I/O-Efficient Algorithms for Graphs of Bounded Treewidth*

Anil Maheshwari†

School of Computer Science
Carleton University
Ottawa, ON K1S 5B6
Canada
maheshwa@scs.carleton.ca

Norbert Zeh†,‡

Faculty of Computer Science
Dalhousie University
Halifax, NS B3H 2Y5
Canada
nzeh@cs.dal.ca

Abstract

We present an algorithm that takes $\mathcal{O}(\text{sort}(N))$ I/Os[1] to compute a tree decomposition of width at most k, for any graph G of treewidth at most k and size N. Given such a tree decomposition, we use a dynamic programming framework to solve a wide variety of problems on G in $\mathcal{O}(N/(DB))$ I/Os, including the single-source shortest path problem and a number of problems that are NP-hard on general graphs. The tree decomposition can also be used to obtain an optimal separator decomposition of G. We use such a decomposition to perform depth-first search in G in $\mathcal{O}(N/(DB))$ I/Os.

As important tools that are used in the tree decomposition algorithm, we introduce *flippable DAGs* and present an algorithm that computes a perfect elimination ordering of a k-tree in $\mathcal{O}(\text{sort}(N))$ I/Os.

The second contribution of our paper, which is of independent interest, is a general and simple framework for obtaining I/O-efficient algorithms for a number of graph problems that can be solved using greedy algorithms in internal memory. We apply this framework in order to obtain an improved algorithm for finding a maximal matching and the first deterministic I/O-efficient algorithm for finding a maximal independent set of an arbitrary graph. Both algorithms take $\mathcal{O}(\text{sort}(|V|+|E|))$ I/Os. The maximal matching algorithm is used in the tree decomposition algorithm.

1 Introduction

1.1 Background and Motivation

I/O-efficient graph algorithms have received considerable attention because massive graphs arise naturally in many applications; such as geographic information systems, web modeling, and telecommunications research. Recent web crawls, for example, produce graphs of on the order of 200 million nodes and 2 billion edges [12]. When working with such large data sets, the transfer of data between

*An abstract of this paper was presented at the 12th Annual ACM-SIAM Symposium on Discrete Algorithms [26].

†Research supported by NSERC and NCE GEOIDE.

‡Part of this work was done while the second author was a Ph.D. student at the School of Computer Science of Carleton University.

[1] $\text{sort}(N) = \Theta((N/(DB)) \log_{M/B}(N/B))$ is the number of I/Os it takes to sort N data items.

internal and external memory, and not the internal memory computation, is often the bottleneck. Thus, I/O-efficient algorithms can lead to considerable run-time improvements.

Recent work in web modeling uses depth-first search, breadth-first search, and the computation of shortest paths and connected components as primitive operations for investigating the structure of the web. While these fundamental problems are well studied in the RAM model of computation, they remain challenging in environments where random access is expensive; all existing internal memory algorithms for these problems exhibit a highly random memory access pattern and hence perform poorly in such environments. Vitter [38] identifies finding I/O-optimal algorithms for the single-source shortest path problem, and thus also for breadth-first search, as one of the most important open problems in the area of I/O-efficient graph algorithms.

Previous efforts to solve the single-source shortest path (SSSP) problem, breadth-first search (BFS), and depth-first search (DFS) without exploiting structural properties of the given graph have led to algorithms that perform well on dense graphs, but whose performance breaks down on sparse graphs. In general, it seems extremely hard to remedy this situation. However, for restricted classes of sparse graphs, such as outerplanar or planar graphs, I/O-optimal SSSP-, BFS-, and DFS-algorithms have been developed. One of the contributions of our paper is the development of such algorithms for yet another class of sparse graphs: graphs of bounded treewidth. Even though the treewidth of a graph has been introduced as a rather theoretical measure of the complexity of a number of NP-hard problems on the graph, recent studies suggest that for instance the treewidth of graphs produced by web crawls is bounded by a small constant. Thus, I/O-efficient algorithms for graphs of bounded treewidth can be used in web modeling applications. From a more theoretical point of view, it is interesting to observe that many important, well-studied graph classes have bounded treewidth. These classes include trees; partial k-trees; series-parallel graphs; k-outerplanar graphs; Halin graphs; control flow graphs of goto-free programs; and chordal, interval, and circular arc graphs with maximum clique size k. Thus, our algorithms can be used to solve the problems we study on any graph that belongs to one of these classes.

At the core of our algorithms is an I/O-efficient algorithm for computing a tree decomposition of a graph of bounded treewidth. In internal memory, such a decomposition can be computed in linear time [8, 11]. Together with the results of [4, 6, 8, 9, 10, 11, 16, 22, 24, 28, 33], this implies that many NP-hard problems can be solved in linear time for graphs of bounded treewidth. However, since a disk access is six orders of magnitude more expensive than an access to main memory, even these specialized algorithms touch on the threshold of intractability as soon as the graphs become too large to fit into internal memory. In this paper, we show that many of the problems solved by these algorithms can be solved in optimal $\mathcal{O}(N/(DB))$ I/Os, once a tree decomposition of the graph is given. Thus, these problems remain tractable even if the graphs are extremely big.

1.2 Model of Computation

The difference in access time between internal and external (disk-based) memory creates a considerable bottleneck as soon as data sets are too large to be held in internal memory. This I/O bottleneck is becoming more significant as parallel computing gains popularity and CPU speeds increase, since disk speeds are not keeping pace [35, 38]. Thus, it is important to take the number of input/output (I/O) operations performed by an algorithm into consideration, when estimating its efficiency. This issue is captured in the *parallel disk model* (PDM) [39], as well as a number of other external memory models [17, 40]. We adopt the PDM as our model of computation for this paper due to its simplicity and the fact that we consider only a single processor.

In the PDM, an *external memory* consisting of D disks is attached to a machine with an internal memory capable of holding M data items. Each of the disks is divided into blocks of B consecutive data items. Up to D blocks, at most one per disk, can be transferred between internal and external memory in a single I/O operation (or I/O for short). The complexity of an algorithm is the number of I/O operations it performs.

In [39], it is shown that sorting N data items takes $\mathrm{sort}(N) = \Theta((N/(DB)) \log_{M/B}(N/B))$ I/Os; permuting them takes $\mathrm{perm}(N) = \Theta(\min\{N, \mathrm{sort}(N)\})$ I/Os; scanning the list of data items takes $\mathrm{scan}(N) = \Theta(N/(DB))$ I/Os.

1.3 Previous Work

Previous work on algorithms for graphs of bounded treewidth has focused on computing a tree decomposition for a given graph of bounded treewidth and exploiting the structural information provided by such a decomposition in order to solve otherwise intractable problems efficiently.

In [5, 6, 7], algorithms are presented that solve a number of NP-hard problems in linear time if the given graph has bounded treewidth and a tree decomposition of the graph is given as part of the input. An interesting framework for solving these problems on graphs of bounded treewidth, without computing a tree decomposition of the graph, is presented in [4]; the resulting algorithms are based on graph reduction and take linear time, but use superlinear space.

The problem of computing a tree decomposition efficiently has been studied by a number of authors [8, 9, 11, 28, 33], culminating in the linear-time algorithm of [8], which uses the following result of [11]: Given a graph G of treewidth at most k and a tree decomposition of G whose width is at most ℓ, for some constant ℓ, a tree decomposition of G whose width is at most k can be found in linear time. Improved algorithms for graphs of treewidth at most two are presented in [9].

Parallel (PRAM) algorithms for computing a tree decomposition are proposed in [9, 10, 16, 24]. From our perspective, the most interesting algorithm is that of [10], which computes a tree decomposition in $\mathcal{O}(\log^2 N)$ time using $\mathcal{O}(N)$ operations. The algorithm is a non-trivial parallelization of the algorithm of [8]. Improved and simplified algorithms for graphs of treewidth at most two are presented in [9, 16].

We are not aware of any results on computing tree decompositions of graphs or solving NP-hard problems on graphs of bounded treewidth I/O-efficiently. However, the outerplanar embedding algorithm of [25] can be used to obtain tree decompositions of width two for outerplanar graphs, in $\mathcal{O}(\mathrm{sort}(N))$ I/Os. The parallel tree decomposition algorithm of [10] can be combined with the PRAM-simulation technique of [14], to obtain an algorithm that takes $\mathcal{O}(\mathrm{sort}(N) \log_2 N)$ I/Os to compute a tree decomposition of width at most k, for a graph of treewidth at most k.

The most I/O-efficient SSSP-algorithm for general undirected graphs with non-negative edge weights [23] takes $\mathcal{O}(|V| + (|E|/B) \log_2(|V|/B))$ I/Os. A recent result of [31] shows that for graphs with minimal edge weight w and maximal edge weight W, the SSSP-problem can be solved in $\mathcal{O}\left(\sqrt{(|V||E|/B) \log_2(W/w)} + \mathrm{sort}(|V| + |E|)\right)$ I/Os. For sparse graphs with a bounded range of edge weights, this greatly improves on the result of [23]. Breadth-first search can be seen as the single-source shortest path problem with unit weights on the edges of G. The best known BFS-algorithm for undirected graphs [29] takes $\mathcal{O}\left(\sqrt{|V||E|/B} + \mathrm{sort}(|V| + |E|)\right)$ I/Os; the best known DFS-algorithm [13] takes $\mathcal{O}((|V| + |E|/B) \log_2 |V|)$ I/Os.

$\mathcal{O}(\mathrm{sort}(|V|))$ I/O algorithms for BFS and DFS in outerplanar graphs are presented in [25]. The same paper proves $\Omega(\mathrm{perm}(|V|))$ I/O lower bounds for outerplanar embedding, BFS and DFS.

Recently, an $\mathcal{O}(\text{sort}(|V|))$ I/O algorithm for the single-source shortest path problem on embedded planar graphs has been proposed in [3]. The algorithm assumes that a small separator of the graph is given. Together with two recent algorithms for computing such a separator and a planar embedding of a planar graph [27], this gives an $\mathcal{O}(\text{sort}(|V|))$ I/O algorithm for the SSSP-problem on planar graphs.

In internal memory, simple greedy algorithms can be used to compute a maximal matching or a maximal independent set of a graph in $\mathcal{O}(|V| + |E|)$ time. The best known deterministic algorithm for computing a maximal matching I/O-efficiently [1] takes $\mathcal{O}(\text{sort}(|E|)\log_2(V/B))$ I/Os. No deterministic algorithm for finding a maximal independent set I/O-efficiently is known. In [1], randomized algorithms for these two problems are proposed; their I/O-complexity is $\mathcal{O}(\text{sort}(|E|))$ with high probability.

A perfect elimination ordering of a chordal graph can be found in linear time using algorithms of [32, 34, 36]. In the PRAM model, Klein [21] shows how to compute a perfect elimination ordering in $\mathcal{O}(\log^2 |V|)$ time, using $\mathcal{O}((|V| + |E|)/\log |V|)$ processors. In external memory, the sequential approaches seem unfeasible, as they use search-strategies similar to breadth-first search, while a simulation of Klein's approach would lead to a suboptimal I/O-complexity. We are not aware of any results on computing a perfect elimination ordering in external memory.

1.4 Our Results

The two main contributions of our paper are an $\mathcal{O}(\text{sort}(N))$ I/O algorithm for computing a tree decomposition of a graph of bounded treewidth and a framework for deriving I/O-efficient algorithms from greedy algorithms for a number of graph problems. The tree-decomposition algorithm is an I/O-efficient version of the algorithm of [8, 11]. We identify the subproblems to be solved in order to make the algorithm I/O-efficient and provide I/O-efficient solutions to these subproblems. Given the tree decomposition, the dynamic programming framework required to solve the single-source shortest path problem, depth-first search, and the NP-hard problems considered in [5, 6, 7] on graphs of bounded treewidth can be realized in $\mathcal{O}(\text{scan}(N))$ I/Os. Using our framework for I/O-efficient greedy algorithms, we obtain improved and much simplified, deterministic algorithms for computing maximal matchings and maximal independent sets for arbitrary graphs.

As part of our tree decomposition algorithm, we present solutions to two problems that are of less general interest, but may nevertheless prove useful in designing I/O-efficient algorithms for other graph problems. We present an $\mathcal{O}(\text{scan}(N))$ I/O algorithm for computing a perfect elimination ordering of a k-tree, given a tree decomposition of the graph. The second result deals with the following generalization of series and parallel compositions used to construct series-parallel graphs: Every series-parallel st-graph G can be constructed from a set of edges by repeated application of series compositions and parallel compositions. We extend these operations so that it is allowed to flip all edges in one of the two graphs before the composition. If we perform these flips explicitly, it is easy to construct an example where $\Omega(N^2)$ edge flips are necessary to construct a DAG of size N. We introduce *flippable DAGs* as a technique to perform these flips implicitly, at the cost of $\mathcal{O}(1)$ updates per composition. Once the final graph is constructed, we perform a post-processing phase, which takes $\mathcal{O}(\text{sort}(N))$ I/Os to determine the correct direction of each edge.

1.5 Organization of the Paper

In Section 2, we introduce the basic terminology and review important results about tree decompositions. The description of our tree decomposition algorithm requires a good understanding of the results on integer sequences shown in [11]. We review these results in Section 2.3. In Section 3, we describe flippable DAGs, which we use to maintain implicit representations of path decompositions in an I/O-efficient manner. In Section 4, we present our framework for making greedy graph algorithms I/O-efficient. In Section 5, we recall the outline of the tree-decomposition algorithm of [8] and show how to perform most of the steps of the algorithm in $\mathcal{O}(\text{sort}(N))$ I/Os. In Section 6, we present an I/O-efficient version of the algorithm of [11], which is used as a subroutine in the tree-decomposition algorithm of [8]. Section 6 is divided into two parts. In the first part, we describe how to test whether a graph has treewidth at most ℓ, given a tree decomposition of width k. This part is a straightforward simulation of the testing phase of the algorithm of [11]. We describe it in detail to gain some insight into the relationship between tree decompositions and their constant size descriptions used in the testing phase. This information is used in the second phase, which constructs a tree decomposition of width at most k. In order to avoid accessing the nodes of the constructed tree decomposition at random, the second phase of the algorithm uses flippable DAGs to represent partial path decompositions that are part of the tree decomposition. As a result, it differs from the construction phase of the algorithm of [11], which makes use of the random access capabilities provided by the RAM model. In Sections 7 and 8, we present two algorithms for subproblems to be solved as part of the algorithm described in Section 6. In Section 7, we show how to obtain a *nice* tree decomposition I/O-efficiently from a given tree decomposition. In Section 8, we show how to compute a perfect elimination ordering of a k-tree. Our SSSP- and DFS-algorithms, and solutions to NP-hard problems are discussed in Section 9. We present concluding remarks and discuss a few open problems in Section 10.

2 Preliminaries

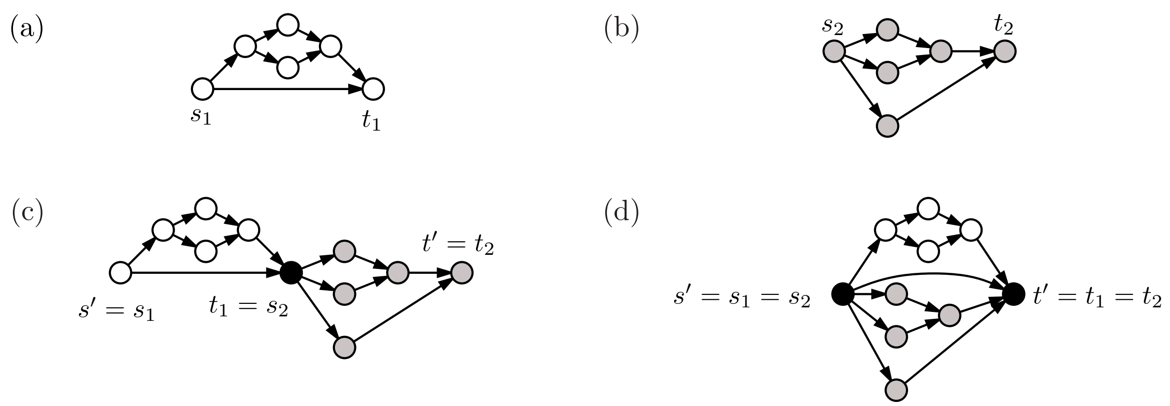
2.1 Basic Concepts

We assume that the reader is familiar with basic graph theoretic concepts. A good introduction to graph theory is given for instance in [20, 37]. In this section, we introduce the notation used in this paper. We denote the edges of an undirected graph by unordered pairs (2-sets) $\{v, w\}$, while we write directed edges as ordered pairs (v, w). We refer to undirected graphs simply as *graphs* and to directed graphs as *digraphs*. For a digraph $G = (V, E)$, we define the *underlying undirected graph* as $U(G) = (V, \{\{v, w\} : (v, w) \in E\})$. We denote the set of neighbors of a vertex v in a graph G by $\Gamma_G(v)$; $\deg_G(v)$ denotes the degree of v in G. For a digraph G, we denote the sets of in- and out-neighbors of a vertex v by $\Gamma^-_G(v)$ and $\Gamma^+_G(v)$, respectively; the in and out-degrees of v are denoted by $\deg^-_G(v)$ and $\deg^+_G(v)$, respectively.

For a graph $G = (V, E)$ and a subset $X \subseteq V$, we denote the subgraph $(X, \{\{v, w\} \in E : v, w \in X\})$ of G induced by vertex set X as $G[X]$. We write $G - X$ to denote the graph $G[V \setminus X]$; for a vertex $x \in V$, we write $G - x$ to denote the graph $G - \{x\}$.

A *directed acyclic graph* (DAG) is a digraph $G = (V, E)$ that does not contain directed cycles. We denote all vertices v with $\deg^-(v) = 0$ as *sources* in G and all vertices with $\deg^+(v) = 0$ as *sinks* in G. An *st-graph* is a DAG with exactly one source s and one sink t. An st-graph is *series-parallel* if it consists of a single edge (s, t) or it can be obtained from two series-parallel graphs G_1 and G_2

Figure 1 Two series-parallel graphs G_1 (a) and G_2 (b), their series composition (c), and their parallel composition (d).



with sources s_1 and s_2 and sinks t_1 and t_2 by identifying t_1 with s_2 (series composition; Figure 1c) or by identifying s_1 with s_2 and t_1 with t_2 (parallel composition; Figure 1d). In this paper, we also consider the graph consisting of a single vertex to be series-parallel, but this graph may be combined with another graph only in a series composition.

Given an assignment $\omega : E \to \mathbb{R}$ of real weights to the edges of graph $G = (V, E)$, we define the weight $\omega(H)$ of a subgraph $H = (W, F)$ of G as $\omega(H) = \sum_{e \in F} \omega(e)$. We call a subgraph H *negative* or *positive* if its weight is negative or positive, respectively. Given a graph G that does not contain negative cycles, the shortest path $\pi(v, w)$ from $v \in V$ to $w \in V$ is the path of minimum weight among all paths from v to w.

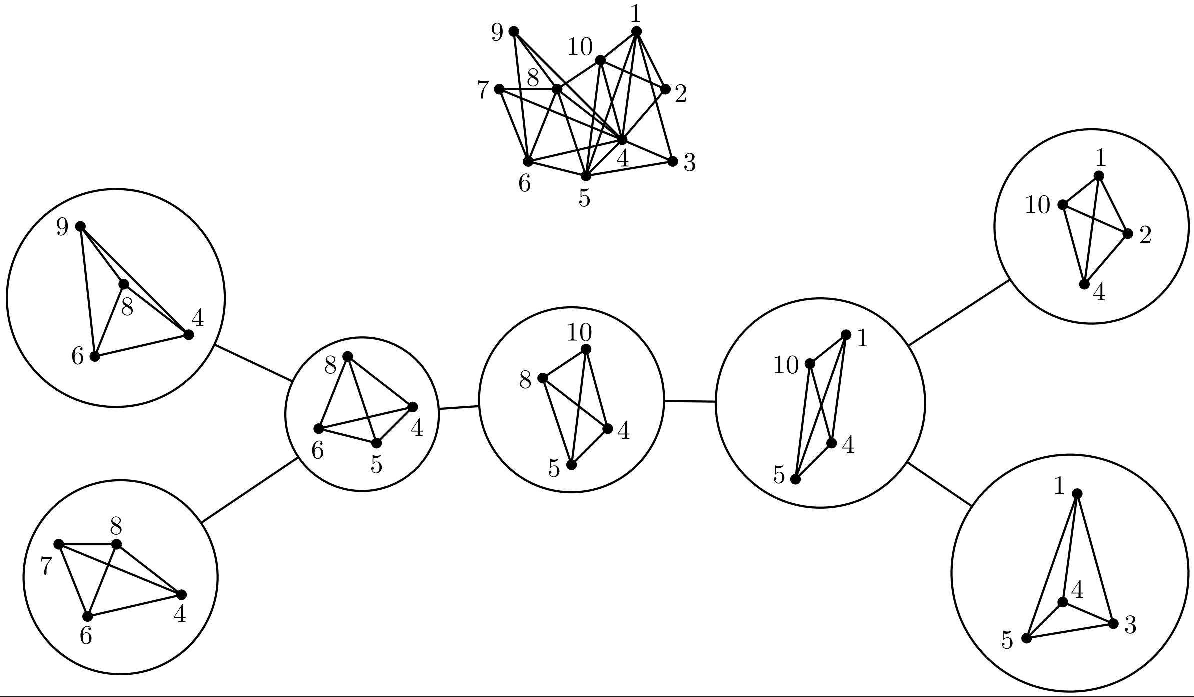
An *independent set* of a graph $G = (V, E)$ is a set $S \subseteq V$ such that no two vertices in S are adjacent. An independent set is *maximal* if every vertex in $V \setminus S$ is adjacent to a vertex in S. A *matching* of a graph $G = (V, E)$ is a set $\mathcal{M} \subseteq E$ of edges such that no two edges in $\mathcal{M}$ share an endpoint. A matching $\mathcal{M}$ is *maximal* if every edge in $E \setminus \mathcal{M}$ shares an endpoint with an edge in $\mathcal{M}$. In other words, maximal independent sets and matchings cannot be augmented to obtain larger independent sets or matchings.

