (weighted) graph $G = (V, E)$ and a semi-nice tree-decomposition* $\mathrm{Tree}(G)$ *of G of width t that consists of $O(n)$ bags be given. Let $\widehat{t} = t \cdot \log(n/t)$. The algorithm* ReachTree *(resp.* ShPaTree*) requires*

- *$O(n \cdot \widehat{t} \cdot t)$ preprocessing time;*
- *$O(n \cdot \lceil \widehat{t}/W \rceil)$ (resp. $O(n \cdot \widehat{t})$) space;*
- *$O(\lceil t/W \rceil)$ (resp. $O(t)$) pair shortest path query time; and*
- *$O(\lceil n \cdot t/W \rceil)$ (resp. $O(n \cdot t)$) single-source shortest path query time.*

Proof. The height of the $4 \cdot (t+1)$-binary-separator-hierarchy S is still $O(\log(n/t))$. Denote with B' the bag B before adding the new nodes in Step 2. Because B has size at most $t+1$, Step 2 of the preprocessing adds at most $t+1$ new nodes to each such B' that is internal in S. Since previously the size of $T(B')$ was at most $\frac{2}{3}$ of the size of its parent B'' (by Proposition 1 for $\Delta = \frac{2}{3}$), the size of $T(B)$ is still some fraction of the size its parent $T(B'')$ in the modified hierarchy (i.e., at most $\frac{11}{12}$ of the size, in case $|T(B'')| = 4 \cdot (t+1)$ nodes and $t+1$ nodes were added in B'). Also, since we do not change the leaves of the binary-separator-hierarchy, they are still disjoint and have size $O(t)$ each. Therefore, we have $O(n/t)$ such leaves and thus $O(n/t)$ internal bags (because there is one less non-leaf bag as compared to leaf bags). Each bag contains $O(t)$ nodes and thus we have at most $O(n)$ nodes over all, even counting multiplicities (i.e., nodes that were added in multiple bags).

Single-source-queries. These queries have not changed and thus have the same running time.

Pair query time. It is clear that the query times are $O(\lceil t/W \rceil)$ and $O(t)$ respectively.

Space usage. Each node has one pointer to a bag of S and a pointer from the map. Also, each bag requires $O(t)$ space for the numbers it stores. Furthermore, each node u has the two bit-arrays, T_u and F_u of length at most $O(t \cdot \log(n/t))$, i.e. the height of the separator hierarchy times the $O(t)$ nodes in each bag. Finally, the size of arrays F'_u and T'_u has not changed as compared to before the modification (the content has in general, though). Hence we see that REACHTREE requires $O(n \cdot t \cdot \lceil \log(n/t)/W \rceil + n) = O(n \cdot t \cdot \lceil \log(n/t)/W \rceil)$ space, as REACH.

Preprocessing time. We only examine the additional time spent by REACHTREE, given the computation performed by REACH. For a fixed level i, we can find $U(B)$ for each B in time $O(m)$, since each edge is only in one subtree of the separator hierarchy. Hence, we use $O(m \cdot \log(n/t)) = O(n \cdot t \cdot \log(n/t))$ time in total to do so. For a fixed level i, we argue that we make only $O(n \cdot t^2)$ queries *to* level i. Fix a bag B with $\mathsf{Lv}(B) = i$. It is clear that we only make queries to B from the trees rooted at the descendants of B. Thus, we will argue that we make at most $O(|T(B)| \cdot t^2)$ and we are done. More precisely we argue that we make $O(|T(B)| \cdot t^2 \cdot (11/12)^{j-1})$ queries from level $i+j$, for $j \geq 1$ and then, summing over all the levels, we get a total of $O(|T(B)| \cdot t^2)$ queries. It is easy to see that we make at most $O(|T(B)| \cdot t^2)$ queries from level $i+1$ (i.e. each of the $|T(B)|$ nodes is in one child C, and for each node $u \in T(C)$ we make a query with each node in C and each node in B). Two levels down we do not make queries from $\frac{1}{12}$ of the nodes, which can be seen as follows: Let B' be the separator bag of one of the children of B. Consider the (unique since it is a tree) path between B and B' in $\mathrm{Tree}(G)$. Only the child B'' of B' that has the bags containing that path will make queries from $T(B')$ to B from level $i+2$. But, by construction $|B''| \leq \frac{11 \cdot |B'|}{12}$. In general, at level $i+j$ at most one of the descendants of B' (the one having the last part of the path from B' to B) at that level makes queries to B. Since each subtree at level $i+j$ contains at most $\frac{11}{12}$ of the nodes of its parent we see that we use $O(|T(B)| \cdot t^2 \cdot (11/12)^{j-1})$ queries from level $i+j$, for each $j \geq 1$. This completes the proof. □

We thus conclude with the following theorem.