A *clique* in a graph $G = (V, E)$ is a subset $W \subseteq V$ of vertices such that $\{v, w\} \in E$, for all $v \neq w$, $v, w \in W$. A vertex $v \in V$ is *simplicial* if $\Gamma_G(v)$ is a clique. Given a cycle $C = (v_0, \dots, v_n)$ in a graph G, a *chord* of C is an edge $\{v_i, v_j\}$ whose endpoints are not adjacent in C. An undirected graph $G = (V, E)$ is *chordal* if every cycle in G of length greater than three has a chord. A *perfect elimination ordering* (PEO) of G is an ordering $\prec$ of the vertices in V such that every vertex $v \in V$ is simplicial in the graph $G[\{w \in V : v \preceq w\}]$. It is well known that a graph is chordal if and only if it has a PEO [18].

2.2 Tree Decompositions and Treewidth

The treewidth of a graph gives an indication of how far away the graph is from being a tree or forest. The closer the graph is to being a forest, the smaller is its treewidth. As trees are among the simplest classes of graphs, and many hard graph problems become easy on trees, the treewidth of a graph is a good measure for the hardness of solving certain problems on this graph. The treewidth of a graph is defined through the concept of tree decompositions.

Figure 2 A graph G and a tree decomposition of width 3 for G.



Given an undirected graph $G = (V, E)$, a *tree decomposition* $\mathcal{D} = (\mathcal{X}, T)$ of G consists of a tree $T = (I, F)$ and a collection $\mathcal{X}$ of sets X_{i}, $\mathsf{i} \in I$, such that

(T1) $\bigcup_{\mathsf{i} \in I} X_{\mathsf{i}} = V$,

(T2) For every edge $\{v, w\} \in E$, there is a node $\mathsf{i} \in I$ such that $\{v, w\} \subseteq X_{\mathsf{i}}$, and

(T3) For any three nodes i, j, and k such that j is on the tree path from i to k, $X_{\mathsf{i}} \cap X_{\mathsf{k}} \subseteq X_{\mathsf{j}}$.

See Figure 2 for an example. To avoid confusion, we refer to the vertices of graph G as *vertices* and represent them using small italic letters, while we refer to the vertices of T as *nodes* and represent them using small sans serif letters. A tree decomposition is said to have *width* k if $|X_{\mathsf{i}}| \leq k + 1$, for all $\mathsf{i} \in I$. The *treewidth* of a graph G is the minimum width of all its tree decompositions. In particular, the treewidth of G is one if and only if G is a forest. We define the treewidth of a directed graph G to be the same as the treewidth of its underlying undirected graph.

A *rooted tree decomposition* is a tree decomposition with a distinguished root node. Given a rooted tree decomposition $\mathcal{D} = (\mathcal{X}, T)$ and a node i of T, let $\mathrm{Desc}(\mathsf{i})$ be the set of descendants of node i in T, including i; let $T_{\mathsf{i}} = T[\mathrm{Desc}(\mathsf{i})]$; let $G_{\mathsf{i}} = G[\bigcup_{\mathsf{j} \in \mathrm{Desc}(\mathsf{i})} X_{\mathsf{j}}]$; and let $\mathcal{D}_{\mathsf{i}} = (\{X_{\mathsf{j}} : \mathsf{j} \in \mathrm{Desc}(\mathsf{i})\}, T_{\mathsf{i}})$. A rooted tree decomposition is *nice* if each node of T is of one of the following types: A *start node* is a leaf. An *introduce node* i has one child j with $X_{\mathsf{i}} = X_{\mathsf{j}} \cup \{x\}$, for some $x \notin X_{\mathsf{j}}$. A *forget node* i has one child j with $X_{\mathsf{i}} = X_{\mathsf{j}} \setminus \{x\}$, for some $x \in X_{\mathsf{j}}$. A *join node* i has two children j and k with $X_{\mathsf{i}} = X_{\mathsf{j}} = X_{\mathsf{k}}$.

A *path decomposition* is a tree decomposition $\mathcal{D} = (\mathcal{X}, T)$ such that T is a path. We can write such a path decomposition as the sequence $Y = (X_1, \ldots, X_{|I|})$ of sets X_{i} along the path T. We define a rooted path decomposition to be a path decomposition one of whose endpoints has been chosen as the root. The *pathwidth* of a graph G is the minimum width of all possible path decompositions of G. Given two path decompositions Y_1 and Y_2, we denote the concatenation of Y_1 and Y_2 by $Y_1 \circ Y_2$. This operation is allowed only if $Y_1 \circ Y_2$ satisfies Properties T1–T3. A path decomposition $Y' = (X'_1, \ldots, X'_t)$ is an extension of a path decomposition $Y = (X_1, \ldots, X_s)$ if there are indices $1 = q_1 < \cdots < q_{s+1} = t + 1$ so that $X'_j = X_i$, for all $1 \leq i \leq s$ and $q_i \leq j < q_{i+1}$. In other words, path decomposition Y' can be obtained from Y by duplicating nodes. We denote the set of all extensions of Y by $E(Y)$.

2.3 Typical Sequences and Typical Lists

Next we recall the most important results from [11] on integer sequences and their typical sequences. These sequences play an important role in the algorithm for reducing the width of a given tree decomposition, which is presented in Section 6.

Given an integer sequence $a = (a_1, \ldots, a_n)$, let the *length* of a be $|a| = n$, and let $\max(a) = \max\{a_i : 1 \leq i \leq n\}$. For two sequences a and b of the same length, the *sum* $a + b$ of a and b is the sequence $c = (c_1, \ldots, c_n)$ with $c_i = a_i + b_i$, for $1 \leq i \leq n$. For a constant λ, let $a + \lambda$ be the sequence $(a_1 + \lambda, \ldots, a_n + \lambda)$. For two integer sequences a and b of the same length, we write $a \leq b$ if $a_i \leq b_i$, for all $1 \leq i \leq n$.

The *typical sequence* $\tau(a)$ of an integer sequence a is the sequence obtained after iterating the following operations until none of these operations is applicable:

Duplicate removal: Remove consecutive repetitions of the same element; that is, if $a_i = a_{i+1}$, remove a_{i+1} from a.

Typical operation: If the sequence contains two elements a_i and a_k, $i \leq k - 2$, such that for all $i \leq j \leq k$, $a_i \leq a_j \leq a_k$ or $a_i \geq a_j \geq a_k$, remove elements $a_{i+1}, \ldots, a_{k-1}$ from a.

For instance, the sequence $a = (1, 4, 4, 3, 5, 7, 8, 8, 6, 4, 1)$ has the typical sequence $\tau(a) = (1, 8, 1)$. To obtain $\tau(a)$, first remove the second 4 and the second 8 (duplicate removal); then delete entries $4, 3, 5, 7$ between the first 1 and the 8 and entries $6, 4$ between the 8 and the last 1 (typical operations). Bodlaender and Kloks [11] show that the typical sequence of an integer sequence is well defined, i.e., the order in which the above operations are applied is irrelevant.

Lemma 1 (Bodlaender/Kloks [11]) *If the elements in sequence a are non-negative integers and $\max(a) = k$, then $|\tau(a)| \leq 2k + 1$ and $\max(\tau(a)) = k$.*

An *extension* of an integer sequence $a = (a_1, \ldots, a_n)$ is a sequence $a^* = (a^*_1, \ldots, a^*_m)$ such that there are indices $1 = t_1 < t_2 < \cdots < t_{n+1} = m + 1$ so that for all $1 \leq i \leq n$ and $t_i \leq j < t_{i+1}$, $a_i = a^*_j$. Let $E(a)$ be the set of all extensions of a.

Lemma 2 (Bodlaender/Kloks [11]) *If $a^* \in E(a)$, then $\tau(a^*) = \tau(a)$.*

For two sequences a and b, the *ringsum* of a and b is the set $a \oplus b = \{a^* + b^* : a^* \in E(a) \wedge b^* \in E(b) \wedge |a^*| = |b^*|\}$.

Lemma 3 (Bodlaender/Kloks [11]) *Let a and b be two integer sequences and $c \in a \oplus b$. Then there exists an integer sequence $c' \in a \oplus b$ with $\tau(c) = \tau(c')$ and $|c'| \leq |a| + |b| - 1$.*

For two sequences $a = (a_1, \ldots, a_m)$ and $b = (b_1, \ldots, b_n)$, let the *concatenation* of a and b be the sequence $a \circ b = (a_1, \ldots, a_m, b_1, \ldots, b_n)$.

Lemma 4 (Bodlaender/Kloks [11]) *For two sequences a and b, $\tau(a \circ b) = \tau(\tau(a) \circ \tau(b))$.*

A *split* of a sequence $a = (a_1, \ldots, a_n)$ is a pair of sequences b and c such that $b = (a_1, \ldots, a_f)$ and either $c = (a_f, \ldots, a_n)$ or $c = (a_{f+1}, \ldots, a_n)$. In the former case, the split is of *type one*; in the latter case, it is of *type two*.

An *(integer) list* is a list $[a] = \left(a^{(1)}, a^{(2)}, \ldots, a^{(n)}\right)$, where each $a^{(i)}$ is an integer sequence.

- The *length* of a list is the number of sequences in the list.
- For a list $[a] = \left(a^{(1)}, \ldots, a^{(n)}\right)$, $\max[a] = \max\left\{\max\left(a^{(i)}\right) : 1 \le i \le n\right\}$.
- Two lists $[a]$ and $[b]$ have the same length *in the strong sense* if they have the same length and $\left|a^{(i)}\right| = \left|b^{(i)}\right|$, for all $1 \le i \le n$.
- For two lists $[a]$ and $[b]$ of the same length in the strong sense, we write $[a] \le [b]$ if $a^{(i)} \le b^{(i)}$, for all $1 \le i \le n$.
- For two lists of the same length in the strong sense, $[a] + [b]$ denotes the list $\left(a^{(1)} + b^{(1)}, \ldots, a^{(n)} + b^{(n)}\right)$.
- The *typical list* of a list $[a]$ is the list $\tau[a] = \left(\tau\left(a^{(1)}\right), \ldots, \tau\left(a^{(n)}\right)\right)$.
- The *extension set* of a list $[a]$ is the set $E[a] = \left\{[b] = \left(b^{(1)}, \ldots, b^{(n)}\right) : \forall_{1 \le i \le n} b^{(i)} \in E\left(a^{(i)}\right)\right\}$.
- The *ringsum* of two lists $[a]$ and $[b]$ of the same length is the set $[a] \oplus [b] = \left\{\left(c^{(1)}, \ldots, c^{(n)}\right) : \forall_{1 \le i \le n} c^{(i)} \in a^{(i)} \oplus b^{(i)}\right\}$.

All of the above results on integer sequences extend to integer lists.

3 Flippable DAGs

In the tree decomposition algorithm of Section 6, we have to perform the following operation repeatedly: Given two path decompositions Y_1 and Y_2 of two graphs G_1 and G_2, which possibly share vertices, construct a path decomposition Y of the graph $G = G_1 \cup G_2$ by "stretching" Y_1 and Y_2 appropriately so that they have the same length and then unioning the sets along path decompositions Y_1 and Y_2 in a pairwise manner. Since this stretch operation is expensive, and we have to perform it many times, we do not perform it explicitly. Instead, we represent Y_1 and Y_2 as series-parallel *st*-graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ and construct a new series-parallel *st*-graph representing Y from $\mathcal{G}_1$ and $\mathcal{G}_2$. Once we have performed the last merge, we apply a post-processing procedure that extracts the path decomposition represented by the final DAG.

It may also be necessary to "turn Y_2 around" before constructing Y, because Y_1 and Y_2 are "oriented in opposite directions." Given that Y_1 and Y_2 are represented as series-parallel *st*-graphs, this can be done efficiently if we have an efficient way to flip all edges in $\mathcal{G}_2$. Similar to the stretching of path decompositions, it is expensive to flip all edges in $\mathcal{G}_2$ explicitly. Hence, we need a technique to perform these flips implicitly. After the final graph $\mathcal{G}$ has been constructed, we perform a mop-up procedure that chooses the final direction for every edge. The resulting graph is then input into the procedure for extracting the final path decomposition.

In this section, we describe a representation of series-parallel st-graphs that allows the edges of a graph G to be flipped by updating only $\mathcal{O}(1)$ information stored at the source and sink of G. We also describe an $\mathcal{O}(\text{sort}(N))$ I/O mop-up procedure that chooses the final direction for every edge in G, after all compositions and edge flips have been performed.

Formally, we denote a series-parallel st-graph G as the quadruple $G = (V, E, s, t)$, where s is the source and t is the sink of G. The *flip* of G is the graph $G^\triangleleft = (V, E^\triangleleft, t, s)$, where $E^\triangleleft = \{(w, v) : (v, w) \in E\}$.

Let $\mathcal{G}$ be a pair $\mathcal{G} = (U(G), \gamma)$ representing the graph $G' = (V, E \cup E^\triangleleft)$, where $\gamma : E \cup E^\triangleleft \rightarrow \{\mathsf{blue}, \mathsf{red}\} \times \{\mathsf{blue}, \mathsf{red}\}$ is a coloring of the edges of G and $G^\triangleleft$ with pairs of colors. Note that the function γ can be conveniently represented by storing colors $\gamma((v, w))$ and $\gamma((w, v))$ with edge $\{v, w\} \in U(G)$. Given a pair $c = (c_1, c_2)$ of colors, we define $c^{(1)} = c_1$ and $c^{(2)} = c_2$. This defines two functions $\gamma^{(1)}$ and $\gamma^{(2)}$, where $\gamma^{(1)}(e) = c_1$ and $\gamma^{(2)}(e) = c_2$, for any edge e with $\gamma(e) = (c_1, c_2)$. For a color $c \in \{\mathsf{blue}, \mathsf{red}\}$, we define $\bar{c}$ to be its opposite color; that is, if $c = \mathsf{blue}$, then $\bar{c} = \mathsf{red}$, and vice versa. We say that coloring a vertex v with color c *selects* an edge e incident to v if either $e = (u, v)$ and $\gamma^{(2)}(e) = c$ or $e = (v, w)$ and $\gamma^{(1)}(e) = c$.

A *flippable DAG* is a pair $\mathcal{G} = (U(G), \gamma)$ as described above, with the following properties:

(F1) Let $e = (v, w) \in E$, and let $e^\triangleleft = (w, v) \in E^\triangleleft$ be its flip. Then coloring v or w with a color $c \in \{\mathsf{red}, \mathsf{blue}\}$ selects exactly one of e and $e^\triangleleft$. In particular, $\gamma^{(1)}(e^\triangleleft) = \overline{\gamma^{(2)}(e)}$ and $\gamma^{(2)}(e^\triangleleft) = \overline{\gamma^{(1)}(e)}$.

(F2) Let $E(v, c)$ be the set of edges in G' that are incident to v and selected by coloring v with color c. Then either $E(v, c) \subseteq E$ or $E(v, c) \subseteq E^\triangleleft$.

(F3) Given a vertex $r \in V$, a color $c \in \{\mathsf{red}, \mathsf{blue}\}$, and a spanning tree T of $U(G)$, let $\gamma(r, c, T) : V \rightarrow \{\mathsf{red}, \mathsf{blue}\}$ be a coloring of the vertices in G, defined as follows: Choose r to be the root of tree T, and define $(\gamma(r, c, T))(r) = c$. For every other vertex v with parent $p(v)$ in T, let

$$(\gamma(r, c, T))(v) = \begin{cases} \gamma^{(2)}(p(v), v) & \text{if } (\gamma(r, c, T))(p(v)) = \gamma^{(1)}(p(v), v) \\ \gamma^{(1)}(v, p(v)) & \text{if } (\gamma(r, c, T))(p(v)) = \gamma^{(2)}(v, p(v)) \end{cases}.$$

Then for any two spanning trees T_1 and T_2 of $U(G)$, any two vertices $v, w \in V$, and any two colors c_1 and c_2 so that $(\gamma(v, c_1, T_1))(w) = c_2$, $\gamma(v, c_1, T_1) = \gamma(w, c_2, T_2)$.

Property (F3) implies in particular that $\gamma(v, c, T_1) = \gamma(v, c, T_2)$, for any two spanning trees T_1 and T_2. Thus, we refer to the unique coloring defined by coloring r with color c as $\gamma(r, c)$. Next we show that for any edge $e = (v, w) \in E$, coloring v with color $(\gamma(r, c))(v)$ and coloring w with color $(\gamma(r, c))(w)$ select the same edge from the set $\{e, e^\triangleleft\}$.

Lemma 5 *For any edge $e = (v, w) \in E$ and any coloring $\gamma(r, c)$, coloring vertex v with color $(\gamma(r, c))(v)$ selects the same edge from the set $\{e, e^\triangleleft\}$ as coloring vertex w with color $(\gamma(r, c))(w)$.*

Proof. Assume w.l.o.g. that $r \neq w$. Also assume that coloring v with color $(\gamma(r,c))(v)$ selects edge $e = (v,w)$, and coloring w with color $(\gamma(r,c))(w)$ selects edge $e^{\triangleleft} = (w,v)$; that is, $(\gamma(r,c))(v) = \gamma^{(1)}(e)$ and $(\gamma(r,c))(w) = \gamma^{(1)}(e^{\triangleleft})$. Let T_1 be a spanning tree of $U(G)$ so that $\gamma(r,c) = \gamma(r,c,T_1)$. Note that neither $v = p(w)$ nor $w = p(v)$ in T_1. In the former case, w would be colored with color $\gamma^{(2)}(e) = \overline{\gamma^{(1)}(e^{\triangleleft})}$. In the latter case, v would be colored with color $\gamma^{(2)}(e^{\triangleleft}) = \overline{\gamma^{(1)}(e)}$. Now let T_2 be the tree obtained from T_1 by removing edge $\{w, p(w)\}$ from T_1 and adding edge $\{v, w\}$. Then $(\gamma(r,c,T_1))(v) = (\gamma(r,c,T_2))(v)$, because the path from r to v is the same in T_1 and T_2. Hence, $(\gamma(r,c,T_2))(w) = \gamma^{(2)}(e) = \overline{\gamma^{(1)}(e^{\triangleleft})} = \overline{(\gamma(r,c,T_1))(w)}$. In particular, $\gamma(r,c,T_1) \neq \gamma(r,c,T_2)$, which contradicts Property (F3). □

Now let $E_{r,c} \subseteq E \cup E^{\triangleleft}$ be the set of edges that are selected by coloring the vertices of G as prescribed by coloring $\gamma(r,c)$. By Lemma 5, we can formally define this set as $E_{r,c} = \{(v,w) \in E \cup E^{\triangleleft} : (\gamma(r,c))(v) = \gamma^{(1)}((v,w))\}$.

Lemma 6 *For any vertex $r \in V$ and any color $c \in \{\textsf{red}, \textsf{blue}\}$, either $E_{r,c} = E$ or $E_{r,c} = E^{\triangleleft}$.*

Proof. Consider vertex r and the set E' of edges incident to r. By Property (F2), $E(r,c) \subseteq E$ or $E(r,c) \subseteq E^{\triangleleft}$. Assume w.l.o.g. that $E(r,c) \subseteq E$. Then, by Property (F1), $E(r,\bar{c}) = E' \setminus E(r,c)$. On the other hand, $E(r,\bar{c}) \subseteq E$ or $E(r,\bar{c}) \subseteq E^{\triangleleft}$. Hence, $E(r,c)$ contains *all* edges in E incident to r, and only those edges.

Now let T be a spanning tree of $U(G)$. We prove by induction on the length of the path from r to v in T that for every vertex $v \in V$, $E(v, (\gamma(r,c))(v)$ contains all edges in E incident to v. We have already considered the base case. Therefore, assume that coloring $p(v)$ with color $(\gamma(r,c))(p(v))$ selects all edges in E incident to $p(v)$, and assume w.l.o.g. that $(p(v),v) \in E$. Then edge $(p(v),v)$ is selected by coloring $p(v)$ with color $(\gamma(r,c))(p(v))$. By Lemma 5, coloring vertex v with color $(\gamma(r,c))(v)$ also selects edge $(p(v),v) \in E$. Hence, by Property (F2), $E(v,(\gamma(r,c))(v)) \subseteq E$. Now it follows from Property (F1) again that $E(v,(\gamma(r,c))(v))$ contains *all* edges in E incident to v, and only those edges. This proves that if $E(r,c) \subseteq E$, then $E_{r,c} = E$. A similar argument with the roles of E and $E^{\triangleleft}$ exchanged shows that if $E(r,c) \subseteq E^{\triangleleft}$, then $E_{r,c} = E^{\triangleleft}$. □

The following corollary is an immediate consequence of Lemma 6 and Property (F1).

Corollary 1 *For any vertex $r \in V$ and any color $c \in \{\textsf{red}, \textsf{blue}\}$, $E_{r,c} = E$ and $E_{r,\bar{c}} = E^{\triangleleft}$, or vice versa.*

Given a flippable DAG $\mathcal{G} = (U(G), \gamma)$, a vertex $v \in G$, and a color $c \in \{\textsf{red}, \textsf{blue}\}$, we refer to the process of extracting the graph $G_{v,c} = (V, E_{v,c})$ as *untangling* $\mathcal{G} = (U(G), \gamma)$. Properties (F1) and (F3) immediately suggests an algorithm to untangle $\mathcal{G}$, which is shown in Algorithm 1.

Lemma 7 *A flippable DAG G of size N can be untangled in $\mathcal{O}(\mathrm{sort}(N))$ I/Os.*

Proof. The correctness of Algorithm 1 follows immediately from the above discussion. Computing the spanning tree T of $U(G)$ in Line 1 of Algorithm 1 takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os [14] because G is series-parallel, hence planar, and thus sparse under edge contraction. Before being able process T from the root towards the leaves, all edges in T have to be directed from parents to children. This can be done in $\mathcal{O}(\mathrm{sort}(N))$ I/Os using the Euler-tour technique and list ranking [14]. Given that all edges are directed from parents to children, we can apply the Euler-tour technique and list ranking again to obtain a preorder numbering of the vertices of T, which provides us with a

Algorithm 1 An algorithm to untangle a flippable DAG $\mathcal{G} = (U(G), \gamma)$.

Procedure UNTANGLE

Input: A flippable DAG $\mathcal{G} = (U(G), \gamma)$ representing a series-parallel st-graph G and its flip $G^{\triangleleft}$, a vertex $r \in G$, and a color $c \in \{\mathsf{red}, \mathsf{blue}\}$.

Output: Graph $G_{r,c}$.

1: Compute a spanning tree T of $U(G)$ and choose r to be its root.
2: Process tree T from the root towards the leaves and compute the color $(\gamma(r, c))(v)$, for every vertex$v \in V$.
3: Scan $E \cup E^{\triangleleft}$ and add every edge (v, w) with $\gamma^{(1)}((v, w)) = (\gamma(r, c))(v)$ to $E_{r,c}$.
4: Return the graph $G_{r,c} = (V, E_{r,c})$.

topological ordering of the vertices of T. We sort the vertices of T by their preorder numbers and then use time-forward processing [2] to realize Step 2. Hence, Step 2 takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os. Step 3 can be realized as follows: We sort the vertices by preorder numbers and the edges in $E \cup E^{\triangleleft}$ by the preorder numbers of their sources. Then we scan the sorted edge and vertex sets to extract the edges matching the condition stated in Line 4. This takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os. □

4 I/O-Efficient Greedy Algorithms

In this section, we describe a simple technique to obtain I/O-efficient algorithms for certain graph problems that can be solved using greedy algorithms in internal memory. Using this technique, we obtain simple deterministic $\mathcal{O}(\mathrm{sort}(|V| + |E|))$ I/O algorithms for finding a maximal matching or maximal independent set of an arbitrary graph. The algorithm for finding a maximal matching is used as part of the tree decomposition algorithm presented in Section 5.

Let us define precisely what we mean by "certain" graph problems. A *vertex-labeling algorithm* is an algorithm $\mathcal{A}$ that computes a function $\lambda : V \to X$. We call $\mathcal{A}$ *single-pass* if it computes λ by visiting every vertex $v \in V$ exactly once and assigns a label $\lambda(v)$ to v during this visit. We call $\mathcal{A}$ *local* if it computes $\lambda(v)$ in $\mathcal{O}(\mathrm{sort}(k))$ I/Os from the labels $\lambda(u_1), \ldots, \lambda(u_k)$ of those neighbors $u_1, \ldots, u_k$ of v that have been labeled before visiting v. We call $\mathcal{A}$ *presortable* if there is an $\mathcal{O}(\mathrm{sort}(|V| + |E|))$ I/O algorithm that establishes an order so that $\mathcal{A}$ produces a correct result if it visits the vertices of G in this order. We consider graph problems that can be solved using presortable local single-pass vertex-labeling algorithms.

Theorem 1 *Every graph problem $\mathcal{P}$ that can be solved using a presortable local single-pass vertex-labeling algorithm can be solved in $\mathcal{O}(\mathrm{sort}(|V| + |E|))$ I/Os.*

Proof. We use Algorithm 2 to solve problem $\mathcal{P}$. Let $\mathcal{A}$ be a presortable local single-pass vertex-labeling algorithm that solves problem $\mathcal{P}$. Since $\mathcal{A}$ is local and the ordering $\prec$ is chosen so that algorithm $\mathcal{A}$ solves problem $\mathcal{P}$ correctly if processing the vertices of G in this order, the label $\lambda(v)$ of every vertex $v \in V$ can be computed from the labels $\lambda(u_1), \ldots, \lambda(u_k)$ of its in-neighbors $u_1, \ldots, u_k$ in G'. This establishes the correctness of Algorithm 2.

As algorithm $\mathcal{A}$ is presortable, Line 1 of Algorithm 2 takes $\mathcal{O}(\mathrm{sort}(|V| + |E|))$ I/Os. The edges of G can easily be directed in $\mathcal{O}(\mathrm{sort}(|V| + |E|))$ I/Os, once every edge $\{v, w\}$ has been "informed" about the numbers $\nu(v)$ and $\nu(w)$ assigned to its endpoints v and w in Line 1. To transfer this information to all edges $\{v, w\} \in E$, we sort the vertices in V by their names (not their numbers $\nu(v)$), choose one endpoint for every edge, and sort the edges by their chosen endpoints. Then a

Algorithm 2 A framework for I/O-efficient greedy algorithms.

Procedure IOGREEDY

Input: A graph $G = (V, E)$ and a labeling problem $\mathcal{P}$ that can be solved using a presortable local single-pass vertex-labeling algorithm $\mathcal{A}$.

Output: The labeling $\lambda : V \to X$ of the vertices of G that would be computed by algorithm $\mathcal{A}$.

1: Establish an order $\prec$ of the vertices of graph G so that algorithm $\mathcal{A}$ produces a correct result if it visits the vertices in V in this order, sort the vertices of G in this order, and number the vertices of G in their order of appearance.
2: Replace every edge $\{v, w\} \in E$ by a directed edge (v, w), $v \prec w$; let G' be the resulting DAG.
3: **for** all vertices $v \in V$, in their order of appearance **do**
4: Let $\Gamma^-_{G'}(v) = \{u_1, \ldots, u_k\}$; compute $\lambda(v)$ from $\lambda(u_1), \ldots, \lambda(u_k)$.
5: **end for**

single scan of the vertex and edge sets of G is sufficient to inform every edge about the preorder number of its chosen endpoint. We sort the edges again, this time by their endpoints that were not chosen in the previous pass, and scan the vertex and edge sets again, in order to inform every edge about the preorder number of its second endpoint. At the end of this step, we sort the vertices in V by their numbers $\nu(v)$ and the directed edges (v, w) by the numbers $\nu(v)$ of their source vertices. After the DAG G' has been prepared in this manner, the loop in Lines 3–5 takes $\mathcal{O}(\text{sort}(|E|))$ I/Os: Assuming that every vertex v has labels $\lambda(u_1), \ldots, \lambda(u_k)$ at its disposal, where $\Gamma^-_{G'}(v) = \{u_1, \ldots, u_k\}$, computing labels $\lambda(v)$, $v \in V$, takes $\mathcal{O}(\text{sort}(\sum_{v \in V} |\Gamma^-_{G'}(v)|)) = \mathcal{O}(\text{sort}(|E|))$ I/Os, by the locality of algorithm $\mathcal{A}$. We can use time-forward processing [2] to provide every vertex $v \in V$ with this information. This takes $\mathcal{O}(\text{sort}(|V| + |E|))$ I/Os because only $\mathcal{O}(1)$ information is sent along every edge of G'. $\square$

Next we apply Theorem 1 in order to obtain deterministic $\mathcal{O}(\text{sort}(|V| + |E|))$ I/O algorithms for finding maximal independent sets and maximal matchings.

4.1 Computing a Maximal Independent Set

In order to compute a maximal independent set S of a graph $G = (V, E)$ in internal memory, we can use the following simple algorithm: *Process the vertices in an arbitrary order; when a vertex $v \in V$ is visited, add it to S if none of its neighbors is in S.* Translated into a vertex-labeling problem, we wish to compute the characteristic function $\chi_S : V \to \{0, 1\}$ of S, where $\chi_S(v) = 1$ if $v \in S$, and $\chi_S(v) = 0$ if $v \notin S$. Also note that if S is initially empty, then any neighbor w of v that is visited after v cannot be in S at the time when v is visited. Hence, it is sufficient for v to inspect all its neighbors that are visited before v, in order to decide whether or not v should be added to S. With these modifications, we obtain a vertex-labeling algorithm that is presortable, since the order in which the vertices are visited is unimportant; local, since only previously visited neighbors of v are inspected to decide whether v has to be added to S and a single scan of their labels is sufficient to decide whether at least one of them is in S; and single-pass. The correctness of the algorithm is obvious. Hence, we obtain the following result.

Theorem 2 *Given an undirected graph $G = (V, E)$, a maximal independent set of G can be found in $\mathcal{O}(\text{sort}(|V| + |E|))$ I/Os.*

4.2 Computing a Maximal Matching

Finding a maximal matching is not quite as straightforward as computing a maximal independent set, because it is an edge-labeling problem: Compute the characteristic function $\chi_{\mathcal{M}} : E \to \{0,1\}$ of a maximal matching $\mathcal{M}$. We can easily transform this problem into a vertex-labeling problem because there exists a natural bijection between the maximal matchings of a graph $G = (V, E)$ and the maximal independent sets of the graph $G' = (E, \{\{e, e'\} : \text{edges } e \text{ and } e' \text{ share an endpoint in } G\})$. However, graph G' may have size $\Omega(|V|^2)$, even if $|E| = \mathcal{O}(|V|)$. (As an example, consider a wagon wheel, which is even planar.) Our goal is to construct a subgraph $G'' = (E, E'')$ of G' with $E'' = \mathcal{O}(|E|)$ and describe a vertex-labeling problem of G'' whose solution corresponds to a maximal matching of G. We begin with a description of graph G''.

Given graph $G = (V, E)$, we number the edges of G in their order of appearance in E. Then we define $e_1 < e_2$ if e_1 has a smaller number than e_2 in this numbering. For every vertex $v \in V$ with incident edges $e_1 < \cdots < e_q$, we add edges $\{e_i, e_{i+1}\}$, $1 \le i < q$, to E''. We denote the resulting path from e_1 to e_q in G'' by P_v. Every vertex $e \in G''$ has at most two in-edges and at most two out-edges, one in-edge and one out-edge per endpoint of edge $e \in G$. Hence, $|E''| = \mathcal{O}(|E|)$, as desired. We have to describe a vertex-labeling problem of G'' whose solution corresponds to a maximal matching of G and which can be solved by a presortable local single-pass algorithm.

Every vertex $e \in G''$ is contained in two paths P_v and P_w in G'', one per endpoint of edge $e = \{v, w\} \in G$. A subset $\mathcal{M} \subseteq E$ is a maximal matching of G if and only if the characteristic function $\chi_{\mathcal{M}} : E \to \{0,1\}$ of $\mathcal{M}$ has the following two properties:

(M1) For every path P_v, $v \in V$, $\sum_{e \in P_v} \chi_{\mathcal{M}}(e) \le 1$.

(M2) For every edge $e = \{v, w\} \in G$, $\sum_{e \in P_v \cup P_w} \chi_{\mathcal{M}}(e) \ge 1$.

Property (M1) expresses the fact that $\mathcal{M}$ is a matching, i.e., that every vertex has at most one incident edge in $\mathcal{M}$. Property (M2) expresses the maximality of $\mathcal{M}$, i.e., the fact that every edge not in $\mathcal{M}$ shares an endpoint with an edge in $\mathcal{M}$. We compute function $\chi_{\mathcal{M}}$ using Algorithm 3. This algorithm is presortable, as it uses the ordering of the edges in E used to construct G''; it is obviously single-pass; and its localilty follows from the way labels $\lambda(e)$ are computed. All that remains to be shown is that the algorithm is correct.

Theorem 3 *Given an undirected graph $G = (V, E)$, a maximal matching $\mathcal{M} \subseteq E$ of G can be computed in $\mathcal{O}(\mathrm{sort}(|V| + |E|))$ I/Os.*

Proof. In order to prove the correctness of Algorithm 3, we have to show that the labeling $\chi_{\mathcal{M}}(e)$ constructed by the algorithm has Properties (M1) and (M2). Since the algorithm processes the vertices of G'' sorted by the "$<$" relation, it is easily verified that $\sigma(P_v, e) = \sum\{\chi_{\mathcal{M}}(e') : e' \in P_v \text{ and } e' < e\}$ and $\sigma(P_w, e) = \sum\{\chi_{\mathcal{M}}(e') : e' \in P_w \text{ and } e' < e\}$. This immediately implies that labeling $\chi_{\mathcal{M}}$ has Property (M2) because the algorithm sets $\chi_{\mathcal{M}}(e) = 1$ unless $\sigma(P_v, e) + \sigma(P_w, e) \ge 1$. In both cases, Property (M2) holds. Property (M1) holds because $\chi_{\mathcal{M}}(e) = 1$ implies that $\sigma(P_v, e_v) = 0$ and $\sigma(P_v, e') = 1$, for all $e' \in P_v$, $e \le e'$. Hence, $\chi_{\mathcal{M}}(e') = 0$, for all $e' \in P_v \setminus \{e\}$. The same argument shows that $\chi_{\mathcal{M}}(e') = 0$, for all $e' \in P_w \setminus \{e\}$, if $\chi_{\mathcal{M}}(e) = 1$. The I/O-complexity follows from Theorem 1. □

Algorithm 3 Computing a maximal matching.

Procedure MaximalMatching

Input: An undirected graph $G = (V, E)$.
Output: A maximal matching $\mathcal{M} \subseteq E$ of G.

1: Construct a graph $G'' = (E, E'')$ as described in the text and label every edge $\{e, e'\} \in E''$ with the name of the endpoint shared by edges e and e' in G (i.e., with the vertex v so that $\{e, e'\} \in P_v$).
2: Sort the vertex set E of G'' by the relation "$<$" defined on the edges of G.
3: **for** every vertex $e \in E$, in their order of appearance **do**
4: Compute a label $\lambda(e) = (\chi_{\mathcal{M}}(e), \sigma(P_v, e), \sigma(P_w, e))$ of vertex e, where $\sigma(P, e) = \sum\{\chi_{\mathcal{M}}(e') : e' \in P \text{ and } e' \leq e\}$: Let $e = \{v, w\}$, and let e_v and e_w be the neighbors of e on paths P_v and P_w so that $e_v < e$ and $e_w < e$. If e_v does not exist, assume that e_v is a dummy vertex with $\lambda(e_v) = (0, 0, 0)$. The same assumption holds for e_w. Then let $\chi_{\mathcal{M}}(e) = 1$ if $\sigma(P_v, e_v) + \sigma(P_w, e_w) = 0$, and $\chi_{\mathcal{M}}(e) = 0$ otherwise. Let $\sigma(P_v, e) = \sigma(P_v, e_v) + \chi_{\mathcal{M}}(e)$ and $\sigma(P_w, e) = \sigma(P_w, e_w) + \chi_{\mathcal{M}}(e)$.
5: **end for**
6: Scan E and extract label $\chi_{\mathcal{M}}(e)$ from label $\lambda(e)$, for every edge $e \in E$.

5 Computing a Tree Decomposition of Width k

Our algorithm for computing a tree decomposition of width at most k for a graph $G = (V, E)$ of treewidth at most k is based on the algorithm of [8]. We first recall this algorithm and then show how to perform each of its steps in $\mathcal{O}(\text{sort}(N))$ I/Os. Before describing the algorithm, however, we need to introduce some terminology. For some threshold d to be defined later, a vertex is said to be of *low degree* if its degree is at most d; otherwise, the vertex is of *high degree*. A vertex is *friendly* if it is of low degree and has at least one neighbor of low degree. The *improved graph* G' of a graph G is obtained by adding an edge $\{v, w\}$ to G, for every pair of vertices v and w that have at least $k + 1$ common neighbors of low degree in G. If the treewidth of G is at most k, the treewidth of G' cannot be greater than k. A vertex of G is *I-simplicial* if it is simplicial in G', of low degree in G, and not friendly in G (i.e., all its neighbors are of high degree).

The algorithm of [8] is based on the following fact, which is proved in [8]: For an appropriately chosen d, a graph G of treewidth at most k contains either a sufficient number of friendly vertices or a sufficient number of I-simplicial vertices, where "a sufficient number" means "a constant fraction." If there is a suffficient number of friendly vertices, a maximal matching of G contains at least αN edges, for some constant $0 < \alpha < 1$. Hence, a graph G' of treewidth at most k and with at most $(1 - \alpha)N$ vertices can be obtained by contracting the edges in a maximal matching of G. Given a tree decomposition $\mathcal{D}'$ of width at most k for G', which can be computed recursively, a tree decomposition $\mathcal{D}''$ of width at most $2k + 1$ for G can be obtained by re-expanding the edges in the matching. In order to obtain a tree decomposition $\mathcal{D}$ of width at most k for G, the algorithm of [8] applies an algorithm of [11] to G and $\mathcal{D}''$. If the number of friendly vertices is too small, G contains at least βN I-simplicial vertices, for some constant $0 < \beta < 1$. Let G^* be the graph obtained by removing all I-simplicial vertices of G from the improved graph of G. Graph G^* has treewidth at most k and at most $(1 - \beta)N$ vertices, so that a tree decomposition of width at most k for G^* can be computed recursively. Since the neighborhood of every I-simplicial vertex v is a clique in G^*, there has to be a node $\mathsf{i}_v \in \mathcal{D}^*$ so that $\Gamma_G(v) \subseteq X_{\mathsf{i}_v}$. Hence, a tree decomposition $\mathcal{D}$ of width at most k for G can be obtained from $\mathcal{D}^*$ by identifying such a node i_v, for every I-simplicial vertex v, and adding a new node j_v with $X_{\mathsf{j}_v} = \Gamma_G(v) \cup \{v\}$ and an edge $\{\mathsf{i}_v, \mathsf{j}_v\}$ to $\mathcal{D}^*$.

Algorithm 4 Computing a tree decomposition.

Procedure TreeDecomposition

Input: A graph $G = (V, E)$ and a constant $k \in \mathbb{N}$.
Output: A tree decomposition $\mathcal{D} = (\mathcal{X}, T)$ of width at most k for G, or the answer that the treewidth of G is greater than k.

if $|G| \leq M$ **then**
  Use the algorithm of [8] to compute a tree decomposition of G.
else
  if $|E| > k|V| - \frac{1}{2}k(k+1)$ **then**
    Output that the treewidth of G is greater than k.
  else
    if there are at least $|V|/(4k^2 + 12k + 16)$ friendly vertices in G **then**
      Find a maximal matching $\mathcal{M} \subseteq E$ of G. (Theorem 3)
      Contract the edges in $\mathcal{M}$ and call the resulting graph $G' = (V', E')$.
      Recursively compute a tree decomposition $\mathcal{D}' = (\mathcal{X}', T')$ of width at most k and size $\mathcal{O}(N)$ for G'.
      if the treewidth of G' is greater than k **then**
        Output that the treewidth of G is greater than k.
      else
        Compute a tree decomposition $\mathcal{D}'' = (\mathcal{X}'', T')$ of G by expanding the edges of $\mathcal{M}$. The width of $\mathcal{D}''$ is at most $2k+1$.
        Apply Algorithm 5 to G and $\mathcal{D}''$, in order to compute a tree decomposition $\mathcal{D} = (\mathcal{X}, T)$ of width at most k and size $\mathcal{O}(N)$ for G. (Theorem 6)
      end if
    else
      Compute the improved graph G' of G, put all I-simplicial vertices into a set SL, and compute a graph $G^* = G' - SL$. (Lemma 8)
      if there is an I-simplicial vertex of degree at least $k+1$ (Lemma 8) **then**
        Output that the treewidth of G is greater than k.
      else
        if $|SL| < c_2|V|$ **then**
          Output that the treewidth of G is greater than k.
        else
          Recursively compute a tree decomposition $\mathcal{D}^* = (\mathcal{X}^*, T^*)$ of width at most k and size $\mathcal{O}(N)$ for G^*.
          if the treewidth of G^* is greater than k **then**
            Output that the treewidth of G is greater than k.
          else
            For each $v \in SL$, find a node $\mathsf{i}_v \in I^*$ such that $\Gamma_G(v) \subseteq X^*_{\mathsf{i}_v}$, add a node j_v to I^*, make it adjacent to i_v, and let $X^*_{\mathsf{j}_v} = \Gamma_G(v) \cup \{v\}$; let $\mathcal{D} = (\mathcal{X}, T)$ be the resulting tree decomposition of width at most k. (Lemma 9)
          end if
        end if
      end if
    end if
  end if
end if

The details of the algorithm are presented in Algorithm 4. The maximal degree d for low-degree vertices is defined using two constants $0 < c_1, c_2 < 1$, which are chosen arbitrarily but satisfy the following condition:

$$c_2 = \frac{1}{4k^2 + 12k + 16} - \frac{c_1 k^2(k+1)}{2}.$$

In particular, $d = \max(k^2 + 4k + 4, \lceil 2k/c1 \rceil)$. Then c_1 is an upper bound on the fraction of high-degree vertices in G; constant c_2 provides a lower bound on the fraction of vertices that are removed from G before calling the algorithm recursively. The correctness of Algorithm 4 is shown in [8]. The lemmas cited in parentheses after each step in the algorithm show how to realize this particular step in $\mathcal{O}(\text{sort}(N))$ I/Os. All other steps take $\mathcal{O}(\text{sort}(N))$ I/Os, using sorting and scanning, and are fairly straightforward. In [8], it is shown that the subgraphs G' and G^* passed to recursive calls of the algorithm in Lines 10 and 25 have size at most $(1 - c_2)N$. Hence, the I/O-complexity of the algorithm is $\mathcal{I}(N) = \mathcal{I}((1 - c_2)N) + \mathcal{O}(\text{sort}(N)) = \mathcal{O}(\text{sort}(N))$, and we obtain the following result.