Theorem 4. *Let a (weighted) graph $G = (V, E)$ with n nodes and m edges and a semi-nice tree-decomposition $\mathrm{Tree}(G)$ of G of width t that consists of $O(n)$ bags be given. Let W be the wordsize of the RAM and let $\widehat{t} = t \cdot \log(n/t)$. The algorithm REACHTREE (resp. SHPATREE) correctly answers pair and single-source reachability (resp. shortest path) queries on G and requires*

- *$O(n \cdot \widehat{t} \cdot t)$ preprocessing time;*
- *$O(n \cdot \lceil \widehat{t}/W \rceil)$ (resp. $O(n \cdot \widehat{t})$) space;*
- *$O(\lceil t/W \rceil)$ (resp. $O(t)$) pair reachability (resp. shortest path) query time; and*
- *$O(\lceil n \cdot t/W \rceil)$ (resp. $O(n \cdot t)$) single-source reachability (resp. shortest path) query time.*

6 Improved Preprocessing for Low Tree-width

In this section we show a different approach for answering shortest path queries. This approach reduces the $\log n$ factor in the preprocessing time and space usage of Theorem 4 down to $(\lambda+2)\cdot\log^{(\lambda)*} n$ for any given $\lambda \in \mathbb{N}$, but incurs an increase in the query time by a factor of $(\lambda+1)\cdot t$. The described algorithm is called ImPreShPa (improved preprocessing for shortest path).

Recall that Lemma 7 shows that for nodes that appear in bags B, B' of the tree-decomposition $T = \mathrm{Tree}(G)$, the distance can be written as a sum of distances $d(x_i, x_{i+1})$ between pairs of nodes (x_i, x_{i+1}) that appear in bags B_i that constitute the unique $B \rightsquigarrow B'$ path in T. The main part of the preprocessing consists of manipulating *summary trees*. Intuitively, given a tree-decomposition T, a summary tree $\overline{T}$ of T consists of a subset of bags of T, such that:

1. For bags B and B' in $\overline{T}$, B is the parent of B' in $\overline{T}$ iff B is the lowest ancestor of B' in T that appears in $\overline{T}$.
2. For bags B with children B' in $\overline{T}$, for all $u \in B$ and $v \in B'$, the distances $d(u,v)$ and $d(v,u)$ are stored in B'.

Hence, in such a summary tree $\overline{T}$, the distances between all nodes in B and B' have been summarized and can be retrieved fast, regardless of the length of the $B \rightsquigarrow B'$ path in T (which we would otherwise have to pay as a cost for retrieving them). The preprocessing then applies recursive summarizations of T, so that in the end, for any two nodes $u, v \in V$, the distance $d(u,v)$ can be written as a sum of $O(\lambda+1)$ summarized distances, which can be retrieved by looking up $O((\lambda+1)\cdot t^2)$ such summaries distances. The algorithmic technique is an adaptation of [2]. A direct application results in higher preprocessing time, space, and query time complexities by at least a factor of t^2 (as in [10]).

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

1. $\overline{T} = (V_{\overline{T}}, E_{\overline{T}})$ is a tree with $V_{\overline{T}} \subseteq V_T$, and each bag of $\overline{T}$ is a child of its lowest ancestor in T that appears in $\overline{T}$.
2. $S = (S_F, S_B)$ are the summary sets defined as follows. For each node u introduced in B_u in $\overline{T}$, and B_i being the parent of B_u in $\overline{T}$, let $S_F(u) = \{(v, d(u,v)) :\ v \in B_i\}$ and $S_B(u) = \{(v, d(v,u)) :\ v \in B_i\}$.

For all u that appear in some bag in $\overline{T}$, we have both $|S_F(u)|, |S_B(u)| \leq t+1$. Each $u \in V$ either does not appear in $\overline{T}$, or appears in a connected component of $\overline{T}$ (i.e., for all $u \in V$, if u appears in bags B and B' in $\overline{T}$ then it must appear in every bag in the unique path from B to B' in $\overline{T}$). We say that u is introduced in bag B_u of $\overline{T}$ if B_u is the bag of smallest level among all bags on $\overline{T}$ that contain u.

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

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

Tree $f(k)$-partitioning. Given a function $f(k)$ and a binary summary tree $(\overline{T}, S)$ of some tree-decomposition T with $|V_{\overline{T}}| = k$ bags, the $f(k)$-partitioning of $\overline{T}$ consists of partitioning $\overline{T}$ into $O\left(\frac{k}{f(k)}\right)$ connected components that contain $f(k)$ bags each. The partitioning of a tree is achieved as follows: use a DFS to keep track of the number of nodes in each subtree $\widetilde{T}$, and whenever the number of nodes in $\widetilde{T}$ becomes at least $\frac{f(k)}{2}$ cut $\widetilde{T}$ off into its own component. Note

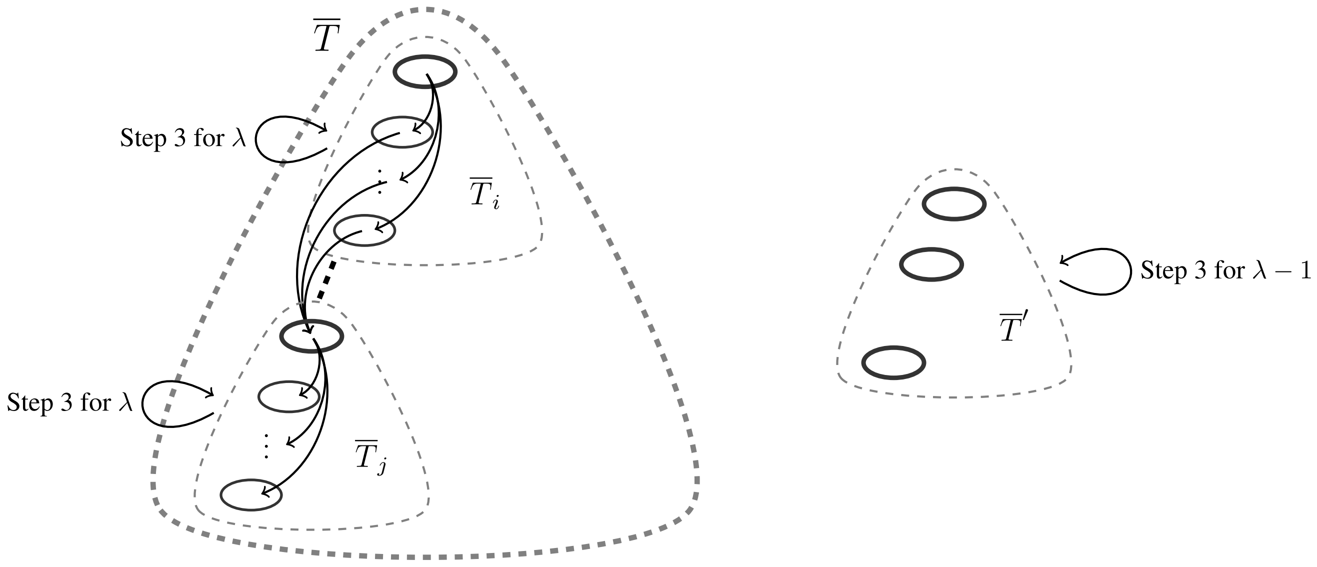

Figure 8: Scheme of the Step 3 recursion of algorithm ImPreShPa for a given λ. A summary tree $\overline{T}$ of size k is partitioned into components of size $f(k)$ each, and the root and leaf distance set computation is executed. The root bags are shown in boldface. The recursion is then two-fold: (1) the roots of the components form a new summary tree $\overline{T}'$ which is recursively processed for $\lambda - 1$, and (2) each component is itself a summary tree, recursively processed for λ.

that since $(\overline{T}, S)$ is binary, no component becomes larger than $f(k)$. This partitioning takes linear time in the size of $\overline{T}$. Each component $\overline{T}_i$ of the partitioning yields a summary tree of T.

Root and leaf distance set computation. Given a summary tree $(\overline{T}, S)$ and f, perform an $f(k)$- partitioning on $\overline{T}$. Examine each component $\overline{T}_i$ with root B_0^i. For all $u \in V$ introduced in some bag B_u^i of $\overline{T}_i$ other than B_0^i, we compute the root and leaf distance sets of u defined as $R(u) = \left\{(v, d(v,u)) :\ v \in B_0^i\right\}$ and $L_j(u) = \left\{(v, d(u,v)) :\ v \in B_0^j\right\}$ where B_0^j is the root of a child component $\overline{T}_j$ of $\overline{T}_i$, such that B_u^i is an ancestor of B_0^j.

- *(Root distance set computation).* For each $v \in B_0^i$, to determine the root distance $d(v,u)$ for all u, execute the following steps. First, construct $\mathsf{LD}\left(B_0^i\right)$ and the set $A = \left\{(y,z) :\ (v,y,z) \in \mathsf{LD}\left(B_0^i\right)\right\}$. Then, execute a BFS in $\overline{T}_i$, and upon examining a bag B_j^i, (i) remove from A all nodes that do not appear in B_j^i, and (ii) for each node u introduced in B_j^i, insert in A the pair (u, z'), where $z' = \min_{(y,z)\in A}(z + d(y,u))$. Note that $(y, d(y,u)) \in \textsc{Bwd}^*(u)$. Finally, insert (v, z') in $R(u)$. The correctness of the construction follows from Lemma 7 and a simple induction on the BFS.
- *(Leaf distance set computation).* The leaf distance computation is similar to the root distance computation: compute the leaf distance sets $L_j(u)$ for every root B_0^j of a child component $\overline{T}_j$ of $\overline{T}_i$, by constructing the set $A_j = \left\{(y,z) :\ (y,v,z) \in \mathsf{LD}\left(B_0^j\right)\right\}$, and traversing the path up to the root of $\overline{T}_i$.

The time and space requirement of the root and leaf distance set computation is as follows: Each root of each component is the source of at most $2 \cdot (t+1)$ traversals, leading to $O\left(t \cdot \frac{k}{f(k)}\right)$ traversals of length $O(f(k))$ each. If each bag of $\overline{T}$ introduces at most n' nodes, then the cost of each step of each traversal is $O(t \cdot n')$ for updating the set A and the distance sets of the n' nodes introduced in the current bag. Hence the total time spent is $O(k \cdot t^2 \cdot n')$. The space required is dominated by the space used for storing the root and leaf distance sets, which is bounded by $O(k \cdot t \cdot n')$, since there exist $O\left(\frac{k}{f(k)} \cdot t\right)$ nodes v introduced in the root of some component $\overline{T}_i$, and each one appears in $O(f(k) \cdot n')$ distance sets.