Theorem 4 *Given a graph $G = (V, E)$ with N vertices and a constant $k \in \mathbb{N}$, Algorithm 4 takes $\mathcal{O}(\text{sort}(N))$ I/Os to decide whether G has treewidth at most k and, if so, compute a tree decomposition $\mathcal{D} = (\mathcal{X}, T)$ of width at most k and size $\mathcal{O}(N)$ for G.*

The next two lemmas show how to realize Steps 18, 19, and 29 of Algorithm 4. The algorithm used to realize Step 15 is discussed in Section 6.

Lemma 8 *The improved graph $G' = (V, E')$ of a graph $G = (V, E)$ as well as all I-simplicial vertices of G can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os.*

Proof. The proof is a straightforward adaptation of the internal memory algorithm for this problem, presented in [8]. We include it for completeness.

First we identify all vertices of low degree. In particular, we compute the adjacency lists of all vertices of G and then extract all vertices of low degree. The latter can be done in a single scan of the computed adjacency lists. The former can be done in $\mathcal{O}(\text{sort}(N))$ I/Os by scanning the edge set of G, creating two directed edges (v, w) and (w, v), for every edge $\{v, w\} \in E$, and sorting the resulting list of directed edges.

Now assume that there exists an ordering $<$ defined on the vertices of G (e.g., the natural order defined by a numbering of the vertices). For each low-degree vertex u with neighborhood $\Gamma_G(u)$, we create a list $L(u) = \{(v, w, u) : \{v, w\} \subseteq \Gamma_G(u) \wedge v < w\}$. From the edge set of G, we create a list $L' = \{(v, w, —) : \{v, w\} \in E \wedge v < w\}$. Let L be the concatenation of list L' and lists $L(v)$. Note that $|L| = \mathcal{O}(N)$ because for fixed k, every low-degree vertex has constant degree. We sort L lexicographically, where the symbol "—" is assumed to be less than any vertex of G. In order to obtain the edge set E' of the improved graph, we add an edge $\{v, w\}$ to E, for every pair of vertices v and w such that there is no triple $(v, w, —)$ in L and there are at least $k + 1$ triples $(v, w, u_1), \ldots, (v, w, u_{k'})$ in L. For every triple $(v, w, u) \in L$, we add an entry (u, v, w) to a new list S if the entry $(v, w, —)$ is in L or there are at least $k+1$ triples $(v, w, u_1), \ldots, (v, w, u_{k'})$ in L. This computation can be carried out in a single scan of list L.

List S contains the edges in G' that connect the neighbors of every low-degree vertex in G. We sort S lexicographically. Since the neighborhood of every low-degree vertex is of constant size, it takes a single scan of list S to identify those low-degree vertices whose neighborhoods in G' are cliques; we add all these vertices to the list SL of I-simplicial vertices.

As we sort and scan lists of linear size a constant number of times, this whole computation can be carried out in $\mathcal{O}(\text{sort}(N))$ I/Os. $\square$

Given lists S and SL, as described in the proof of Lemma 8, we can decide whether there exists an I-simplicial vertex of degree at least $k+1$ by scanning these two lists (Step 19).

Lemma 9 *Given a tree decomposition $\mathcal{D}^* = (\mathcal{X}^*, T^*)$ of width at most k and size $\mathcal{O}(N)$ for the graph $G^* = G' - SL$, a tree decomposition $\mathcal{D} = (\mathcal{X}, T)$ of width at most k and size $\mathcal{O}(N)$ for the improved graph G' of G, and thus for G, can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os.*

Proof. Again, the computation described in [8] can easily be carried out in an I/O-efficient manner. Let $T^* = (I^*, F^*)$. For every I-simplicial vertex u with $\Gamma_G(u) = \{v_1, \dots, v_l\}$, $v_1 < v_2 < \cdots < v_l$, we create a tuple $(v_1, \dots, v_l, u)$ and add it to a list L. For every node $\mathsf{i} \in I^*$ and every subset $\{x_1, \dots, x_l\} \subseteq X_{\mathsf{i}}$, we create a tuple $(x_1, \dots, x_l, \mathsf{i})$, $x_1 < x_2 < \cdots < x_l$, and add it to L. The resulting list has size $\mathcal{O}(N)$: there are at most N I-simplicial vertices; the tree T^* has $\mathcal{O}(N)$ nodes; and $|X_{\mathsf{i}}| = \mathcal{O}(1)$, for all $\mathsf{i} \in I^*$. We sort list L lexicographically, where we assume that $\mathsf{i} < v$, for every node $\mathsf{i} \in I^*$ and every vertex $v \in V$. As a result, every set of I-simplicial vertices with the same neighborhood is preceded by a tuple corresponding to a tree node i whose associated set X_{i} contains this neighborhood. We scan list L to create another list S containing pairs (i_v, v), where v is an I-simplicial vertex and i_v is a node of T^* such that $\Gamma_G(v) \subseteq X_{\mathsf{i}_v}$. For every I-simplicial vertex v, we add a new node j_v to I^*, a set $X^*_{\mathsf{j}_v} = \Gamma_G(v) \cup \{v\}$ to $\mathcal{X}^*$, and an edge $\{\mathsf{i}_v, \mathsf{j}_v\}$ to T^*; the result is the desired tree decomposition $\mathcal{D} = (\mathcal{X}, T)$.

In the course of this procedure, we sort and scan lists of linear size a constant number of times. Hence, the whole computation takes $\mathcal{O}(\text{sort}(N))$ I/Os. $\square$

6 Improving the Tree Decomposition

In this section, we present an algorithm that solves the following problem: Given a graph $G = (V, E)$, a tree decomposition $\mathcal{D} = (\mathcal{X}, T)$ of width k for G, and a constant $\ell < k$, test whether the treewidth of G is at most ℓ and, if so, compute a tree decomposition $\mathcal{E} = (\mathcal{Y}, U)$ of width at most ℓ for G. This algorithm is used in Step 15 of Algorithm 4. It is based on the internal memory algorithm of [11] and consists of two phases. The first phase applies dynamic programming to the given tree decomposition $\mathcal{D}$, in order to decide whether graph G has treewidth at most ℓ. This phase is a straightforward simulation of the testing phase of the internal memory algorithm using the time-forward processing technique of [14]. The second phase uses the information produced by the first phase to construct a tree decomposition of width at most ℓ for graph G. The details of this phase differ considerably from those of the internal memory algorithm, as we have to avoid the random memory access pattern of that algorithm. Our algorithm for this phase carefully combines the time-forward processing technique and flippable DAGs to achieve this.

In Section 6.1, we recall the necessary details of the testing phase of the algorithm of [11]. This lays the foundation for the description of our algorithm for the construction phase, which we present in Section 6.2. In order for the testing phase to produce information that can be used by the construction phase, we describe an augmented version of the testing phase at the beginning of Section 6.2.

6.1 Bounded Treewidth Testing

The algorithm we describe assumes that the given tree decomposition $\mathcal{D} = (\mathcal{X}, T)$ is nice. In Section 7, we describe an $\mathcal{O}(\text{sort}(N))$ I/O algorithm for transforming any tree decomposition into an equivalent nice tree decomposition; hence, this is not a restriction. Recall that the nodes in a nice tree decomposition can be of four different types: *start*, *join*, *forget*, and *introduce* nodes. The algorithm computes a *full set of characteristics* $FS(\mathsf{i})$, for every node $\mathsf{i} \in T$. This set contains constant-size descriptions (characteristics) of a constant number of tree decompositions of G_{i} that are optimal in the sense that for every tree decomposition $\mathcal{F}$ of width at most ℓ for G, there exists a tree decomposition $\mathcal{F}'$ whose characteristic is in $FS(\mathsf{i})$ and which is "better" than $\mathcal{F}$ in a sense formalized in [11]. It follows immediately that G has treewidth at most ℓ if and only if the set $FS(\mathsf{r})$ computed for the root r of T is non-empty. To compute these full sets of characteristics, the algorithm processes T from the leaves towards the root. For every leaf i, set $FS(\mathsf{i})$ is computed in a brute-force manner. For every internal node i, this set is computed from the sets computed for its childen.

To describe the algorithm rigorously, we need some more terminology. A tree decomposition (path decomposition) of the subgraph G_{i} rooted at node i is called a *partial* tree decomposition (path decomposition) rooted at node i. Given a partial path decomposition $Y = (Y_1, \dots, Y_r)$ rooted at node i, the *restriction* of Y is the path decomposition $Z = (Z_1, \dots, Z_r)$ of the graph $G[X_{\mathsf{i}}]$, where $Z_j = Y_j \cap X_{\mathsf{i}}$, for $1 \leq j \leq r$. The *interval model* of Y is the list $Z' = (Z_{q_1}, \dots, Z_{q_t})$ obtained by removing consecutive duplicates from Z; that is, $Z_{q_s} \neq Z_{q_{s+1}}$, for $1 \leq s < t$, and $Z_j = Z_{q_s}$, for $q_s \leq j < q_{s+1}$. (Assume that $q_{t+1} = r + 1$.) Given a partial path decomposition $Y = (Y_1, \dots, Y_r)$ with interval model $Z = (Z_{q_1}, \dots, Z_{q_t})$, the *list representation* of Y is the pair $(Z, [Y])$, where $[Y] = \left(Y^{(1)}, Y^{(2)}, \dots, Y^{(t)}\right)$ and $Y^{(s)} = (Y_{q_s}, \dots, Y_{q_{s+1}-1})$, for $1 \leq s \leq t$. Given the list representation $(Z, [Y])$ of Y, the *list* of Y is defined as $[y] = \left(y^{(1)}, y^{(2)}, \dots, y^{(t)}\right)$, where $y^{(s)} = (|Y_{q_s}|, \dots, |Y_{q_{s+1}-1}|)$, for $1 \leq s \leq t$. The *characteristic* of Y is the pair $C(Y) = (Z, \tau[y])$, where $\tau[y]$ is the typical list of $[y]$.

A tree decomposition is *non-trivial* if for any two adjacent nodes i and j in the tree, $X_{\mathsf{i}} \neq X_{\mathsf{j}}$. A leaf i of a tree decomposition is *maximal* if there is a vertex $v \in X_{\mathsf{i}}$ that is not contained in any other set X_{j}. In particular, a leaf i is maximal if and only if there is a vertex $v \in X_{\mathsf{i}}$ that is not contained in X_{j}, where j is the only neighbor of i in the tree. A tree decomposition is *minimal* if it is non-trivial and all its leaves are maximal. Intuitively, the number of nodes in a minimal tree decomposition cannot be reduced by pruning redundant leaves and merging neighbors that store the same sets. It is is easily verified that each graph of treewidth ℓ has a minimal tree decomposition of width ℓ; hence, we can restrict our attention to minimal tree decompositions when trying to find a tree decomposition of width ℓ for G. The following lemmas are useful in bounding the amount of computation performed per node of $\mathcal{T}$ as well as the amount of data sent along the edges of $\mathcal{T}$, which is the factor that determines the I/O-complexity of our algorithm.

Lemma 10 (Bodlaender/Kloks [11]) *The number of nodes in a minimal tree decomposition of an N-vertex graph is at most $(2N-1)^2$.*

Lemma 11 (Bodlaender/Kloks [11]) *The number of nodes in a minimal path decomposition of an N-vertex graph is at most $2N + 1$.*

The *restriction* of a partial tree decomposition $\mathcal{E}$ rooted at node i is defined in a manner analogous to the restriction of a partial path decomposition. The *trunk* of a partial tree decomposition is the tree obtained from its restriction by recursively removing all non-maximal leaves and replacing each path whose internal vertices have degree 2 by a single edge.

Lemma 12 (Bodlaender/Kloks [11]) *The size of the trunk of a partial tree decomposition is at most $2k$.*

Every edge e in the trunk represents a path of edges in the tree decomposition. Note that such a path is a path decomposition Y_e of the graph induced by its nodes. The *filled trunk* is the tree obtained by replacing every edge e in the trunk with its corresponding path decomposition Y_e. Let Z_e be the interval model of the path decomposition Y_e, for every edge $e \in \mathcal{T}$; then the *tree model* of $\mathcal{E}$ is the pair $(\mathcal{T}, (Z_e)_{e\in\mathcal{T}})$, where $\mathcal{T}$ is the trunk of $\mathcal{E}$. The *trunk representation* of $\mathcal{E}$ is the triple $(\mathcal{T}, (Z_e)_{e\in\mathcal{T}}, ([Y_e])_{e\in\mathcal{T}})$, where $(Z_e, [Y_e])$ is the list representation of Y_e, for every $e \in \mathcal{T}$. Finally, the *characteristic* of $\mathcal{E}$ is the triple $(\mathcal{T}, (Z_e)_{e\in\mathcal{T}}, (\tau[y_e])_{e\in\mathcal{T}})$, where $(Z_e, \tau[y_e])$ is the characteristic of path decomposition $\mathcal{Y}_e$, for every $e \in \mathcal{T}$.

Lemma 13 (Bodlaender/Kloks [11]) *The characteristic of a partial tree decomposition has constant size.*

The full set of characteristics $FS(\mathsf{i})$ of a node i in $\mathcal{D}$ has the following property: For every characteristic C in $FS(\mathsf{i})$, there is a partial tree decomposition rooted at i that has width at most ℓ and characteristic C. For every partial tree decomposition $\mathcal{F}$ rooted at i that has width at most ℓ and whose trunk representation is $(\mathcal{T}, (Z_e)_{e\in\mathcal{T}}, ([Y_e])_{e\in\mathcal{T}})$, there exists a partial tree decomposition $\mathcal{F}'$ rooted at i and with trunk representation $(\mathcal{T}, (Z_e)_{e\in\mathcal{T}}, ([Y'_e]_{e\in\mathcal{T}})$ whose characteristic is in $FS(\mathsf{i})$ and so that for every edge e, there are extensions $[y''_e]$ and $[y'''_e]$ of lists $[y_e]$ and $[y'_e]$ so that $[y''_e] \leq [y'''_e]$, where $(Z_e, [y_e])$ and $(Z_e, [y'_e])$ are the lists of path decompositions Y_e and Y'_e, respectively.

Lemma 14 (Bodlaender/Kloks [11]) *For every node i of tree decomposition $\mathcal{D} = (\mathcal{X}, T)$, there exists a full set of characteristics of constant size.*

The linear-time algorithm of [11] for testing whether a given graph with nice tree decomposition $\mathcal{D} = (\mathcal{X}, T)$ has treewidth at most ℓ processes the tree T bottom-up and computes the full set of characteristics for every node from the sets computed for its children; for a leaf i, the full set of characteristics is constructed by generating all minimal tree decompositions of G_{i}, testing each of them whether it has width at most ℓ, and adding its characteristic to $FS(\mathsf{i})$ if this is the case. As there are only a constant number of minimal tree decompositions to be tested, this takes constant time. In external memory, the algorithm takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os using the time-forward processing technique of [2, 14], since we send only a constant amount of information from each node to its parent (Lemmas 13 and 14). Thus, we obtain the following theorem.

Theorem 5 *Given a graph $G = (V, E)$, two constants k and ℓ, and a tree decomposition $\mathcal{D} = (\mathcal{X}, T)$ of width k for G, it takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os to decide whether G has treewidth at most ℓ.*

Proof. This follows from [11, 14] and Lemmas 13 and 14. □

In order to provide a basis for the description of the construction phase in Section 6.2, we spend the rest of this section to describe in more detail how the full sets of characteristics are computed for the four different node types in a nice tree decomposition. The reader who is familiar with this procedure may wish to skip to Section 6.2.

Start node: Generate all minimal tree decompositions $\mathcal{F} = (\mathcal{W}, H)$ of width at most ℓ for the graph $G[X_{\mathsf{i}}]$ and put their characteristics into $FS(\mathsf{i})$. Compute the trunk $\mathcal{T}$ of $\mathcal{F}$ by removing all nodes of degree 2 from H. For every trunk edge e, the interval model Z_e is the sequence Y_e because $\mathcal{F}$ is minimal. The typical sequence for the i-th interval of Y_e consists of the single element $\left|Y_e^{(i)}\right|$. This implies that $\tau[y_e] = [y_e]$.

Join node: If i is a join node with children j and k, compute set $FS(\mathsf{i})$ from sets $FS(\mathsf{j})$ and $FS(\mathsf{k})$ as follows: First observe that $X_{\mathsf{i}} = X_{\mathsf{j}} = X_{\mathsf{k}}$. Consider all pairs of characteristics in $FS(\mathsf{j}) \times FS(\mathsf{k})$ with the same tree model. Such a pair consists of two characteristics $(\mathcal{T}, (Z_e)_{e\in\mathcal{T}}, (\tau[a_e])_{e\in\mathcal{T}}) \in FS(\mathsf{j})$ and $(\mathcal{T}, (Z_e)_{e\in\mathcal{T}}, (\tau[b_e])_{e\in\mathcal{T}}) \in FS(\mathsf{k})$. For each edge $e \in \mathcal{T}$, compute a list $[a_e^*] = \left(\tau\left(a_e^{(1)}\right) - \left|Z_e^{(1)}\right|, \tau\left(a_e^{(2)}\right) - \left|Z_e^{(2)}\right|, \ldots\right)$. Then compute the typical lists $\tau[c_e]$ of all lists $[c_e] \in [a_e^*] \oplus \tau[b_e]$ with $\max[c_e] \le \ell + 1$ and add the characteristic $(\mathcal{T}, (Z_e)_{e\in\mathcal{T}}, (\tau[c_e])_{e\in\mathcal{T}})$ to $FS(\mathsf{i})$.

Forget node: If i is a forget node with child j and $X_{\mathsf{i}} = X_{\mathsf{j}} \setminus \{x\}$, compute set $FS(\mathsf{i})$ from set $FS(\mathsf{j})$ as follows: For each characteristic $(\mathcal{T}, (Z_e)_{e\in\mathcal{T}}, (\tau[y_e])_{e\in\mathcal{T}}) \in FS(\mathsf{j})$, add one characteristic $(\mathcal{T}^*, (Z_e^*)_{e\in\mathcal{T}^*}, (\tau[y_e^*])_{e\in\mathcal{T}^*})$ to $FS(\mathsf{i})$. To obtain this charactersitic, remove vertex x from all sets $Z_e^{(q)}$ and compute the new trunk $\mathcal{T}^*$; for every edge $e \in \mathcal{T}^*$, remove repetitions from the interval model Z_e and define Z_e^* to be the resulting interval model; finally, change the typical list $\tau[y_e]$ into $\tau[y_e^*]$ as described next.

Consider an interval model $Z_e = \left(Z_e^{(1)}, \ldots, Z_e^{(s)}\right)$. If vertex x is contained in some sets of Z_e, then these sets have to be consecutive; that is, x is contained in sets $Z_e^{(a)}, \ldots, Z_e^{(b)}$, for two indices $1 \le a \le b \le s$. As all sets in Z_e are different, the removal of x can cause at most two pairs of consecutive sets to become equal: $Z_e^{(a-1)} = Z_e^{(a)} \setminus \{x\}$ and $Z_e^{(b)} \setminus \{x\} = Z_e^{(b+1)}$. Depending on which case applies, set $Z_e^{(a)}$, set $Z_e^{(b)}$, or both are removed from Z_e to obtain Z_e^*. Hence, there are four different cases to consider for the computation of typical list $\tau[y_e]$:

1. If $|Z_e^*| = |Z_e|$, let $\tau[y_e^*] = \tau[y_e]$.
2. If $Z_e^{(a-1)} = Z_e^{(a)} \setminus \{x\}$, but $Z_e^{(b)} \setminus \{x\} \ne Z + e^{(b+1)}$, let
$$\tau^* = \tau\left(\tau\left(y_e^{(a-1)}\right) \circ \tau\left(y_e^{(a)}\right)\right)$$
and
$$\tau[y_e^*] = \left(\tau\left(y_e^{(1)}\right), \ldots, \tau\left(y_e^{(a-2)}\right), \tau^*, \tau\left(y_e^{(a+1)}\right), \ldots, \tau\left(y_e^{(s)}\right)\right).$$
3. If $Z_e^{(b)} \setminus \{x\} = Z_e^{(b+1)}$, but $Z_e^{(a-1)} \ne Z_e^{(a)} \setminus \{x\}$, compute $\tau[y_e^*]$ from $\tau[y_e]$ similar to Case 2.
4. If $Z_e^{(a-1)} = Z_e^{(a)} \setminus \{x\}$ and $Z_e^{(b)} \setminus \{x\} = Z_e^{(b+1)}$, let
$$\tau_1 = \tau\left(\tau\left(y_e^{(a-1)}\right) \circ \tau\left(y_e^{(a)}\right)\right) \text{ and } \tau_2 = \tau\left(\tau\left(y_e^{(b)}\right) \circ \tau\left(y_e^{(b+1)}\right)\right),$$

and

$$\tau[y_e^*] = \left(\tau\left(y_e^{(1)}\right), \ldots, \tau\left(y_e^{(a-2)}\right), \tau_1, \right.$$
$$\left. \tau\left(y_e^{(a+1)}\right), \ldots, \tau\left(y_e^{(b-1)}\right), \tau_2, \tau\left(y_e^{(b+2)}\right), \ldots, \tau\left(y_e^{(s)}\right)\right).$$

In the last case, if $a = b$, compute a single typical list $\tau\left(\tau\left(y_e^{(a-1)}\right) \circ \tau\left(y_e^{(a)}\right) \circ \tau\left(y_e^{(a+1)}\right)\right)$ and insert it at the right position into $\tau[y_e^*]$.

Introduce node: If i is an introduce node with child j so that $X_\mathsf{i} = X_\mathsf{j} \cup \{x\}$, every characteristic in $FS(\mathsf{i})$ is computed from a characteristic in $FS(\mathsf{j})$ and a "matching" characteristic of a minimal tree decomposition of $G[X_\mathsf{i}]$. For every minimal tree decomposition $\mathcal{F}^*$ of $G[X_\mathsf{i}]$, remove vertex x from all sets in the tree decomposition; the result is a tree decomposition $\mathcal{F}'$ for $G[X_\mathsf{j}]$. For every characteristic $C(\mathcal{F})$ in $FS(\mathsf{j})$ that has the same tree model as $\mathcal{F}'$, compute a set of characteristics $C(\mathcal{F}^\circ)$ so that $\mathcal{F}^\circ$ is minimal and can be obtained from $\mathcal{F}$ or $\mathcal{F}^*$ by augmenting either of the two tree decompositions appropriately. Add every characteristic in this set so that $\mathcal{F}^\circ$ has width at most ℓ to $FS(\mathsf{i})$. The details of this construction are as follows:

Let $\mathcal{F}^*$ be a tree-decompostion of $G[X_\mathsf{i}]$ with characteristic $C(\mathcal{F}^*) = (\mathcal{T}^*, (Z_e^*)_{e\in\mathcal{T}}, (\tau[y_e^*])_{e\in\mathcal{T}})$, let $\mathcal{F}'$ be the tree decomposition of $G[X_\mathsf{j}]$ obtained by removing vertex x from all sets X_i in $\mathcal{F}^*$, let $C(\mathcal{F}') = (\mathcal{T}, (Z_e)_{e\in\mathcal{T}}, (\tau[y'_e])_{e\in\mathcal{T}})$ be the characteristic of $\mathcal{F}'$, and let $\mathcal{F}$ be a tree decomposition of G_j with characteristic $C(\mathcal{F}) = (\mathcal{T}, (Z_e)_{e\in\mathcal{T}}, (\tau[y_e])_{e\in\mathcal{T}}) \in FS(\mathsf{j})$. Add all those characteristics $C(\mathcal{F}^\circ) = (\mathcal{T}^*, (Z_e^*)_{e\in\mathcal{T}^*}, (\tau[y_e^\circ])_{e\in\mathcal{T}^*})$ to $FS(\mathsf{i})$ that can be derived from $C(\mathcal{F})$ and $C(\mathcal{F}^*)$ using the following rules and satisfy $\max[y_e^\circ] \le \ell + 1$, for every edge e of $\mathcal{T}^*$. Since $\mathcal{T}^*$ and Z_e^*, $e \in \mathcal{T}^*$, are fixed, we only describe how to derive typical lists $\tau[y_e^\circ]$, for all $e \in \mathcal{T}^*$:

1. If $\mathcal{T} = \mathcal{T}^*$, there are a number of different typical lists $\tau[y_e^\circ]$, for each edge $e \in \mathcal{T}^*$, that can be valid for edge e. Every combination of the possible choices of one list per edge creates another characteristic that is added to $FS(\mathsf{i})$. To determine the set of choices for each edge $e \in \mathcal{T}^*$, let $Z_e = \left(Z_e^{(1)}, \ldots, Z_e^{(s)}\right)$ be the interval model for edge e in $\mathcal{F}$ (and $\mathcal{F}'$), let $Z_e^* = \left(Z_e^{*(1)}, \ldots, Z_e^{*(t)}\right)$ be the interval model for edge e in $\mathcal{F}^*$, and let $\tau[y_e] = \left(\tau\left(y_e^{(1)}\right), \ldots, \tau\left(y_e^{(s)}\right)\right)$ be the typical list for edge e in $\mathcal{F}$. Let $a \le b$ be such that $Z_e^{*(a)}$ and $Z_e^{*(b)}$ are the first and last sets in Z_e^*, respectively, that contain vertex x. Analogous to the discussion for a forget node, $s \le t \le s + 2$. Hence, there are four cases to distinguish:

   (a) If $s = t$, there is only one possible typical list $\tau[y_e^\circ]$ for edge e:

   $$\tau[y_e^\circ] = \left(\tau\left(y_e^{(1)}\right), \ldots, \tau\left(y_e^{(a-1)}\right), 1 + \tau\left(y_e^{(a)}\right), \ldots \right.$$
   $$\left. \ldots, 1 + \tau\left(y_e^{(b)}\right), \tau\left(y_e^{(b+1)}\right), \ldots, \tau\left(y_e^{(s)}\right)\right).$$

   (b) If $t = s + 1$ and $Z_e^{*(a-1)} = Z_e^{*(a)} \setminus \{x\}$, there is more than one choice for list $\tau[y_e^\circ]$. The typical list of Y_e can be written as

   $$\tau[y_e] = \left(\tau\left(y_e^{(1)}\right), \ldots, \tau\left(y_e^{(a-2)}\right), \tau\left(y_e^{(a)}\right), \ldots, \tau\left(y_e^{(t)}\right)\right).$$

Consider all possible splits of the typical sequence $\tau\left(y_e^{(a)}\right) = (y_1,\ldots,y_r)$ into two sequences $\tau_1 = (y_1,\ldots,y_f)$ and $\tau_2 = (y_f,\ldots,y_r)$, or $\tau_1 = (y_1,\ldots,y_f)$ and $\tau_2 = (y_{f+1},\ldots,y_r)$; add the typical list

$$\tau[y_e^\circ] = \Big(\tau\left(y_e^{(1)}\right),\ldots,\tau\left(y_e^{(a-2)}\right),\tau_1, 1+\tau_2, 1+\tau\left(y_e^{(a+1)}\right),\ldots$$
$$\ldots, 1+\tau\left(y_e^{(b)}\right),\tau\left(y_e^{(b+1)}\right),\ldots,\tau\left(y_e^{(t)}\right)\Big)$$

to the set of choices for edge e.

(c) If $t = s+1$ and $Z_e^{*(b+1)} = Z_e^{*(b)} \setminus \{x\}$, proceed in a manner similar to the previous case.

(d) If $t = s+2$, the typical list of Y_e can be written as

$$\tau[y_e] = \Big(\tau\left(y_e^{(1)}\right),\ldots,\tau\left(y_e^{(a-2)}\right),\tau\left(y_e^{(a)}\right),\ldots,\tau\left(y_e^{(b)}\right),\tau\left(y_e^{(b+2)}\right),\ldots,\tau\left(y_e^{(t)}\right)\Big).$$

Consider all possible splits of $\tau\left(y_e^{(a)}\right)$ into two sequences τ_1 and τ_2 as in Case (b) and all possible splits of $\tau\left(y_e^{(b)}\right)$ into two sequences τ_3 and τ_4 as in Case (c); add the typical list

$$\tau[y_e^\circ] = \Big(\tau\left(y_e^{(1)}\right),\ldots,\tau\left(y_e^{(a-2)}\right),\tau_1, 1+\tau_2, 1+\tau\left(y_e^{(a+1)}\right),\ldots$$
$$\ldots, 1+\tau\left(y_e^{(b-1)}\right), 1+\tau_3, \tau_4, \tau\left(y_e^{(b+2)}\right),\ldots,\tau\left(y_e^{(t)}\right)\Big)$$

to the set of choices for edge e. If $a = b$, split $\tau\left(y_e^{(a)}\right)$ into three parts τ_1,τ_2,τ_3 and replace $\tau\left(y_e^{(a)}\right)$ by the sequences $\tau_1, 1+\tau_2, \tau_3$ in $\tau[y_e^\circ]$.

2. If $\mathcal{T} \neq \mathcal{T}^*$, the trunk $\mathcal{T}^*$ contains a leaf a that is not a leaf of $\mathcal{T}$. This leaf is the only node in tree decomposition $\mathcal{F}^*$ that stores vertex x. Let b be the neighbor of a in $\mathcal{T}^*$.

   If b is a node of $\mathcal{T}$, add exactly one characteristic $(\mathcal{T}^*, (Z_e^*)_{e\in\mathcal{T}^*}, (\tau[y_e^\circ])_{e\in\mathcal{T}^*})$ to $FS(\mathsf{i})$; the typical lists in this characteristic are defined as $\tau[y_e^\circ] = \tau[y_e]$, for all $e \in \mathcal{T}$, and $\tau[y_e^\circ] = \tau[y_e^*]$, for $e = (\mathsf{a},\mathsf{b})$.

   If b is not a node of $\mathcal{T}$, it has degree three in $\mathcal{T}^*$. Let c and d be the other two neighbors of b in $\mathcal{T}^*$. Vertices c and d are adjacent in $\mathcal{T}$. Let Z_b be the set corresponding to node b in $\mathcal{T}^*$, and let $Z_{e'} = \left(Z_{e'}^{(1)},\ldots,Z_{e'}^{(q)},\ldots,Z_{e'}^{(s)}\right)$ be the interval model for edge $e' = (\mathsf{c},\mathsf{d})$ in $\mathcal{T}$, where $Z_{e'}^{(q)} = Z_\mathsf{b}$. In $C(\mathcal{F}^*)$, this interval model is split into two parts $Z_{e_1}^* = \left(Z_{e'}^{(1)},\ldots,Z_{e'}^{(q)}\right)$ and $Z_{e_2}^* = \left(Z_{e'}^{(q)},\ldots,Z_{e'}^{(s)}\right)$, for the two edges $e_1 = (\mathsf{c},\mathsf{b})$ and $e_2 = (\mathsf{b},\mathsf{d})$ in $\mathcal{T}^*$. Let the typical list of e' in $\mathcal{T}$ be $\tau[y_e'] = \left(\tau\left(y_{e'}^{(1)}\right),\ldots,\tau\left(y_{e'}^{(q)}\right),\ldots,\tau\left(y_{e'}^{(s)}\right)\right)$. Compute all possible type-I splits of $\tau\left(y_{e'}^{(q)}\right) = (y_1,\ldots,y_s)$ into two sequences $\tau_1 = (y_1,\ldots,y_f)$ and $\tau_2 = (y_f,\ldots,y_s)$. Each such split creates one characteristic to be added to $FS(\mathsf{i})$, which is defined by choosing the typical lists for all edges of $\mathcal{T}^*$ as $\tau[y_{e_1}^\circ] = \left(\tau\left(y_{e'}^{(1)}\right),\ldots,\tau\left(y_{e'}^{(q-1)}\right),\tau_1\right)$, $\tau[y_{e_2}^\circ] = \left(\tau_2,\tau\left(y_{e'}^{(q+1)}\right),\ldots,\tau\left(y_{e'}^{(s)}\right)\right)$, and $\tau[y_e^\circ] = \tau[y_e]$, for $e \notin \{e_1,e_2\}$.

6.2 Constructing a Tree Decomposition

In this section, we describe an algorithm that solves the following problem: Given a graph $G = (V, E)$ of treewidth at most ℓ and a tree decomposition $\mathcal{D} = (\mathcal{X}, T)$ of width $k \geq \ell$ for G, compute a tree decomposition of width at most ℓ for G. The algorithm proceeds in four phases, as sketched in Algorithm 5. Next we describe each of these phases in detail.

6.2.1 Phase 1: An Augmented Test Algorithm

In order to facilitate subsequent phases, the testing algorithm of Section 6.1 needs to be augmented to compute additional information.

For a node of T with one child j (i.e., a forget or introduce node), we augment every characteristic $C \in FS(\mathsf{i})$ with a "pointer" to the characteristic $C' \in FS(\mathsf{j})$ from which C has been produced by applying the rules described in Section 6.1. The "pointer" is realized by assigning a unique ID to every characteristic and storing the ID of C' with C. A characteristic C in the full set of a join node stores pointers to the two characteristics in the full sets of its two children from which C has been produced.

In addition to these pointers between characteristics in the full sets of characteristics stored at adjacent nodes, we need more detailed information about how the characteristics in the full set of a node i are obtained from the characteristics in the full sets of its children. We represent this information using additional pointers between elements of related characteristics. We describe these pointers in detail for the case when i is a join node with children j and k. For introduce and forget nodes, the pointers are computed in a similar manner.

For a join node i with children j and k, every characteristic $(\mathcal{T}, (Z_e)_{e\in\mathcal{T}}, (\tau[y_e])_{e\in\mathcal{T}}) \in FS(\mathsf{i})$ is computed from two characteristics $(\mathcal{T}, (Z_e)_{e\in\mathcal{T}}, (\tau[a_e])_{e\in\mathcal{T}}) \in FS(\mathsf{j})$ and $(\mathcal{T}, (Z_e)_{e\in\mathcal{T}}, (\tau[b_e])_{e\in\mathcal{T}}) \in FS(\mathsf{k})$. Since the tree models are the same for all three characteristics, we only have to record how the typical lists $\tau[y_e]$, $e \in \mathcal{T}$, are derived from typical lists $\tau[a_e]$ and $\tau[b_e]$. Consider the computation for a particular edge $e \in \mathcal{T}$. Let $Z_e = \left(Z_e^{(1)}, \ldots, Z_e^{(s)}\right)$ and $\tau[y_e] = \left(\tau\left(y_e^{(1)}\right), \ldots, \tau\left(y_e^{(s)}\right)\right)$. Each sequence $\tau\left(y_e^{(q)}\right)$ is computed from two sequences a^* and $\tau\left(b_e^{(q)}\right)$, where $a^* = \tau\left(a_e^{(q)}\right) - \left|Z_e^{(q)}\right|$. In particular, $\tau\left(y_e^{(q)}\right) = \tau(y^\circ)$, where $y^\circ \in a^* \oplus \tau\left(b_e^{(q)}\right)$; that is, $y^\circ = a^\circ + b^\circ$, for two sequences $a^\circ \in E(a^*)$ and $b^\circ \in E\left(\tau\left(b_e^{(q)}\right)\right)$. Let $a^* = (a_1, \ldots, a_n)$ and $\tau\left(b_e^{(q)}\right) = (b_1, \ldots, b_{n'})$. As $a^\circ \in E(a^*)$, we can store for every element $a_f^\circ \in a^\circ$, the index of the element $a_{p(a_f^\circ)} \in a^*$ of which a_f° is a copy. Similarly, we can store for every $b_f^\circ \in b^\circ$, the index of the element $b_{q(b_f^\circ)} \in \tau\left(b_e^{(q)}\right)$ of which b_f° is a copy. Since $y_f^\circ = a_f^\circ + b_f^\circ$, we define $p(y_f^\circ) = p(a_f^\circ)$ and $q(y_f^\circ) = q(b_f^\circ)$. Finally, every element $y_h \in \tau\left(y_e^{(q)}\right)$ corresponds to an interval of elements $y_{f_h}^\circ, \ldots, y_{g_h}^\circ$ in y° such that $y_f^\circ \leq y_h$, for $f_h \leq f \leq g_h$. Let $r(y_h) = f_h$. This information can easily be computed during the construction of the characteristics in $FS(\mathsf{i})$. Note that $|y^\circ| \leq \left|\tau\left(a_e^{(q)}\right)\right| + \left|\tau\left(b_e^{(q)}\right)\right| - 1 = \mathcal{O}(1)$, by Lemma 3; so we store $\mathcal{O}(1)$ pointers per interval in the interval model of edge e. By Lemma 11, there are $\mathcal{O}(1)$ intervals in the interval model of every edge $e \in \mathcal{T}$ and, by Lemma 12, the trunk $\mathcal{T}$ has $\mathcal{O}(1)$ edges. Hence, we store $\mathcal{O}(1)$ pointers per characteristic in $FS(\mathsf{i})$.

Algorithm 5 Improving the tree decomposition.

Procedure ImproveTreeDecomposition

Input: A graph $G = (V, E)$ with N vertices, a tree decomposition $\mathcal{D} = (\mathcal{X}, T)$ of width at most k and size $\mathcal{O}(N)$ for G, and a constant $\ell \in \mathbb{N}$.

Output: A tree decomposition $\mathcal{E} = (\mathcal{Y}, U)$ of width at most ℓ and size $\mathcal{O}(N)$ for G, or the answer that G has treewidth greater than ℓ.

Run the testing algorithm of Section 6.1.

if G has width at most ℓ **then**

  Process T top-down to build a tree of characteristics:

  For every node i of T, choose a characteristic $C_\mathsf{i} \in FS(\mathsf{i})$: For the root r of T, choose an arbitrary characteristic $C_\mathsf{r} \in FS(\mathsf{r})$ as the root characteristic. For a forget or introduce node i with child j, choose C_j so that C_i has been produced from C_j during the construction of $FS(\mathsf{i})$. If i is a join node with children j and k, choose C_j and C_k so that C_i has been produced from C_j and C_k during the construction of $FS(\mathsf{i})$.

  Compute an implicit representation of tree decomposition $\mathcal{E} = (\mathcal{Y}, U)$:

  Let $U_1, \ldots, U_r$ be the maximal paths in U whose internal nodes have degree two in U. The implicit representation of $\mathcal{E}$ consists of two parts: (1) a graph $\mathcal{G}$ whose connected components are flippable DAGs representing the path decompositions induced by paths $U_1, \ldots, U_r$; (2) a "link list" $\mathcal{L}$ that connects paths $U_1, \ldots, U_r$ to form tree U.

  Compute $\mathcal{E}$ explicitly:

  Apply time-forward processing to $\mathcal{G}$ in order to compute all path decompositions mentioned in the previous step; link them together to form U.

end if

6.2.2 Phase 2: Building a Tree of Characteristics

If G has treewidth at most ℓ, the full set of characteristics $FS(\mathsf{r})$ of the root r of T is non-empty. Every characteristic in $FS(\mathsf{r})$ is obtained from tree decompositions computed for the graphs represented by the leaves of T, by appropriately merging and augmenting characteristics along the way from the leaves of T to the root. We choose one characteristic C_r in $FS(\mathsf{r})$; our goal is to construct the tree decomposition of G represented by C_r. We accomplish the first step towards this goal by tracing back all characteristics down to the leaves of T that were involved in the construction of C_r. In particular, we extract one characteristic C_i per node i of T: For the root r, we choose characteristic C_r arbitrarily. For every introduce or forget node i with child j, we choose characteristic C_j as the characteristic from which characteristic C_i was computed in Phase 1 of the algorithm. For a join node i with children j and k, we choose characteristics C_j and C_k so that C_i was computed from C_i and C_j during Phase 1 of the algorithm. The computation of all characteristics C_i, $\mathsf{i} \in T$, can be carried out in $\mathcal{O}(\text{sort}(N))$ I/Os using the time-forward processing technique to process T top-down, and using the pointers between characteristics computed in Phase 1 of the algorithm to choose characteristics C_i, for all non-root nodes i of T.

6.2.3 The Structure of a Tree Decomposition for G

Before describing Phases 3 and 4, we discuss in this section how the characteristics chosen in Phase 2 define a tree decomposition $\mathcal{E}$ of width at most ℓ for G. In particular, we show how to derive for every node $\mathsf{i} \in T$, a partial tree decomposition $\mathcal{E}_\mathsf{i}$ rooted at i whose width is at most ℓ. The bound on the width of $\mathcal{E}_\mathsf{i}$ follows immediately, if each set Y_l in the tree decomposition can be associated with a particular entry y_h in a typical sequence so that $|Y_l| \le y_h$, because for all these entries, $y_h \le \ell$. Again, we discuss the four different node types separately:

Start node: If i is a start node, consider an edge e of the trunk $\mathcal{T}$ in the characteristic C_i, and let $Z_e = \left(Z_e^{(1)}, \dots, Z_e^{(s)}\right)$. Recall that the path decomposition Y_e corresponding to edge e is equal to Z_e. Hence, a tree decomposition $\mathcal{E}_\mathsf{i}$ of G_i can be obtained by replacing every edge e of the trunk with the path decomposition Z_e. Every typical sequence $\tau\left(y_e^{(q)}\right)$ consists of a single element y_1, which corresponds to the only set Y_1 in the path decomposition $Y_e^{(q)}$; that is, $y_1 = |Y_1|$.

Forget node: For a forget node i with child j, the partial tree decomposition $\mathcal{E}_\mathsf{i}$ is the same as $\mathcal{E}_\mathsf{j}$ if $\mathcal{T} = \mathcal{T}^*$; otherwise, $\mathcal{E}_\mathsf{i}$ contains one or two more nodes than $\mathcal{E}_\mathsf{j}$, as discussed below. While the addition of nodes in the case $\mathcal{T} \ne \mathcal{T}^*$ is not strictly necessary, it simplifies the I/O-efficient construction of $\mathcal{E}_\mathsf{i}$, described in Sections 6.2.4 and 6.2.5.