Tree summarization. Consider a summary tree $(\overline{T}, S)$ of size k that has been partitioned into $\frac{k}{f(k)}$ components, for some f, and the root and leaf distance set computation has been carried out. The summarization of $\overline{T}$ is done by constructing a new summary tree $(\overline{T}', S')$ as follows. The bag set $V_{\overline{T}'}$ contains all the bags B_0^i that appeared as the root of some component $\overline{T}_i$ of the partitioning. A bag B_0^i is a parent of B_0^j in $\overline{T}'$ iff $\overline{T}_j$ is a child component of $\overline{T}_i$ in $\overline{T}$. For every node u introduced in a bag B_0^j in $\overline{T}'$ with parent B_0^i we construct

$$S'_B(u) = \left\{ (v, z) : \ v \in B_0^i \text{ and } z = \min_{(y,z') \in S_B(u)} z' + d(v, y) \right\}$$

if the bag B_0^j introduces u in both $\overline{T}$ and $\overline{T}'$, otherwise $S'_B(u) = R(u)$. Note that in the former case, $(v, d(v, y)) \in R(y)$, hence this value can be retrieved from $R(y)$. The computation of $S'_F(u)$ is done in a similar way, but requires constructing the sets $R'(u) = \left\{ (v, d(u, v)) : \ v \in B_0^i \right\}$ during the root distance set computation. This construction is very similar to the one of $R(u)$ and has the same complexity, and its description is omitted. The computation of the summary sets requires $O(t^2)$ time for each node u introduced in a bag that appears in $\overline{T}$ and $\overline{T}'$, and $O(t)$ time for all other nodes in $\overline{T}'$. If there are at most n' such nodes u, the time spent in this step is $O\left(\frac{k}{f(k)} \cdot t^2 \cdot (n' + 1)\right)$. The space required is $O\left(\frac{k}{f(k)} \cdot t \cdot (n' + 1)\right)$ for storing the newly computed summary sets.

Preprocessing $T = \mathrm{Tree}(G)$. Now we describe the preprocessing of T in order to answer shortest path queries of the form (u, v), where B_u is an ancestor of B_v. The preprocessing for the case where B_v is an ancestor of B_u is similar. We later show how to answer general pair queries. The preprocessing is parametric on an arbitrarily chosen $\lambda \in \mathbb{N}$, which results in $O((\lambda + 2) \cdot n \cdot t^2 \log^{(\lambda)*} n)$ preprocessing time and $O\left((\lambda + 1) \cdot t^2\right)$ query time (see Figure 8). The steps of IMPRESHPA are as follows.

Step 1 First, use LOCDIS to compute the local distance sets in T, and construct the summary tree $(\overline{T}, S)$, of T with $V_{\overline{T}} = V_T$. The computation of summary edges S is done by traversing T via DFS, and for each bag B_u with parent B, constructing $\mathsf{LD}(B)$. Lemma 7 implies that for each $v \in B$, we have $d(u, v) = \min_{y \in B_u \cap B} (d(u, y) + d(y, v))$ and $d(v, u) = \min_{y \in B_u \cap B} (d(v, y) + d(y, u))$. These distances exist in $\mathsf{LD}(B)$, $\mathrm{FWD}^*(u)$ and $\mathrm{BWD}^*(u)$, and are used to construct $S_F(u)$ and $S_B(u)$.

Step 2 Apply the root and leaf distance set computation on $\overline{T}$, recursively for $\log^{(\lambda+1)*} n$ levels, and $f(k) = t \cdot \log^{(\lambda)*} k$. That is, each time consider a summary tree of k bags, partition it into $O\left(\frac{k}{t \cdot \log^{(\lambda)*} k}\right)$ components of size $f(k)$ each, and compute the root and leaf distance sets. The next level processes summary trees $\overline{T}_i$ corresponding to components in the current level. Initially we have $k = O(n)$. For every partitioned summary tree $\overline{T}_i$ constructed in this recursion, perform a tree summarization, and let $\overline{T}_i'$ be the resulting summary tree. Execute Step 3 on $\overline{T}_i'$ for $\lambda - 1$.

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

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

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

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

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

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

Lemma 10. *Given a semi-nice tree-decomposition T of G and some $\lambda \in \mathbb{N}$, the preprocessing requires $O\left((\lambda+2)\cdot t^2 \cdot n \cdot \log^{(\lambda)*} n\right)$ time and $O\left((\lambda+2)\cdot t \cdot n \log^{(\lambda)*} n\right)$ space.*

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

Step 1 LocDis requires $O(n\cdot t^2)$ time and $O(n \cdot t)$ space (Lemma 5). The construction of the summary sets $S_F(u)$, $S_B(u)$ happens at most once for each node u, requiring $O(t^2)$ time for building the local distance set $\mathsf{LD}\,(B)$ of the parent bag B of B_u, and $O(t^2)$ time for calculating $d(u,v)$ for all $v \in B$. Hence this step requires $O(n\cdot t^2)$ time. The space required is $O(n\cdot t)$ for storing the computed summary tree.

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

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

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

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

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

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

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

Step 4 Note that every summary tree $\overline{T}$ in the last level of the recursion of Step 2 is a semi-nice tree-decomposition of size at most

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

since in discrete context, for all $x\geq 0$, we have $\log^{(\lambda)*} x \leq \frac{2\cdot x}{3}$, and hence $\sum_i \log^{((\lambda)*)^i} t \leq 3\cdot \log^{(\lambda)*} t$.