Although tree decompositions $\mathcal{E}_\mathsf{i}$ and $\mathcal{E}_\mathsf{j}$ are the same if $\mathcal{T} = \mathcal{T}^*$, the way different parts the decomposition correspond to entries in the characteristics changes. Every path decomposition Y_e corresponding to a trunk edge e can be split into intervals, each corresponding to an entry in $\tau[y_e]$. More precisely, if $y_h \in \tau\left(y_e^{(q)}\right)$, then for all sets Y_l in the interval Y_{y_h} corresponding to y_h, $X_\mathsf{i} \cap Y_l = Z_e^{(q)}$ and $|Y_l| \le y_h$. If $|Z_e| = |Z_e^*|$, $\tau[y_e^*] = \tau[y_e]$, and we associate the same intervals in Y_e with the elements in $\tau[y_e^*]$ as with their counterparts in $\tau[y_e]$. Otherwise, consider the case $|Z_e^*| = |Z_e| - 1$ and $Z_e^{(a-1)} = Z_e^{(a)} \setminus \{x\}$. (The two other cases are similar.) Then sequences$\tau\left(y_e^{(a-1)}\right)$ and $\tau\left(y_e^{(a)}\right)$ in $\tau[y_e]$ are replaced with a sequence $\tau(y^\circ)$ in $\tau[y_e^*]$, where $y^\circ = \tau\left(y_e^{(a-1)}\right) \circ \tau\left(y_e^{(a)}\right)$. Every entry $y_f^\circ \in y^\circ$ represents the same interval of Y_e as its corresponding entry y_h in $\tau\left(y_e^{(a-1)}\right)$ or $\tau\left(y_e^{(a)}\right)$; that is, $Y_{y_f^\circ} = Y_{y_h}$. Hence, for every set $Y_l \in Y_{y_f^\circ}$, $|Y_l| \le y_f^\circ$. Sequence $\tau(y^\circ)$ is derived from y° by the application of typical operations and the removal of consecutive duplicates. Therefore, every element y_k in $\tau\left(y_e^{*(a)}\right)$ corresponds to an interval $I(y_k)$ of elements in y° such that for every element $y_f^\circ \in I(y_k)$, $y_f^\circ \le y_k$. We let $Y_{y_k} = \bigcirc_{y_f^\circ \in I(y_k)} Y_{y_f^\circ}$. Since for each element $y_f^\circ \le y_k$ and for every set $Y_l \in Y_{y_f^\circ}$, $|Y_l| \le y_f^\circ$, it follows that for every set $Y_l \in Y_{y_k}$, $|Y_l| \le y_k$.

If $\mathcal{T} \ne \mathcal{T}^*$, there is an edge $e = (\mathsf{a}, \mathsf{b})$, incident to a leaf a of $\mathcal{T}$, that is not present in $\mathcal{T}^*$. In addition, if the non-leaf endpoint b of e has degree three in $\mathcal{T}$, it is not present in $\mathcal{T}^*$ either, and its two other neighbors c and d in $\mathcal{T}$ are connected by an edge (c, d) in $\mathcal{T}^*$. For every edge e' of $\mathcal{T}$ that is present in $\mathcal{T}^*$, the corresponding path decomposition $Y_{e'}$ does not change, neither do the associations of the elements of $Y_{e'}$ with the elements of $(Z_{e'}, \tau[y_{e'}])$. The only differences between tree decompositions $\mathcal{E}_\mathsf{i}$ and $\mathcal{E}_\mathsf{j}$ are in the path decompositions corresponding to edges (a, b) and (c, d).

For the removed edge $e = (\mathsf{a}, \mathsf{b})$, the corresponding path decomposition Y_e is no longer part of the filled trunk of C_i; it is still part of the tree decomposition $\mathcal{E}_\mathsf{i}$, but can no longer be modified in subsequent augmentations of $\mathcal{E}_\mathsf{i}$ because only parts of the filled trunk can be involved in these augmentations. In order to record the fact that Y_e is part of $\mathcal{E}_\mathsf{i}$, we will store a link that records the attachment of Y_e to the filled trunk of C_{i}. By leaving Y_e unchanged and attaching it to the filled trunk in this manner, we implicitly duplicate one endpoint x of Y_e, the one corresponding to node b. This is true because the path decomposition of every edge in the filled trunk of $\mathcal{E}_{\mathsf{j}}$ contains one copy of x. By detaching path decomposition Y_e from the filled trunk, one copy of x remains in the filled trunk, and another copy remains in Y_e. We could avoid this duplication, either by removing node x from Y_e before detaching Y_e or by excluding node x from the path decompositions corresponding to all trunk edges incident to b and including it only when node b disappears as the result of merging the last two edges incident to b. The former is not feasible because we use flippable DAGs to represent the path decompositions corresponding to the edges in the filled trunk and flippable DAGs are strictly incremental; the latter would complicate the whole construction phase unnecessarily.

For the same reason, the replacement of edges (c, b) and (b, d) in $\mathcal{T}$ with edge (c, d) in $\mathcal{T}^*$ results in the duplication of the same node x that is duplicated as a result of detaching edge (a, b). In particular, we compute $Y^*_{(\mathsf{c},\mathsf{d})} = Y_{(\mathsf{c},\mathsf{b})} \circ Y_{(\mathsf{b},\mathsf{d})}$, so that $Y_{(\mathsf{c},\mathsf{d})}$ contains two consecutive copies of node x. The characteristic $(Z^*_{(\mathsf{c},\mathsf{d})}, \tau[y^*_{(\mathsf{c},\mathsf{d})}])$ of edge (c, d) is obtained from characteristics $(Z_{(\mathsf{c},\mathsf{b})}, \tau[y_{(\mathsf{c},\mathsf{b})}])$ and $(Z_{(\mathsf{b},\mathsf{d})}, \tau[y_{(\mathsf{b},\mathsf{d})}])$ by computing two sequences $Z^\circ = Z_{(\mathsf{c},\mathsf{b})} \circ Z_{(\mathsf{b},\mathsf{d})}$ and $y^\circ = \tau[y_{(\mathsf{c},\mathsf{b})}] \circ \tau[y_{(\mathsf{b},\mathsf{d})}]$, removing consecutive duplicates from Z°, and computing $\tau[y^*_{(\mathsf{c},\mathsf{d})}] = \tau[y^\circ]$. The associations between the elements in $Y^*_{(\mathsf{c},\mathsf{d})}$ and the elements in Z° and y° are the same as the associations between the corresponding elements in $Y_{(\mathsf{c},\mathsf{b})}$, $Y_{(\mathsf{b},\mathsf{d})}$, $Z_{(\mathsf{c},\mathsf{b})}$, $Z_{(\mathsf{b},\mathsf{d})}$, $\tau[y_{(\mathsf{c},\mathsf{b})}]$, and $\tau[y_{(\mathsf{b},\mathsf{d})}]$. The associations of the elements in $Z^*_{(\mathsf{c},\mathsf{d})}$ and $\tau[y^*_{(\mathsf{c},\mathsf{d})}]$ with the elements of $Y^*_{(\mathsf{c},\mathsf{d})}$ are derived from the associations of the elements of Z° and y° with the elements of $Y^*_{(\mathsf{c},\mathsf{d})}$ in the same manner as described for the case $\mathcal{T} = \mathcal{T}^*$.

Clearly, the expansion of a tree-node into two tree nodes connected by an edge does not affect the validity or width of the tree decomposition. For all other updates, it follows from the same arguments as in the case $\mathcal{T} = \mathcal{T}^*$ that for every element Y_l of a path decomposition Y_e and its corresponding element y_h of $\tau[y_e]$, $|Y_l| \le y_h$.

Join node: Now consider a join node i with children j and k. We are given two tree decompositions $\mathcal{E}_{\mathsf{j}}$ and $\mathcal{E}_{\mathsf{k}}$ for graphs G_{j} and G_{k}, both partially represented by characteristics C_{j} and C_{k}. We know that $G_{\mathsf{i}} = G_{\mathsf{j}} \cup G_{\mathsf{k}}$. Our goal is to merge tree decompositions $\mathcal{E}_{\mathsf{j}}$ and $\mathcal{E}_{\mathsf{k}}$ into a new tree decomposition $\mathcal{E}_{\mathsf{i}}$ for G_{i}. Intuitively, we do this as follows: First we "stretch" the edges of the filled trunks of C_{j} and C_{k} so that the path decompositions in $\mathcal{E}_{\mathsf{j}}$ and $\mathcal{E}_{\mathsf{k}}$ corresponding to these edges contain the same number of sets. Then we identify the nodes of the filled trunks in $\mathcal{E}_{\mathsf{j}}$ and $\mathcal{E}_{\mathsf{k}}$ with each other and compute for every node v with corresponding sets A_{v} and B_{v} in $\mathcal{E}_{\mathsf{j}}$ and $\mathcal{E}_{\mathsf{k}}$, respectively, a new set $Y_{\mathsf{v}} = A_{\mathsf{v}} \cup B_{\mathsf{v}}$ in $\mathcal{E}_{\mathsf{i}}$. Clearly, the resulting decomposition is a tree decomposition for G_{i}, as every edge in G_{i} must be either in G_{j} or in G_{k}. We have to bound the width of the tree decomposition. Again, we can ignore the parts of both decompositions $\mathcal{E}_{\mathsf{j}}$ and $\mathcal{E}_{\mathsf{k}}$ that are not in the filled trunks because they are not involved in any updates and thus remain valid. So let us see how the "stretching" is done:

Consider an edge $e \in \mathcal{T}$. For the path decomposition corresponding to edge e, we have to match up the parts of A_e and B_e corresponding to the same interval $Z_e^{(q)}$; that is, certain entries in A_e and B_e have to be duplicated so that for each interval $Z_e^{(q)}$, the number of corresponding entries in A_e and B_e is the same. Also, we have to guarantee a bound on the width of the resulting tree decomposition.

Consider an interval $Z_e^{(q)}$ with typical sequences $\tau\left(a_e^{(q)}\right)$ and $\tau\left(b_e^{(q)}\right)$ in C_{j} and C_{k}, respectively. Let $a^* = \tau\left(a_e^{(q)}\right) - \left|Z_e^{(q)}\right| = (a_1, \ldots, a_n)$, $\tau\left(b_e^{(q)}\right) = (b_1, \ldots, b_{n'})$, and $\tau\left(y_e^{(q)}\right) = (y_1, \ldots, y_m)$. Recall that $\tau\left(y_e^{(q)}\right) = \tau(y^\circ)$, for some sequence $y^\circ \in a * \oplus \tau\left(b_e^{(q)}\right)$; that is, $y^\circ = a^\circ + b^\circ$, where a° and b° are extensions of a^* and $\tau\left(b_e^{(q)}\right)$, respectively. Let $y^\circ = (y_1^\circ, \ldots, y_c^\circ)$. Initally, there are paths A_{a_h} and B_{b_h} associated with each entry $a_h \in a^*$ and $b_h \in \tau\left(b_e^{(q)}\right)$. Next we associate paths $A_{a_f^\circ}$ and $B_{b_f^\circ}$ with elements $a_f^\circ \in a^\circ$ and $b_f^\circ \in b^\circ$ so that $\bigcirc_{f=1}^c A_{a_f^\circ} \in E(\bigcirc_{h=1}^n A_{a_h})$ and $\bigcirc_{f=1}^c B_{b_f^\circ} \in E(\bigcirc_{h=1}^{n'} B_{b_h})$: For every path A_{a_h}, let $\hat{A}_{a_h}$ be the path consisting of only the first set in A_{a_h}. Analogously, let $\hat{B}_{b_h}$ be the path consisting of only the first set in B_{b_h}. Then we define

$$A_{a_f^\circ} = \begin{cases} A_{a_{p(a_f^\circ)}} & f = c \vee p(a_f^\circ) < p(a_{f+1}^\circ) \\ \hat{A}_{a_{p(a_f^\circ)}} & f < c \wedge p(a_f^\circ) = p(a_{f+1}^\circ) \end{cases},$$

$$B_{b_f^\circ} = \begin{cases} B_{b_{q(b_f^\circ)}} & f = c \vee q(b_f^\circ) < q(b_{f+1}^\circ) \\ \hat{B}_{b_{q(b_f^\circ)}} & f < c \wedge q(b_f^\circ) = q(b_{f+1}^\circ) \end{cases}.$$

Now let $\kappa(y_f^\circ) = \max\{|A_{a_f^\circ}|, |B_{b_f^\circ}|\}$. We define two new path decompositions $\bar{A}_{a_f^\circ}$ and $\bar{B}_{b_f^\circ}$. First we define $\bar{A}_{a_f^\circ} = A_{a_f^\circ}$, then we increase the length of $\bar{A}_{a_f^\circ}$ to $\kappa(y_f^\circ)$ by duplicating the first set in $\bar{A}_{a_f^\circ}$. $\bar{B}_{b_f^\circ}$ is defined similarly. Let $\bar{A}_{a_f^\circ} = (\bar{A}_1, \ldots, \bar{A}_r)$ and $\bar{B}_{b_f^\circ} = (\bar{B}_1, \ldots, \bar{B}_r)$. We define $Y_{y_f^\circ} = (Y_1, \ldots, Y_r)$, where $Y_l = \bar{A}_l \cup \bar{B}_l$. Finally, the path decomposition associated with every element y_h is

$$Y_{y_h} = \begin{cases} \bigcirc_{f=r(y_h)}^{r(y_{h+1})-1} Y_{y_f^\circ} & h < m \\ \bigcirc_{f=r(y_h)}^{c} Y_{y_f^\circ} & h = m \end{cases}.$$

Clearly, the resulting tree decomposition is valid. It remains to show that for every set $Y_l \in Y_{y_h}$, $|Y_l| \leq y_h$. It is easily verified that for each $a_f^\circ \in a^\circ$, every set in $\bar{A}_{a_f^\circ}$ has size at most $a_f^\circ + \left|Z_e^{(q)}\right|$, and for each $b_f^\circ \in b^\circ$, every set in $\bar{B}_{b_f^\circ}$ has size at most b_f°. Hence, every set in the path decomposition $Y_{y_f^\circ}$ has size at most $a_f^\circ + b_f^\circ = y_f^\circ$. Now it remains to observe that for every y_h and every $y_f^\circ \in I(y_h)$, $y_f^\circ \leq y_h$.

Introduce node: Finally, if i is an introduce node with child j, we distinguish two cases again, depending on whether $\mathcal{T} = \mathcal{T}^*$ or $\mathcal{T} \neq \mathcal{T}^*$. In the former case, we have to update the path decompositions associated with all those trunk edges whose characteristics in C_{i} and C_{j} differ. Consider such a trunk edge e. If $|Z_e| = |Z_e^*|$, let $\tau[y_e] = \left(\tau\left(y_e^{(1)}\right), \ldots, \tau\left(y_e^{(q)}\right)\right)$ and $\tau[y_e^*] = \left(\tau\left(y_e^{(1)}\right), \ldots, \tau\left(y_e^{(a-1)}\right), 1 + \tau\left(y_e^{(a)}\right), \ldots, 1 + \tau\left(y_e^{(b)}\right), \tau\left(y_e^{(b+1)}\right), \ldots, \tau\left(y_e^{(q)}\right)\right)$. Then we add the introduced vertex x to all sets in Y_e corresponding to intervals $Z_e^{(a)}, \ldots, Z_e^{(b)}$. If $|Z_e| < |Z_e^*|$, assume that $|Z_e^*| = |Z_e| + 1$ and $Z_e^{*(a-1)} = Z_e^{*(a)} \setminus \{x\}$. (The other two cases are similar.) In this case, we obtain $\tau[y_e^*]$ from $\tau[y_e]$ by splitting the typical sequence $\tau\left(y_e^{(a)}\right)$ into two sequences τ_1 and τ_2. We associate with every element in τ_1 and τ_2 the same subsequence of Y_e as with the corresponding element in $\tau\left(y_e^{(a)}\right)$. If the split of sequence $\tau\left(y_e^{(a)}\right)$ is of type I, let $\tau_1 = (y_1, \ldots, y_f)$ and $\tau_2 = (y_f, \ldots, y_r)$. Then we make a copy of the first set in the path decomposition associated with y_f in $\tau\left(y_e^{(a)}\right)$ and associate this copy with y_f in τ_1. Afterwards, we proceed as in the case $|Z_e| = |Z_e^*|$.

If $\mathcal{T} \neq \mathcal{T}^*$, let (a, b) be the edge to be attached to $\mathcal{T}$. The path decomposition $Y_{(\mathsf{a},\mathsf{b})}$ is the same as given in the tree decomposition computed for $G[X_\mathsf{i}]$. If b is a node of $\mathcal{T}$, the attachment of this path decomposition to $\mathcal{E}_\mathsf{j}$ is all that has to be done. Otherwise, the path decomposition $Y_{(\mathsf{c},\mathsf{d})}$ has to be split into two path decompositions $Y_{(\mathsf{c},\mathsf{b})}$ and $Y_{(\mathsf{b},\mathsf{c})}$. Note that the split is of type one; hence, we perform same set-duplication as described above for this type of split.

It is straightforward to verify that for every element y_h of a typical list and every set $Y_l \in Y_{y_h}$, $|Y_l| \leq y_h$.

Lemma 15 *For every node $\mathsf{i} \in T$, $\mathcal{E}_\mathsf{i}$ is a tree decomposition of G_i.*

Proof. We have to verify Properties T1–T3, which is easily done for Properties T1 and T3. To prove Property T2, we use induction on the size of the tree T_i rooted at node i. If $|T_\mathsf{i}| = 1$, then i is a start node. Property T2 is obviously satisfied in this case, because $\mathcal{E}_\mathsf{i} = \mathcal{F}$ is the tree decomposition of $G_\mathsf{i} = G[X_\mathsf{i}]$ from which C_i has been derived.

If $|T_\mathsf{i}| > 1$, node i is a forget, introduce, or join node. For a forget node i with child j, we start with a tree decomposition $\mathcal{E}_\mathsf{j}$ for $G_\mathsf{j} = G_\mathsf{i}$ and possibly augment it by expanding a single node in $\mathcal{E}_\mathsf{j}$ into a tree of size two or three, all of whose nodes store the same sets. Hence, $\mathcal{E}_\mathsf{i}$ has Property T2.

For an introduce node i with child j, we start with a tree decomposition $\mathcal{E}_\mathsf{j}$ for G_j. The only edgesof G_i not represented in $\mathcal{E}_\mathsf{j}$ are those between x and its neighbors in X_i. These edges are represented in the tree decomposition $\mathcal{F}$ of $G[X_\mathsf{i}]$ from which the tree model of C_i was derived. Hence, all these edges must be in the sets $Z_e^{*(q)}$ of the tree model $(\mathcal{T}^*, (Z_e^*)_{e \in \mathcal{T}^*})$ of $\mathcal{F}$. It is easy to verify that $\mathcal{E}_\mathsf{i}$ has the same tree model as $\mathcal{F}$. Thus, for every edge $e = \{x, v\}$ incident to x, there must be a set in $\mathcal{E}_\mathsf{i}$ containing both x and v.

For a join node i with children j and k, $\mathcal{E}_\mathsf{i}$ can be obtained by augmenting either $\mathcal{E}_\mathsf{j}$ or $\mathcal{E}_\mathsf{k}$ appropriately. Since $G_\mathsf{i} = G_\mathsf{j} \cup G_\mathsf{k}$, every edge in G_i must be either in G_j or in G_k. Hence, there is a node in $\mathcal{E}_\mathsf{i}$ containing both endpoints of the edge. □

Lemma 16 *The tree decomposition $\mathcal{E} = \mathcal{E}_\mathsf{r}$ of G has size $\mathcal{O}(N)$ and width at most ℓ.*

Proof. The fact that $\mathcal{E}$ has width ℓ follows immediately from the above discussion. In particular, every set Y_l in thefilled trunk has cardinality at most $y_h \leq \ell + 1$, where $Y_l \in Y_{y_h}$. Every set that is not in the filled trunk had cardinality at most $\ell + 1$ when it was part of the filled trunk and cannot grow any further after its corresponding trunk edge is removed from the trunk. In order to show that $\mathcal{E}$ has size $\mathcal{O}(N)$, we begin with some observations:

For every start node $\mathsf{i} \in T$, it follows from Lemma 10 that $|\mathcal{E}_\mathsf{i}| \leq (2k+1)^2$, because $\mathcal{E}_\mathsf{i}$ is a minimal tree decomposition of $G_\mathsf{i} = G[X_\mathsf{i}]$.

For a forget node i with child j, $|\mathcal{E}_\mathsf{i}| \leq |\mathcal{E}_\mathsf{j}| + 2$. Equality holds if $\mathcal{T} \neq \mathcal{T}^*$ and the neighbor b of the removed leaf a is not a node of $\mathcal{T}^*$. Indeed, if $\mathcal{T} = \mathcal{T}^*$, the path decomposition does not change. If $\mathcal{T} \neq \mathcal{T}^*$, the size of the tree decomposition increases by one as a result of the duplication of node b before detaching path decomposition $Y_{(\mathsf{a},\mathsf{b})}$ from the filled trunk. If node b has degree three in $\mathcal{T}$, the tree decomposition gains another node as a result of the duplication of node b when merging path decompositions $Y_{(\mathsf{c},\mathsf{b})}$ and $Y_{(\mathsf{b},\mathsf{d})}$ in $\mathcal{E}_\mathsf{i}$.

For an introduce node i with child j, $|\mathcal{E}_\mathsf{i}| \le |\mathcal{E}_\mathsf{j}| + 4k$. Indeed, if $\mathcal{T} = \mathcal{T}^*$, only the type-I splits we perform change the size of the tree decomposition. A type-I split leads to the duplication of the node corresponding to the element y_f shared by the two sequences resulting from the split. As we perform at most two type-I splits per edge of $\mathcal{T}$ and $|\mathcal{T}| \le 2k$, by Lemma 12, the size of the tree decomposition increases by at most $4k$ in this case. (A more careful analysis shows that in fact the size of the tree decomposition increases by at most $k+1$.) If $\mathcal{T} \ne \mathcal{T}^*$, we possibly perform a type-I split of an edge of $\mathcal{T}$ and then attach the path decomposition corresponding to the attached edge of $\mathcal{T}^*$. The type-I split increases the size of the tree decomposition by one; the attached path decomposition has length at most $2k+3$, by Lemma 11.