By Remark 2, the total number of nodes that appear in $\overline{T}$ is $O\left(t \cdot \log^{(\lambda)*} t\right)$. It follows by the way edges are stored in T that the number of edges in $\overline{T}$ is $O\left(t^2 \cdot \log^{(\lambda)*} t\right)$. We conclude that the all pairs reachability computation in $\overline{T}$ requires $O\left(t^3 \cdot \left(\log^{(\lambda)*} t\right)^2\right)$ time, and there are $O\left(\frac{n}{t \cdot \log^{(\lambda)*} t}\right)$ such summary trees $\overline{T}$, resulting in $O\left(t^2 \cdot n \cdot \log^{(\lambda)*} t\right)$ total time. The space required is $O\left(t \cdot n \cdot \log^{(\lambda)*} t\right)$ for storing $\frac{n}{t \cdot \log^{(\lambda)*} t}$ lookup matrices of size $O\left(\left(t \cdot \log^{(\lambda)*} t\right)^2\right)$ each.

Step 5 We can preprocess each recursion tree in time and space proportional to its size [25] so this step adds no overhead to the complexity.

The desired result follows. □

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

1. If $\overline{T}_v$ is a child component of $\overline{T}_u$, then $d(u, v) = \min_{y \in B_0^j} (d(u, y) + d(y, v))$, where both distances have been computed in $L_j(u)$ and $R(v)$. This process requires $O(t)$ time.
2. If $\overline{T}_v$ is not a child component of $\overline{T}_u$, the process repeats recursively for the recursion of Step 3 and bags B_0^j and B_0^v, where B_0^v is the root of $\overline{T}_v$. Lemma 7 implies that

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

and the goal is to retrieve all distances $d(u', v')$. Using the same process as for u, v, all $d(u', v')$ are retrieved recursively. The process might be repeated further on the recursion of Step 3, for up to $\lambda + 1$ levels. Hence the worst case time for answering the query is $O((\lambda + 1) \cdot t^2)$. The core process is depicted in Figure 9.

Pair query. The preprocessing and query phases for answering shortest path queries (u, v) where B_v is an ancestor of B_u is similar to that where B_u is ancestor of B_v. In order to handle general pair queries, additionally preprocess T to answer LCA queries in constant time. Let B be the LCA of B_u and B_v. As in the description of queries above, we can compute the sets $M = \{(y, d(u, y)) : y \in B\}$ and $N = \{(y, d(y, v)) : y \in B\}$ in $O((\lambda + 1) \cdot t^2)$ time, and return $d(u, v) = \min_{(y, z_1) \in M, (y, z_2) \in N} (z_1 + z_2)$.

Single-source queries. Since we have computed local distances for each bag, we can use the single-source query algorithm ImpSiSoShPa from Theorem 3 to answer single-source queries in $O(n \cdot t)$ time.

Correctness. The preprocessing consists of summarizing distances along paths $B \rightsquigarrow B'$ of T, for all $u \in B$ and $v \in B'$, where B and B' are chosen conveniently to allow for fast queries. The correctness of ImPreShPa then follows directly from Lemma 7.

Theorem 5. *Let $\lambda \in \mathbb{N}$, a weighted graph $G = (V, E)$ of n nodes and m edges and a semi-nice tree-decomposition* $\mathrm{Tree}(G)$ *of G of width t that consists of $O(n)$ bags be given. The algorithm* ImPreShPa *correctly answers pair and single-source shortest path queries on G and requires*

- $O\left((\lambda + 2) \cdot n \cdot t^2 \cdot \log^{(\lambda)*} n\right)$ *preprocessing time;*

[2]This can be done using the algorithm in [25] which we currently use for LCA queries