For a join node i with children j and k, $|\mathcal{E}_\mathsf{i}| \le |\mathcal{E}_\mathsf{j}| + |\mathcal{E}_\mathsf{k}|$. Indeed, consider an edge e in the trunk $\mathcal{T}$, and an interval $Z_e^{(q)}$ along this edge. Let A, B, and Y be the path decompositions corresponding to this interval. Then we claim that $|Y| \le |A| + |B| - 1$. Applying this claim to all intervals in the tree model gives the desired result.

Let a° and b° be the usual extensions of $a^* = \tau\left(a_e^{(q)}\right) - \left|Z_e^{(q)}\right|$, and let $\tau\left(b_e^{(q)}\right)$ and $y^\circ = a^\circ + b^\circ$. By Lemma 3, $|y^\circ| \le |a^*| + \left|\tau\left(b_e^{(q)}\right)\right| - 1$. For every element $a_h \in a^*$ or $b_h \in \tau\left(b_e^{(q)}\right)$, let A_h or B_h be the corresponding path decomposition. Then $\sum_{h=1}^{n} |A_h| = |A|$ and $\sum_{h=1}^{n'} |B_h| = |B|$. For every element $a_f^\circ \in a^\circ$ or $b_f^\circ \in b^\circ$, let A_f° or B_f° be the corresponding path decomposition. Observe that every path decompositions A_h appears exactly once in the list of path decompositions A_f°; all other path decompositions A_f° are path decompositions $\hat{A}_h$ and have size one. Hence, $\sum_{f=1}^{c} |A_f^\circ| \le |A| + (c-n)$. Analogously, $\sum_{f=1}^{c} |B_f^\circ| \le |B| + (c-n')$. Finally, for every path decomposition Y_f° corresponding to an element y_f°, $|Y_f^\circ| = \max\{|A_f^\circ|, |B_f^\circ|\} \le |A_f^\circ| + |B_f^\circ| - 1$. Hence,

$$\begin{aligned}
|Y| &= \sum_{f=1}^{c} |Y_f^\circ| \\
&\le \sum_{f=1}^{c} (|A_f^\circ| + |B_f^\circ| - 1) \\
&= \sum_{f=1}^{c} |A_f^\circ| + \sum_{f=1}^{c} |B_f^\circ| - c \\
&\le |A| + (c-n) + |B| + (c-n') - c \\
&= |A| + |B| + (c - n - n') \\
&\le |A| + |B| - 1.
\end{aligned}$$

Now we claim that the tree decomposition $\mathcal{E}_\mathsf{i}$ rooted at node i has size at most $(2k+1)^2|T_\mathsf{i}|$. This then implies that $|\mathcal{E}| = |\mathcal{E}_\mathsf{r}| \le (2k+1)^2|T_\mathsf{r}| \le 4(2k+1)^2N = \mathcal{O}(N)$ because we will show in Section 7 that we can construct a nice tree decomposition of size at most $4N$, for every graph of bounded treewidth.

The proof of the claim is by induction on $|T_\mathsf{i}|$. If $|T_\mathsf{i}| = 1$, i is a start node and $|\mathcal{E}_\mathsf{i}| \le (2k+1)^2 = (2k+1)^2|T_\mathsf{i}|$. Otherwise, i is a join, forget, or introduce node.

If i is a forget node with child j, $|\mathcal{E}_\mathsf{j}| \le (2k+1)^2|T_\mathsf{j}|$, by the induction hypothesis, and $|\mathcal{E}_\mathsf{i}| \le |\mathcal{E}_\mathsf{j}| + 2$. Also, $|T_\mathsf{i}| = |T_\mathsf{j}| + 1$. Hence, $|\mathcal{E}_\mathsf{i}| \le (2k+1)^2|T_\mathsf{i}|$.

If i is an introduce node with child j, $|\mathcal{E}_\mathsf{j}| \le (2k+1)^2|T_\mathsf{j}|$, by the induction hypothesis, and $|\mathcal{E}_\mathsf{i}| \le |\mathcal{E}_\mathsf{j}| + 4k \le (2k+1)^2|T_\mathsf{j}| + (2k+1)^2 = (2k+1)^2(|T_\mathsf{j}|+1) = (2k+1)^2|T_\mathsf{i}|$.

Finally, if i is a join node with children j and k, $|\mathcal{E}_\mathsf{j}| \le (2k+1)^2|T_\mathsf{j}|$, $|\mathcal{E}_\mathsf{k}| \le (2k+1)^2|T_\mathsf{k}|$, and $|\mathcal{E}_\mathsf{i}| \le |\mathcal{E}_\mathsf{j}| + |\mathcal{E}_\mathsf{k}| \le (2k+1)^2(|T_\mathsf{j}| + |T_\mathsf{k}|) < (2k+1)^2|T_\mathsf{i}|$. □

6.2.4 Phase 3: Constructing the Tree Decomposition Implicitly

The goal of the next two phases of the algorithm is to construct tree decomposition $\mathcal{E} = \mathcal{E}_\mathsf{r}$. First we build an implicit representation of $\mathcal{E}$. To do this, we process tree T from the leaves towards the root; at every node i, we compute an implicit representation of $\mathcal{E}_\mathsf{i}$ from the representations computed for the partial tree decompositions rooted at its children. in Section 6.2.5, we show how to extract $\mathcal{E}$ from the computed implicit representation.

Before we start describing the third phase of our algorithm, we define the implicit representation of a tree decomposition $\mathcal{E}_\mathsf{i} = (\mathcal{Y}_\mathsf{i}, U_\mathsf{i})$, $\mathsf{i} \in T$. First we concentrate on the representation of the filled trunk of $\mathcal{E}_\mathsf{i}$. Consider a characteristic $C_\mathsf{i} = (\mathcal{T}, (Z_e)_{e\in\mathcal{T}}, (\tau[y_e])_{e\in\mathcal{T}})$, an edge $e \in \mathcal{T}$, and an interval $Z_e^{(q)} \in Z_e$. An entry $y_h \in \tau\left(y_e^{(q)}\right)$ corresponds to a path decomposition Y_{y_h} that is part of $\mathcal{E}_\mathsf{i}$. We represent each such path decomposition Y_{y_h} by a flippable DAG $\mathcal{G}(y_h)$. Entry y_h stores the source and sink of $\mathcal{G}(y_h)$, and entry $\kappa(y_h) = |Y_{y_h}|$, and colors $c(\sigma)$ and $c(\tau)$. Graph $\mathcal{G}(y_h)$ has the following properties:

(G1) $\mathcal{G}(y_h)$ is a flippable DAG.

(G2) Every node $\alpha \in \mathcal{G}(y_h)$ is labeled with a triple $(L_\alpha, R_\alpha, \rho_\alpha)$. The integer ρ_α is the "stretch" of node α; that is, in path decomposition Y_{y_h}, the content of node α is to be duplicated over ρ_α consecutive nodes. The set L_α contains all vertices $x \in G$ that have to appear in all copies of node α as well as all nodes succeeding the last copy of node α in path decomposition Y_{y_h}, up to the last copy of a node α' with $x \in R_{\alpha'}$.

(G3) Let σ and τ be the source and sink of $\mathcal{G}(y_h)$, respectively. For any $\sigma\tau$-path $p = \langle\sigma = \alpha_0, \alpha_1, \ldots, \alpha_k = \tau\rangle$ in $\mathcal{G}(y_h)$, $\sum_{l=0}^{k} \rho(\alpha_l) = |Y_{y_h}|$. In particular, we can define an interval $I(\alpha_l) = [a, b]$ for every node α_l on this path, where $a = 1 + \sum_{j=0}^{l-1} \rho(\alpha_j)$ and $b = \sum_{j=0}^{l} \rho(\alpha_j)$. If $\rho_{\alpha_l} = 0$, let $I(\alpha_l) = \emptyset$. Note that this interval is independent of the path used to compute it.

(G4) For every set $Y_l \in Y_{y_h}$ and every vertex $x \in Y_l$, there exist unique nodes $\mu(x)$ and $\nu(x)$ in $\mathcal{G}(y_h)$ such that $x \in L_{\mu(x)}$ and $x \in R_{\nu(x)}$.

(G5) For every set $Y_l \in Y_{y_h}$ and every vertex $x \in Y_l$, let $I(x)$ be the smallest interval containing $I(\mu(x))$ and $I(\nu(x))$. Let $Y_{y_h} = (Y_1, \ldots, Y_r)$. Then $Y_l = \{x \in V : l \in I(x)\}$, for $1 \le l \le r$.

The portion of $\mathcal{E}_\mathsf{i}$ that is not in the filled trunk can be partitioned into path decompositions. In particular, whenever an edge (a, b) is detached from the filled trunk, we construct a flippable DAG $\mathcal{G}((\mathsf{a}, \mathsf{b}))$ that has Properties (G1)–(G5) w.r.t. $Y_{(\mathsf{a},\mathsf{b})}$. We record the attachment of $Y_{(\mathsf{a},\mathsf{b})}$ to the filled trunk of the tree decomposition by storing a *link record* in a link list $\mathcal{L}$. As a result of detaching edges from the filled trunks at descendants of i, the portion of $\mathcal{E}_\mathsf{i}$ that is not in the filled trunk is partitioned into a set of path decompositions Y_e, each represented by a flippable DAG $\mathcal{G}(e)$; each such path decomposition is linked either to the filled trunk of $\mathcal{E}_\mathsf{i}$ or to another path decomposition $Y_{e'}$ by a link record.

In the remainder of this section, we describe how to construct this information by processing T bottom-up. As before, we deal with the four different node types separately:

Start node: For a start node i, the above information can be set up easily. The filled trunk of $\mathcal{E}_\mathsf{i}$ equals $\mathcal{E}_\mathsf{i}$. For every trunk edge e with $\tau[y_e] = \left(\tau\left(y_e^{(1)}\right), \ldots, \tau\left(y_e^{(s)}\right)\right)$ and every $1 \leq q \leq s$, $\tau\left(y_e^{(q)}\right)$ consists of a single entry. For this entry y, the graph $\mathcal{G}(y)$ consists of a single node $\sigma = \tau$ with $L_\sigma = R_\sigma = Z_e^{(q)}$ and $\rho_\sigma = 1$; $\kappa(y) = 1$, $c(\sigma) = c(\tau) = \mathsf{red}$.

Forget node: If i is a forget node with child j, let $C_\mathsf{i} = (\mathcal{T}^*, (Z_e^*)_{e\in\mathcal{T}^*}, (\tau[y_e^*])_{e\in\mathcal{T}^*})$ and $C_\mathsf{j} = (\mathcal{T}, (Z_e)_{e\in\mathcal{T}}, (\tau[y_e])_{e\in\mathcal{T}})$ be the characteristics associated with nodes i and j, respectively. Again, we distinguish two cases, depending on whether nor not $\mathcal{T} = \mathcal{T}^*$. If $\mathcal{T} = \mathcal{T}^*$, consider the path decomposition Y_e for an edge $e \in \mathcal{T}$. If $|Z_e| = |Z_e^*|$, $\tau[y_e] = \tau[y_e^*]$. In this case, we just copy the information of each entry $y_h \in \tau[y_e]$ to the corresponding entry in $\tau[y_e^*]$. So assume that $|Z_e^*| = |Z_e| - 1$ and that there is an index a such that $Z_e^{(a-1)} = Z_e^{(a)} \setminus \{x\}$, where x is the forgotten vertex. (The other two cases are similar.) We write

$$Z_e^* = \left(Z_e^{*(1)}, \ldots, Z_e^{*(a-2)}, Z_e^{*(a)}, \ldots, Z_e^{*(s)}\right).$$

For $q < a-1$ and $q > a$, $\tau\left(y_e^{*(q)}\right) = \tau\left(y_e^{(q)}\right)$. For these sequences, we just copy the labels from every entry in $\tau\left(y_e^{(q)}\right)$ to the corresponding entry in $\tau\left(y_e^{*(q)}\right)$. For $q = a$, recall that $\tau\left(y_e^{*(a)}\right) = \tau\left(\tau\left(y_e^{(a-1)}\right) \circ \tau\left(y_e^{(a)}\right)\right)$. In particular, we computed sequence $\tau\left(y_e^{*(a)}\right)$ by first constructing a sequence $y^\circ = \tau\left(y_e^{(a-1)}\right) \circ \tau\left(y_e^{(a)}\right)$ and then computing $\tau\left(y_e^{*(a)}\right) = \tau(y^\circ)$. Every entry in y° inherits its labels from its corresponding entry in $\tau\left(y_e^{(a-1)}\right)$ or $\tau\left(y_e^{(a)}\right)$, respectively. We derived sequence $\tau\left(y_e^{*(a)}\right)$ from y° by means of two operations: duplicate removal and typical operations. As a result, every entry $y_h \in \tau\left(y_e^{*(a)}\right)$ corresponds to an interval $y_k^\circ, \ldots, y_l^\circ$ of entries in y°. We compute $\kappa(y_h) = \sum_{f=k}^{l} \kappa(y_f^\circ)$, concatenate graphs $\mathcal{G}(y_k^\circ), \ldots, \mathcal{G}(y_l^\circ)$ to obtain the graph $\mathcal{G}(y_h)$, and store the source and sink of this graph with entry y_h. The concatenation of graphs $\mathcal{G}(y_k^\circ), \ldots, \mathcal{G}(y_l^\circ)$ is done as follows: Consider two graphs $\mathcal{G}(y_f^\circ)$ and $\mathcal{G}(y_{f+1}^\circ)$, and let τ and σ be the sink of $\mathcal{G}(y_f^\circ)$ and source of $\mathcal{G}(y_{f+1}^\circ)$, respectively. Then we add edges (τ, σ) and (σ, τ) to $\mathcal{G}(y_h)$, compute $R_\tau \leftarrow R_\tau \setminus L_\sigma$ and $L_\sigma \leftarrow L_\sigma \setminus R_\tau$, and assign color $(c(\tau), c(\sigma))$ to edge (τ, σ) and color $(\bar{c}(\sigma), \bar{c}(\tau))$ to edge (σ, τ).

If $\mathcal{T} \neq \mathcal{T}^*$, there is a leaf a of $\mathcal{T}$ that is not a node of $\mathcal{T}^*$. Let b be the neighbor of a in $\mathcal{T}$. If b is a node of $\mathcal{T}^*$, all we have to do is detach edge (a, b) from $\mathcal{T}$. For any other trunk edge, neither the associated path decomposition nor the characteristic of this path decomposition changes; hence, we simply copy labels between corresponding entries in the typical lists, as described for the case $\mathcal{T} = \mathcal{T}^*$ and $|Z_e| = |Z_e^*|$. The detachment of edge (a, b) involves the computation of a flippable graph $\mathcal{G}((\mathsf{a}, \mathsf{b}))$ that represents path decomposition $Y_{(\mathsf{a},\mathsf{b})}$ and the creation of a link record that connects the copy of b in $Y_{(\mathsf{a},\mathsf{b})}$ to the copy that remains in $\mathcal{T}^*$; we add this link record to the link list $\mathcal{L}$. To construct graph $\mathcal{G}((\mathsf{a}, \mathsf{b}))$, we concatenate graphs $\mathcal{G}(y)$, for all typical sequences $\tau\left(y_{(\mathsf{a},\mathsf{b})}^{(q)}\right)$ in $\tau[y_e]$ and all entries y in these sequences. This concatenation is done as described for the case $\mathcal{T} = \mathcal{T}^*$ and $|Z_e^*| = |Z_e| - 1$ above. To add the link between the copy of b in $Y_{(\mathsf{a},\mathsf{b})}$ and the copy that remains in $\mathcal{T}^*$, consider any edge e other than (a, b) incident to b. Assume that edge (a, b) is directed from a to b, and that edge e is directed away from b. Let y be the first entry in $\tau\left(y_e^{(1)}\right)$, let σ be the source of graph $\mathcal{G}(y)$, and let τ be the sink of graph $\mathcal{G}((\mathsf{a}, \mathsf{b}))$. Then we add the link (σ, τ) to the link list $\mathcal{L}$.

If b is not a node of $\mathcal{T}^*$, let c and d be the other two neighbors of b in $\mathcal{T}$. In $\mathcal{T}^*$, edges (c,b) and (b,d) are replaced by an edge (c,d). In order to compute the information associated with the entries of the typical lists of all trunk edges, we proceed as if b were a node of $\mathcal{T}^*$—i.e., we detach edge (a,b) from the trunk, as described above, and leave the information associated with the other edges unchanged—and then we compute the inforamtion associated with the entries of typical list $\tau[y^*_{(\mathsf{c},\mathsf{d})}]$. Note that the characteristic of path decomposition $Y^*_{(\mathsf{c},\mathsf{d})}$ is obtained in three steps: First concatenate interval models $Z_{(\mathsf{c},\mathsf{b})}$ and $Z_{(\mathsf{b},\mathsf{d})}$, and concatenate $\tau[y_{(\mathsf{c},\mathsf{b})}]$ and $\tau[y_{(\mathsf{b},\mathsf{d})}]$; then remove consecutive duplicates from $Z_{(\mathsf{c},\mathsf{b})} \circ \mathcal{Z}_{(\mathsf{b},\mathsf{d})}$ and concatenate the corresponding typical sequences in $\tau[y_{(\mathsf{c},\mathsf{b})}] \circ \tau[y_{(\mathsf{b},\mathsf{d})}]$; finally, apply duplicate removals and typical operations to each of the concatenations of typical sequences. The concatenation of interval models $Z_{(\mathsf{c},\mathsf{b})}$ and $Z_{(\mathsf{b},\mathsf{d})}$ and of typical lists $\tau[y_{(\mathsf{c},\mathsf{b})}]$ and $\tau[y_{(\mathsf{b},\mathsf{d})}]$ do not require any updates of the graphs associated with the entries in $\tau[y_{(\mathsf{c},\mathsf{b})}] \circ \tau[y_{(\mathsf{b},\mathsf{d})}]$. The removal of duplicates from $Z_{(\mathsf{c},\mathsf{b})} \circ Z_{(\mathsf{b},\mathsf{d})}$ and the resulting application of duplicate removals and typical operations to the affected typical sequences can be handled as described above for the case $\mathcal{T} - \mathcal{T}^*$ and $|Z^*_e| = |Z_e| - 1$.

In the discussion so far, we have ignored a detail that has to be taken care of when concatenating characteristics. The problem is that the characteristics and thus the corresponding path decompositions may have opposite directions; that is, $\tau[y_{(\mathsf{c},\mathsf{b})}]$ may in fact be represented as a list $\tau[y_{(\mathsf{b},\mathsf{c})}]$ sorted from b to c, while $\tau[y_{(\mathsf{b},\mathsf{d})}]$ is sorted from b to d. This means that the graphs $\mathcal{G}(y_h)$ stored with the entries of $\tau[y_{(\mathsf{b},\mathsf{c})}]$ represent the path decomposition $Y_{(\mathsf{b},\mathsf{c})}$ directed from b to c; but we need to direct it from c to b, in order to concatenate path decomposition $Y_{(\mathsf{c},\mathsf{b})}$ and $Y_{(\mathsf{b},\mathsf{d})}$ correctly.

Turning the interval model $Z_{(\mathsf{b},\mathsf{c})}$ and the typical list $\tau[y_{(\mathsf{b},\mathsf{c})}]$ around is easy: Let $Z_{(\mathsf{b},\mathsf{c})} = \left(Z^{(1)}_{(\mathsf{b},\mathsf{c})}, \ldots, Z^{(s)}_{(\mathsf{b},\mathsf{c})}\right)$; let $\tau[y_{(\mathsf{b},\mathsf{c})}] = \left(\tau\left(y^{(1)}_{(\mathsf{b},\mathsf{c})}\right), \ldots, \tau\left(y^{(s)}_{(\mathsf{b},\mathsf{c})}\right)\right)$; and let $\tau\left(y^{(q)}_{(\mathsf{b},\mathsf{c})}\right) = (y_1, \ldots, y_{r_q})$, for all $1 \le q \le s$. Then we define $Z_{(\mathsf{c},\mathsf{b})} = \left(Z^{(s)}_{(\mathsf{b},\mathsf{c})}, Z^{(s-1)}_{(\mathsf{b},\mathsf{c})}, \ldots, Z^{(1)}_{(\mathsf{b},\mathsf{c})}\right)$ and $\tau[y_{(\mathsf{c},\mathsf{b})}] = \left(\tau\left(y^{(1)}_{(\mathsf{c},\mathsf{b})}\right), \ldots, \tau\left(y^{(s)}_{(\mathsf{c},\mathsf{b})}\right)\right)$, where $\tau\left(y^{(s-q+1)}_{(\mathsf{c},\mathsf{b})}\right) = (y_{r_q}, y_{r_q - 1}, \ldots, y_1)$, for $1 \le q \le s$.

In order to complete the flip of the path decomposition $Y_{(\mathsf{b},\mathsf{c})}$, we flip all graphs $\mathcal{G}(y_h)$ along edge (b,c) by changing the colors of their sources and sinks; exchanging the source and sink pointers of every entry y_h of $\tau[y_{(\mathsf{c},\mathsf{b})}]$; and (conceptually) exchanging sets L_α and R_α, for all vertices $\alpha \in \mathcal{G}(y_h)$. The latter operation may be quite costly if performed explicitly, because graph $\mathcal{G}(y_h)$ may be large. In order to avoid this cost, we perform this exchange of sets L_α and R_α only for the source σ and sink τ of $\mathcal{G}(y_h)$. This is sufficient because subsequent augmentations of graph $\mathcal{G}(y_h)$ rely only on the correct contents of sets L_σ and R_τ, and the extraction of the path decomposition represented by $\mathcal{G}(y_h)$, described in Section 6.2.5, treats sets L_α and R_α as a single set $S_\alpha = L_\alpha \cup R_\alpha$.