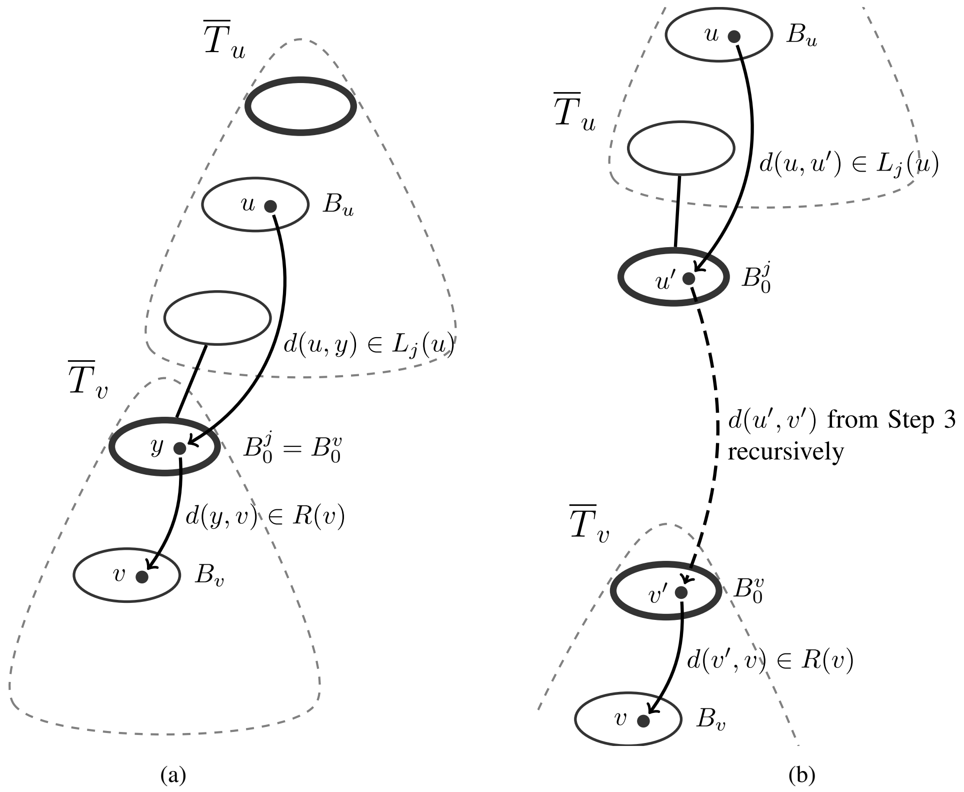

Figure 9: The two cases described in the query (u, v). Boldface bags are the root bags of their components. (a) If $\overline{T}_v$ is a child component of $\overline{T}_u$, the answer is retrieved from Step 2 recursion, by combining $L_j(u)$ and $R(v)$.(b) If $\overline{T}_v$ is not a child component of $\overline{T}_u$, the additional distances $d(u', v')$ are retrieved from Step 3 recursion.

- $O\left((\lambda+2) \cdot n \cdot t \cdot \log^{(\lambda)*} n\right)$ *space;*
- $O\left((\lambda+1) \cdot t^2\right)$ *pair shortest path query time; and*
- $O(n \cdot t)$ *single-source shortest path query time.*

7 Preprocessing Linear in n

In this section we describe a modification of the preprocessing described in Section 6 that reduces the preprocessing time to linear in n (i.e., $O\left(n \cdot t^2\right)$), at the expense of increasing the pair query time by $\alpha^2(n)$ (i.e., $O\left(t^2 \cdot \alpha^2(n)\right)$.

Note that by using $\lambda = \alpha(n) - 1$ in the preprocessing phase of algorithm IMPRESHPA of Section 6, we achieve preprocessing time $O\left(t^2 \cdot n \cdot \alpha^2(n)\right)$. Intuitively, the super-linear bound in n arises because of $O(\alpha(n))$ levels of recursion in Step 2, each one spawning $\alpha(n)$ recursions in Step 3, until the parameter λ becomes -1. In this section we modify IMPRESHPA to obtain an algorithm called LIPRESHPA (linear preprocessing for shortest path) that slightly alters Step 2 to remove the dependency on $\alpha^2(n)$. In this direction, we consider only the case where $t \leq \frac{n}{\alpha^2(n)}$. The algorithm LIPRESHPA is obtained by applying the following modifications to IMPRESHPA.

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

2. Skip Step 4.

It follows easily from the analysis of Lemma 10 that the preprocessing of LiPreShPa requires $O\left(n \cdot t^2\right)$ time and $O(n \cdot t)$ space.

Ancestor pair query. Given a query (u, v), where B_u is an ancestor of B_v, proceed as follows. If B_u and B_v belong to the same component $\overline{T}_i$ of the modified Step 2, the query is answered by performing the single-source shortest path search in the component $\overline{T}_i$ starting from B_u. This is run as the single-source query of Section 4.2, with additionally restricting the DFS in the nodes that appear in $\overline{T}_i$. If B_u and B_v appear in different components $\overline{T}_u$ and $\overline{T}_v$ respectively, a single-source shortest path algorithm is executed in each one to determine the distances $d(u, u')$ and $d(v', v)$ for all $u' \in B_0^j$ and $v' \in B_0^v$, where B_0^j is the the root of the unique child component of $\overline{T}_u$ on the path $B_u \rightsquigarrow B_v$, and B_0^v is the root of $\overline{T}_v$. Then for all such u', v', the distances $d(u', v')$ are determined as in the algorithm ImPreShPa, from the recursion of Step 3, and $d(u, v)$ is then

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

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

Pair query. The preprocessing and query phases for answering shortest path queries (u, v) where B_v is an ancestor of B_u is similar to that where B_u is ancestor of B_v. In order to handle general pair queries, additionally preprocess T to answer LCA queries in constant time. Let B_i be the LCA of B_u and B_v. As in the description of queries above, we can compute the sets $M = \{(y, d(u, y)) : y \in B_i\}$ and $N = \{(y, d(y, v)) : y \in B_i\}$ in $O((\lambda + 1) \cdot t^2)$ time, and return $d(u, v) = \min_{(y, z_1) \in M, (y, z_2) \in N}(z_1 + z_2)$.

Correctness. The correctness of LiPreShPa follows from the correctness of LiPreShPa and ImPreShPa, and we thus obtain the following theorem.

Theorem 6. *Let a weighted graph $G = (V, E)$ of n nodes and m edges and a semi-nice tree-decomposition* $\mathrm{Tree}(G)$ *of G of width t, for $t \leq \frac{n}{\alpha^2(n)}$, that consists of $O(n)$ bags be given. The algorithm* LiPreShPa *correctly answers single-source and pair shortest path queries on G and requires*

- *$O(n \cdot t^2)$ preprocessing time;*
- *$O(n \cdot t)$ space;*
- *$O(t^2 \cdot \alpha^2(n))$ pair shortest path query time; and*
- *$O(n \cdot t)$ single-source shortest path query time.*

Remark 4. Note that the algorithm LiPreShPa requires $O(t^2 \cdot \alpha^2(n))$ pair query time compared to the query time of $O(t^4 \cdot \alpha(n))$ from [10], and thus, is slower when $t^2 \leq \alpha(n)$. To obtain an algorithm which is faster, even for such small t, in each component created in Step 2, we preprocess the component similarly to an algorithm in [2] with linear preprocessing time in n' and logarithmic query time in n' where n' is the size of the component. By using the technique of local distance computation introduced in this paper, we obtain an $O(n' \cdot t^2)$ preprocessing time and $O(n' \cdot t)$ space for a component of size n' and query time of $O(\log(n') \cdot t^2)$. Since each component has size $O(\alpha^2(n) \cdot t)$ and there are $O(\frac{n}{\alpha^2(n) \cdot t})$ of them, this requires $O(n \cdot t^2)$ time for preprocessing, $O(t^2 \cdot \log t \cdot \log(\alpha^2(n))) = O(t^2 \cdot \log t \cdot \alpha(n))$ time for a pair query, and $O(n \cdot t)$ space. Thus we obtain bounds that are better than the previous preprocessing time and space, and pair query time [10].

8 Space vs Query Time Trade-off for Sub-linear Space

The algorithm in Section 7 suggests a way to trade query time for space. We call this algorithm the LowSpShPa (for low space shortest path) algorithm. The idea is to create sufficiently large components in the initial partitioning of $\overline{T}$, for which no preprocessing is done. Then, a summarization $\overline{T}'$ of $\overline{T}$ is of sufficiently small size to be preprocessed by LIPRESHPA. Answering a query (u, v) is handled similarly as in LIPRESHPA, but requires additional time for processing the components in which u and v appear (since they have not been preprocessed).

Remark 5 (Oracle). To distinguish the input space from the working space, we consider that the algorithm LOWSPSHPA has oracle access to (1) the weighted graph G; and (2) a semi-nice tree-decomposition $\mathrm{Tree}(G)$ of G of width t, for $t \leq \frac{n}{\alpha^2(n)}$, that consists of $O(n)$ bags; and (3) for each bag $B \in \mathrm{Tree}(G)$ the level of B in $\mathrm{Tree}(G)$ and the number of bags in the subtree $T(B)$.

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

Tree partitioning: The algorithm LOWSPTREEPART**.** We first present the algorithm LOWSPTREEPART for computing a tree partitioning in little space. Given a semi-nice tree-decomposition $T = \mathrm{Tree}(G)$ for a given graph G over n nodes and a number $j \leq n$ the algorithm LOWSPTREEPART partitions T into $O(\frac{n}{j})$ connected components forming a component tree, each containing $O(j)$ bags, in $O(n)$ time and $O(\frac{n}{j})$ extra space. The partitioning is, like in Section 6, based on a post-order DFS traversal of the tree. In the DFS traversal, the algorithm has a *component forest* (which forms a component tree at the end of the traversal), a set of trees, where each tree consists of bags, which are roots of components. Also, in the DFS traversal, the algorithm has a stack of triples (ℓ, c, L) where ℓ is a level, c is a number (of bags), and L is a list of trees in the component forest. Consider a step of the DFS traversal, in which the algorithm considers some bag B at level i:

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

Lemma 11. *Given an oracle as described in Remark 5 and a number j, the algorithm* LOWSPTREEPART *computes a partitioning of T into $O(n/j)$ partitions each of size between j and $2 \cdot j$, except for one component, in time $O(n)$ and requiring $O(n/j)$ extra space.*

Proof. It is easy to see that the algorithm is correct.

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

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

Local root distance computation: The algorithm LOWSPLOCDIS**.** Let $\epsilon > 0$ be a given constant, and let $A = n^\epsilon \cdot t^2$. Consider some component tree $\overline{T}$. We now describe how to recursively perform the local distance computation (as described in Section 4) of roots of components in $\overline{T}$ (and find negative cycles anywhere). The resulting algorithm will be called LOWSPLOCDIS and will require $O(n \cdot t^2)$ time and $O(A)$ space. We will describe the computation on a (sub)-component, using recursion. Consider some partitioning of $\overline{T}$ and a component C and let K be the size of C. Let $\{B_1, B_2, \ldots B_j\}$ be the set of roots of components in $\mathrm{Ext}(C)$. We have two parts, each corresponding to a pass of the algorithm LOCDIS. The first pass is as follows:

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

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

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

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

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

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

Lemma 12. *Given an $\epsilon > 0$, the algorithm* LOWSPLOCDIS *requires $O(n \cdot t^2)$ time and $O(n^\epsilon \cdot t^2)$ space.*

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

We will now prove the following claim:

Claim 3. *A given bag B of the tree-decomposition is in at most 2 components at the lowest level.*

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

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

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

Correctness. The correctness of the base case follows directly from Lemma 4 (which shows the correctness of LocDis). Note that instead of starting the first pass from the leaves and processing bottom-up in LocDis, it suffices to iterate over bags, such that all bags below the bag have already been processed by the first pass. This gives us the ordering used in LowSpLocDis on the components. Futheremore, since LowSpLocDis do not recurse on components C, but on the extended component $\mathrm{Ext}(C)$, we see that all leaves of $\mathrm{Ext}(C)$ (which are either leaves of the tree-decomposition or roots of some lower component) have either been processed by the first pass of LowSpLocDis in case they are roots of some lower components, or are leaves in the tree-decomposition. It follows that the first pass of LowSpLocDis is correct. The correctness of the second pass is similar. We run both passes in the second pass because the matrices computed in the sub-components would otherwise be thrown away between passes. This give us the following lemma.

Lemma 13. *Given an oracle as described in Remark 5, a constant $\epsilon > 0$, and a component tree $\overline{T}$, the algorithm* LowSpLocDis *finds a negative cycle if it exists in G, and otherwise computes the local distances for the roots of the components in $\overline{T}$ and in either case requires $O(n \cdot t^2)$ time and $O(n^\epsilon \cdot t^2)$ space.*

Similarly to LowSpLocDis one can also compute the root and leaf distances, see Section 6, in $O(n \cdot t^2)$ time and $O(n^\epsilon \cdot t^2)$ space.

Remark 6. It is straightforward to see that if instead of splitting into components of size $O(\frac{K}{n^\epsilon})$, we consider a modified algorithm that splits into components of size $\frac{n}{4^i}$ at recursion depth i (using LowSpTreePart), until components of size at most 4 are obtained, then the recursion depth is at most $\log(n)$ (note that we use 4^i because the partitions might be a factor of two larger). This modified algorithm (of LowSpLocDis) would still be correct following the same argument. Also, using a similar argument to the one in Lemma 12 (especially we can still use the claim because we still never recurse on a component of size 1) we get that the time required is $O(n \cdot t^2 \cdot \log^2(n))$ and space required is $O(t^2 \cdot \log^2(n))$. Similar modified algorithms are also obtained for root and leaf distance computation. Using the modified algorithms we can solve pair queries u, v as follows: First find the LCA L of B_u and B_v (by processing bottom-up from B_u and B_v in the tree-decomposition using the oracle for the tree-decomposition as well as for finding the level of B_u and B_v — the latter to synchronize the movement). Then using the local distances $d(u, w)$ and $d(w, v)$, for each $w \in L$ we can the find the distance from u to v as $\min_{w \in L} d(u, w) + d(w, v)$. Those distances can be obtained as follows: Let the partition P be the one that partitions the tree into four components, one with root each of B_u, B_v, L, and the root of T. The distances $d(u, w)$ and $d(w, v)$ are then part of the root and leaf distances computed on T if we use the partitioning P. Thus, we can compute the shortest path from u to v in G in $O(n \cdot t^2 \cdot \log^2(n))$ time and $O(t^2 \cdot \log^2(n))$ space. Note that with the modified algorithm for local distance computation we can also report the existence of a negative cycle in $O(n \cdot t^2 \cdot \log^2(n))$ time and $O(t^2 \cdot \log^2(n))$ space.

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

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

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

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

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

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

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

Correctness. In each case of the algorithm we find the shortest path among paths of the form $w_0 = u \rightsquigarrow w_1 \rightsquigarrow w_2 \rightsquigarrow \cdots \rightsquigarrow v = w_k$, where each of w_i ranges over some bag B_i. It is easy to see that the bags B_i are bags on the path from B_u to B_v in T. We see that in any path from u to v there must be a node in each B_i following Remark 1. Also, it is straightforward to see that we compute the distance between each pair w_i, w_{i+1} correctly for all i, using either pair queries from algorithm ImPreShPa in Section 6 or using LowSpLocDis. Finally, it is easy to see that we compute the distance from u to v correctly given that we computed the distance between each pair w_i, w_{i+1} correctly.

Time and space requirements. It is clear that our preprocessing can be done as described in time $O(n \cdot t^2)$ and $O(n^\epsilon \cdot t)$ space. In regards to the query, we see that the local and root and leaf distance preprocessing (of which we do at most 4) requires $O(n^{1-\epsilon} \cdot t^2 \cdot \alpha^2(n))$ time (the size of each component times t^2) similarly to Lemma 3 and $O(n^\epsilon \cdot t^2)$ space, using the algorithms LowSpTreePart and LowSpLocDis. In the query we also use our data-structure (computed in the preprocessing) upto two times, which takes $O(t^2 \cdot \alpha^2(n))$ time each and then at the end we use $O(t^2)$ time to answer the query (we only find $O(t^2)$ edges and only need to consider certain paths of length at most 5). This then takes $O(n^{1-\epsilon} \cdot t^2 \cdot \alpha^2(n))$ time and $O(n^\epsilon \cdot t^2)$ space.

This establishes the following theorem.

Theorem 7. *Given a constant $\epsilon > 0$ and oracle access to (1) a weighted graph $G = (V, E)$ of n nodes and m edges; and (2) a semi-nice tree-decomposition $\mathrm{Tree}(G)$ of G of width t, for $t \leq \frac{n}{\alpha^2(n)}$, that consists of $O(n)$ bags; and (3) for each bag $B \in \mathrm{Tree}(G)$ the number of bags in the subtree under B. Then the algorithm* LowSpShPa *correctly answers pair shortest path queries on G and requires*

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

Remark 7. Technically, we only have access to an oracle of the tree-decomposition and thus cannot store the information in the roots of the components as described. This can be solved by having the component tree as a data-structure and various sub-components at any given 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] N. Alon and B. Schieber. Optimal preprocessing for answering on-line product queries. Technical report, Tel Aviv University, 1987.

[3] 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.

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

[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] 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.

[7] 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.

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

[9] 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.

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

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

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

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

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

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

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

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

[18] 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.

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

[20] T. Kloks. *Treewidth, Computations and Approximations*, volume 842 of *Lecture Notes in Computer Science*. Springer, 1994.

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

[22] 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.

[23] 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.

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

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

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

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

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