Introduce node: Let i be an introduce node with child j, and let $C_\mathsf{i} = (\mathcal{T}^*, (Z^*_e)_{e\in\mathcal{T}^*}, (\tau[y^*_e])_{e\in\mathcal{T}^*})$ and $C_\mathsf{j} = (\mathcal{T}, (Z_e)_{e\in\mathcal{T}}, (\tau[y_e])_{e\in\mathcal{T}})$. Let x be the introduced vertex. Again, we have to distinguish two cases, depending on whether or not $\mathcal{T} = \mathcal{T}^*$. If $\mathcal{T} = \mathcal{T}^*$, consider an edge $e \in \mathcal{T}$. If $|Z_e| = |Z^*_e|$, then $\tau[y_e] = \left(\tau\left(y^{(1)}_e\right), \ldots, \tau\left(y^{(s)}_e\right)\right)$ and $\tau[y^*_e] = \left(\tau\left(y^{(1)}_e\right), \ldots, \tau\left(y^{(a-1)}_e\right), 1 + \tau\left(y^{(a)}_e\right), \ldots, 1 + \tau\left(y^{(b)}_e\right), \tau\left(y^{(b+1)}_e\right), \ldots, \tau\left(y^{(s)}_e\right)\right)$, for appropriate indices a and b. To update the path decomposition represented by the graphs $\mathcal{G}(y_h)$ along edge e, we first copy the information from each entry in $\tau[y_e]$ to the corresponding entry in $\tau[y^*_e]$. Now let $y_h \in \tau\left(y^{(q)}_e\right)$,

$a \le q \le b$, and let σ and τ be the source and sink nodes stored with y_h. Then we add vertex x to sets L_σ and R_τ, thereby adding x to every set in the path decomposition represented by graph $\mathcal{G}(y_h)$.

Now consider the case when $|Z_e^*| = |Z_e| + 1$ and $Z_e^{*(a-1)} = Z_e^{*(a)} \setminus \{x\}$. (The other two cases are similar.) In this case, $\tau[y_e] = \left(\tau\left(y_e^{(1)}\right), \ldots, \tau\left(y_e^{(s)}\right)\right)$ and

$$\tau[y_e^*] = \left(\tau\left(y_e^{(1)}\right), \ldots, \tau\left(y_e^{(a-1)}\right), \tau_1, 1+\tau_2, 1+\tau\left(y_e^{(a+1)}\right), \ldots \right.$$
$$\left. \ldots, 1+\tau\left(y_e^{(b)}\right), \tau\left(y_e^{(b+1)}\right), \ldots, \tau\left(y_e^{(s)}\right)\right),$$

where (τ_1, τ_2) is a split of $\tau\left(y_e^{(a)}\right)$. For all typical sequences except $\tau\left(y_e^{(a)}\right)$, we proceed as in the case $|Z_e| = |Z_e^*|$. If (τ_1, τ_2) is a type-two split, there is a one-to-one correspondence between the elements in τ_1 and τ_2 and the elements in $\tau\left(y_e^{(a)}\right)$. We Copy the necessary information and then add vertex x to sets L_σ and R_τ, for the source σ and sink τ of every graph $\mathcal{G}(y_h)$, $y_h \in \tau_2$. If (τ_1, τ_2) is a type-one split, then $\tau\left(y_e^{(a)}\right) = (y_1, \ldots, y_r)$, $\tau_1 = (y_1, \ldots, y_f)$, and $\tau_2 = (y_f, \ldots, y_r)$. For $1 \le h < f$, we copy the information for y_h from $\tau\left(y_e^{(a)}\right)$ to τ_1. For every elementy_h in τ_2, we copy the information from $\tau\left(y_e^{(a)}\right)$ to τ_2 and add x to the sets L_σ and R_τ associated with its source and sink. Finally, for the entry y_f in τ_1, we create a new graph $\mathcal{G}(y_f)$ with a single node $\sigma = \tau$ and set $L_\sigma \leftarrow R_\tau \leftarrow L_{\sigma'} \setminus \{x\}$, $\rho_\sigma \leftarrow 1$, and $c(\sigma) = c(\tau) = \mathsf{red}$, where σ' is the source node stored with y_f in τ_2; we set $\kappa(y_f) \leftarrow 1$ in τ_1.

If $\mathcal{T}^* \ne \mathcal{T}$, $\mathcal{T}^*$ contains one edge from a leaf a to its neighbor b that is not in $\mathcal{T}$. Moreover, node b may not be in $\mathcal{T}$. If b is in $\mathcal{T}$, the information stored with the edges that are in both $\mathcal{T}$ and $\mathcal{T}^*$ does not change. If b is not in $\mathcal{T}$, let c and d be the other two neighbors of b in $\mathcal{T}^*$. The characteristics of path decompositions $Y_{(\mathsf{c},\mathsf{b})}$ and $Y_{(\mathsf{b},\mathsf{d})}$ are obtained from the characteristic of path decomposition $Y_{(\mathsf{c},\mathsf{d})}$ using a type-one split. The necessary information stored with edges (c,b) and (b,d) can be computed using the procedure for type-one splits described above, excluding the addition of vertex x to the source and sink sets of the graphs associated with the entries in the second sequence. Finally, let $e = (\mathsf{a},\mathsf{b})$ and $\tau[y_e] = \left(\tau\left(y_e^{(1)}\right), \ldots, \tau\left(y_e^{(s)}\right)\right)$. Then $\tau\left(y_e^{(q)}\right) = \left(\left|Z_e^{(q)}\right|\right)$ and $\tau\left(y_e^{(q)}\right) = (y_1)$. We define the graph $\mathcal{G}(y_1)$ to consist of a single node $\sigma = \tau$ with $L_\sigma = R_\tau = Z_e^{(q)}$, $\rho_\sigma = 1$, and $c(\sigma) = c(\tau) = \mathsf{red}$; $\kappa(y_1) = 1$.

Join node: Let i be a join node with children j and k. The corresponding characteristics are $C_\mathsf{i} = (\mathcal{T}, (Z_e)_{e\in\mathcal{T}}, (\tau[y_e])_{e\in\mathcal{T}})$, $C_\mathsf{j} = (\mathcal{T}, (Z_e)_{e\in\mathcal{T}}, (\tau[a_e])_{e\in\mathcal{T}})$, and $C_\mathsf{k} = (\mathcal{T}, (Z_e)_{e\in\mathcal{T}}, (\tau[b_e])_{e\in\mathcal{T}})$. For every edge $e \in \mathcal{T}$, we have to compute the information stored with $\tau[y_e]$ from the information stored with $\tau[a_e]$ and $\tau[b_e]$.

A complication that we have to deal with is the fact that for an edge $e = (\mathsf{v},\mathsf{w})$ in the trunk $\mathcal{T}$, $\tau[a_e]$ may be sorted from v to w and $\tau[b_e]$ may be sorted from w to v. The same is then true for the graphs $\mathcal{G}(a_h)$ and $\mathcal{G}(b_h)$ associated with the entries in these lists. If this is the case, we flip one of the lists including all its associated graphs using the same procedure as described for forget nodes. So we can assume for the sake of simplicity that all edge lists $\tau[a_e]$ and $\tau[b_e]$ are sorted in the same directed.

Every sequence $\tau\left(y_e^{(q)}\right)$ is computed from two sequences $\tau\left(a_e^{(q)}\right)$ and $\tau\left(b_e^{(q)}\right)$. Let $a^* = (a_1, \dots, a_n)$, $\tau\left(b_e^{(q)}\right) = (b_1, \dots, b_{n'})$, $y^\circ = (y_1^\circ, \dots, y_c^\circ)$, and $\tau\left(y_e^{(q)}\right) = \tau(y^\circ) = (y_1, \dots, y_m)$, as defined in the description of Phase 1.

Given the pointers $r(y_h)$, for all elements $y_h \in \tau\left(y_e^{(q)}\right)$, as computed in Phase 1, the information stored with every element $y_h \in \tau\left(y_e^{(q)}\right)$ can be computed from the information stored with the elements of y° similar to the processing of a forget node, as only typical operations and duplicate removals are involved in this computation.

We describe how to compute the appropriate information for every element $y_f^\circ \in y^\circ$. Recall that every element y_f° stores two pointers $p(y_f^\circ)$ and $q(y_f^\circ)$ so that $y_f^\circ = a_{p(y_f^\circ)} + b_{q(y_f^\circ)}$. We first construct two graphs $\mathcal{G}_a(y_f^\circ)$ and $\mathcal{G}_b(y_f^\circ)$ defined as follows: If $f < c$ and $p(y_f^\circ) = p(y_{f+1}^\circ)$, then $\mathcal{G}_a(y_f^\circ)$ consists of a single node $\sigma = \tau$ with $L_\sigma = R_\tau = L_{\sigma'}$, $\rho_\sigma = 1$, and $c(\sigma) = \mathsf{red}$, where σ' is the source of $\mathcal{G}(a_{p(y_f^\circ)})$. In this case, we set $\kappa_a(y_f^\circ) = 1$. If $f = c$ or $p(y_f^\circ) < p(y_{f+1}^\circ)$, then $\mathcal{G}_a(y_f^\circ) = \mathcal{G}(a_{p(y_f^\circ)})$ and $\kappa_a(y_f^\circ) = \kappa(a_{p(y_f^\circ)})$. The graph $\mathcal{G}_b(y_f^\circ)$ and the value $\kappa_b(y_f^\circ)$ are defined analogously.

To compute graph $\mathcal{G}(y_h)$, we first compute the length of the path decomposition represented by $\mathcal{G}(y_h)$ and stretch the path decompositions represented by graphs $\mathcal{G}_a(y_h)$ and $\mathcal{G}_b(y_h)$ to this length: The length of the path decomposition represented by graph $\mathcal{G}(y_h)$ is $\kappa(y_f^\circ) = \max\{\kappa_a(y_f^\circ), \kappa_b(y_f^\circ)\}$. Let σ_a be the source of $\mathcal{G}_a(y_f^\circ)$ and σ_b be the source of $\mathcal{G}_b(y_f^\circ)$. Then we change ρ_{σ_a} and ρ_{σ_b} to $\rho_{\sigma_a} + (\kappa(y_f^\circ) - \kappa_a(y_f^\circ))$ and $\rho_{\sigma_b} + (\kappa(y_f^\circ) - \kappa_b(y_f^\circ))$, respectively. Next we add two new vertices σ and τ and edges (σ, σ_a), (σ_a, σ), (σ, σ_b), (σ_b, σ), (τ_a, τ), (τ, τ_a), (τ_b, τ), and (τ, τ_b) to the union of graphs $\mathcal{G}_a(y_h)$ and $\mathcal{G}_b(y_h)$, where τ_a and τ_b are the sinks of $\mathcal{G}_a(y_f^\circ)$ and $\mathcal{G}_b(y_f^\circ)$, respectively. We define $L_\sigma \leftarrow L_{\sigma_a} \cup L_{\sigma_b}$, $R_\sigma \leftarrow \emptyset$, $L_{\sigma_a} \leftarrow L_{\sigma_b} \leftarrow \emptyset$, $L_\tau \leftarrow \emptyset$, $R_\tau \leftarrow R_{\tau_a} \cup R_{\tau_b}$, $R_{\tau_a} \leftarrow R_{\tau_b} \leftarrow \emptyset$, $\rho_\sigma \leftarrow 0$, $\rho_\tau \leftarrow 0$, and $c(\sigma) = c(\tau) = \mathsf{red}$. The new edges are colored as follows:

$$\begin{aligned}
c((\sigma, \sigma_a)) &= (c(\sigma), c(\sigma_a)), & c((\sigma_a, \sigma)) &= (\bar{c}(\sigma_a), \bar{c}(\sigma)),\\
c((\sigma, \sigma_b)) &= (c(\sigma), c(\sigma_b)), & c((\sigma_b, \sigma)) &= (\bar{c}(\sigma_b), \bar{c}(\sigma)),\\
c((\tau_a, \tau)) &= (c(\tau_a), c(\tau)), & c((\tau, \tau_a)) &= (\bar{c}(\tau), \bar{c}(\tau_a)),\\
c((\tau_b, \tau)) &= (c(\tau_b), c(\tau)), & c((\tau, \tau_b)) &= (\bar{c}(\tau), \bar{c}(\tau_b)).
\end{aligned}$$

Once we have applied these rules bottom-up in T, tree decomposition $\mathcal{E}$ is represented as a collection of path decompositions. Each such path decomposition is either part of the filled trunk and hence represented by a graph $\mathcal{G}(y_h)$; or it is not part of the filled trunk, in which case it is represented by a graph $\mathcal{G}(e)$ that was constructed whon detaching edge e from the filled trunk. In order to avoid having to deal with the trunk of $\mathcal{E}$ in a specialized manner in Phase 4 of the algorithm, we decompose it into path decompositions by detaching its edges bottom-up and using the same procedure as for a forget node to construct the graph $\mathcal{G}(e)$ corresponding to the detached edge e. Note that each detachment introduces one extra node into the tree decomposition. Since the trunk of $\mathcal{E}_\mathsf{r}$ has constant size, we introduce only $\mathcal{O}(1)$ extra nodes. The following lemma shows that Phase 3 computes the correct input for Phase 4.

Lemma 17 *Let* i *be a node in* T, *let* $C_\mathsf{i} = (\mathcal{T}, (Z_e)_{e\in\mathcal{T}}, (\tau[y_e])_{e\in\mathcal{T}})$ *be the characteristic stored at* i, *let* e *be an edge of* $\mathcal{T}$, *let* $\tau\left(y_e^{(q)}\right)$ *be a typical sequence in* $\tau[y_e]$, *and let* $y_h \in \tau\left(y_e^{(q)}\right)$. *Then*

$\mathcal{G}(y_h)$ *has Properties G1–G5, and coloring the source* σ *of* $\mathcal{G}(y_h)$ *with color* $c(\sigma)$ *directs the edges of* $\mathcal{G}(y_h)$ *from the source* σ *to the sink* τ *of* $\mathcal{G}(y_h)$ *and colors* τ *with color* $c(\tau)$*.*

Proof. The proof is by induction on the size of the subtree T_i rooted at node i. If $|T_\mathsf{i}| = 1$, i.e., i is a start node, then every graph $\mathcal{G}(y_h)$ consists of a single vertex v with $L_v = R_v = Z_e^{(q)} = Y_e^{(q)}$ and $\rho_v = 1$. On the other hand $Y_{y_h} = \left(Z_e^{(q)}\right)$. Properties G1–G5 are now easily verified. Moreover, as $\mathcal{G}(y_h)$ has no edges, it is trivially true that coloring $\sigma = \tau$ with color $c(\sigma)$ directs the edges in $\mathcal{G}(y_h)$ from σ to τ and colors τ with color $c(\tau)$.

If $|T_\mathsf{i}| > 1$, i is an internal node of T, i.e., a join, introduce, or forget node. If i is an introduce or forget node with child j, $|T_\mathsf{j}| < |T_i|$. Hence, by the induction hypothesis, all graphs $\mathcal{G}(y_h)$ associated with the entries y_h of the typical lists characteristic C_j have Properties G1–G5. Analogously, if i is a join node with children j and k, $|T_\mathsf{j}| < |T_\mathsf{i}|$ and $|T_\mathsf{k}| < |T_\mathsf{i}|$; so the graphs $\mathcal{G}(y_h)$ stored with characteristics C_j and C_k have Properties G1–G5. We show that, for all three node types, the algorithm constructs graphs $\mathcal{G}(y_h)$ for the elements of the typical sequences in C_i correctly. In order to verify that these graphs $\mathcal{G}(y_h)$ have Property G1, i.e., that $\mathcal{G}(y_h)$ is a flippable DAG, we restrict our attention to Property F3 of flippable DAGs, as the other two properties are easily verified.

Forget node: If i is a forget node with child j, we distinguish two cases again. If $\mathcal{T} = \mathcal{T}^*$, the path decomposition Y_e corresponding to every edge $e \in \mathcal{T}$ does not change. If $|Z_e| = |Z_e^*|$, then $\tau[y_e] = \tau[y_e^*]$. Hence, for every element $y_h \in \tau\left(y_e^{*(q)}\right) = \tau\left(y_e^{(q)}\right)$, $\mathcal{G}(y_h)$ has Properties G1–G5, by the induction hypothesis. So assume that $|Z_e| \neq |Z_e^*|$. Again, we restrict our attention to the case $|Z_e| = |Z_e^*| + 1$ and $Z_e^{(a-1)} = Z_e^{(a)} \setminus \{x\}$.

Consider the path decompositions $Y_e^{(a-1)}$ and $Y_e^{(a)}$ that correspond to intervals $Z_e^{(a-1)}$ and $Z_e^{(a)}$. By the induction hypothesis, $Y_e^{(a-1)} = Y_{y_1} \circ Y_{y_2} \circ \cdots \circ Y_{y_m}$, where $\tau\left(y_e^{(a-1)}\right) = (y_1, \ldots, y_m)$, and for every $1 \leq h \leq m$, $\mathcal{G}(y_h)$ has Properties G1–G5, i.e., represents Y_{y_h} correctly. Analogously, $Y_e^{(a)} = Y_{y'_1} \circ Y_{y'_2} \circ \cdots \circ Y_{y'_n}$, where $\tau\left(y_e^{(a)}\right) = (y'_1, \ldots, y'_n)$, and for every $1 \leq h \leq n$, $\mathcal{G}(y'_h)$ has Properties G1–G5. Thus, every graph $\mathcal{G}(y_f^\circ)$, $1 \leq f \leq m+n$, has Properties G1–G5, and $Y_e^{*(a)} = Y_{y_1^\circ} \circ \cdots \circ Y_{y_{m+n}^\circ}$, where $y^\circ = \tau\left(y_e^{(a-1)}\right) \circ \tau\left(y_e^{(a)}\right) = (y_1^\circ, \ldots, y_{m+n}^\circ)$. Every element $y_h \in \tau\left(y_e^{*(a)}\right)$ corresponds to an interval $y_k^\circ, \ldots, y_l^\circ$ of elements in y°, and we compute $\mathcal{G}(y_h) = \mathcal{G}(y_k^\circ) \circ \cdots \circ \mathcal{G}(y_l^\circ)$. Let σ and τ be the source and sink of $\mathcal{G}(y_h)$. First we prove Property G1 and that coloring σ with color $c(\sigma)$ directs all edges in $\mathcal{G}(y_h)$ from σ to τ and colors τ with color $c(\tau)$. We do this by induction on the number of concatenated graphs; that is, we consider graphs $\mathcal{G}_j$, $0 \leq j \leq l-k$, defined as $\mathcal{G}_j = \mathcal{G}(y_k^\circ) \circ \cdots \circ \mathcal{G}(y_{k+j}^\circ)$.

For $j = 0$, the claim holds, by the induction hypothesis (on $|T_\mathsf{i}|$). So assume that $j > 0$. To show that $\mathcal{G}_j$ is a flippable DAG, we have to prove that the coloring of $\mathcal{G}_j$ is independent of the spanning tree, once a color for σ has been chosen. This is true for $\mathcal{G}_{j-1}$, by the induction hypothesis. Let τ_{j-1} be the sink of $\mathcal{G}_{j-1}$. Then the color of τ_{j-1} depends only on the color of σ. The same is true for the source σ_j of $\mathcal{G}(y_{k+j}^\circ)$ because every spanning tree of $\mathcal{G}_j$ must contain edge $\{\tau_{j-1}, \sigma_j\}$. By the induction hypothesis (on $|T_\mathsf{i}|$), the colors of all vertices in $\mathcal{G}(y_{k+j})$ are fixed, once the color of σ_j is fixed. Hence, the coloring of $\mathcal{G}_j$ is independent of the spanning tree chosen for $\mathcal{G}_j$, and $\mathcal{G}_j$ is a flippable DAG. In order to show that coloring σ with color $c(\sigma)$ colors the sink τ_j of $\mathcal{G}_j$ with color $c(\tau_j)$ and directs all edges in $\mathcal{G}_j$ from σ to τ_j, we make the following observations: By the induction hypothesis, coloring σ with color $c(\sigma)$ colors τ_{j-1} with color $c(\tau_{j-1})$. Hence, σ_j is colored with color $c(\sigma_j)$, because edge $\{\tau_{j-1}, \sigma_j)\}$ has color $(c(\tau_{j-1}), c(\sigma_j))$. By the induction hypothesis (on $|T_\mathsf{i}|$), τ_j

is thus colored with color $c(\tau_j)$. Also, coloring σ with color $c(\sigma)$ directs all edges in $\mathcal{G}_{j-1}$ from σ to τ_{j-1}; chooses edge (τ_{j-1}, σ_j) from the two possible edges between τ_{j-1} and σ_j; and directs all edges in $\mathcal{G}(y_{k+j})$ from σ_j to τ_j, because σ_j receives color $c(\sigma_j)$.

Properties G2 and G3 are readily verified. We split the proof of Property G4 into two parts. The first part deals with vertices x that appear in only one subgraph $\mathcal{G}(y_f^\circ)$ of $\mathcal{G}(y_h)$. By the induction hypothesis, there are two unique vertices $\mu(x)$ and $\nu(x)$ in $\mathcal{G}(y_f^\circ)$ such that $x \in L_{\mu(x)}$ and $x \in R_{\nu(x)}$. Since $\mathcal{G}(y_f^\circ)$ is the only graph containing x, these are the only such vertices in $\mathcal{G}(y_h)$.

If a vertex x occurs in more than one subgraph $\mathcal{G}(y_f^\circ)$, let $\mathcal{G}(y_p^\circ)$ and $\mathcal{G}(y_q^\circ)$ be the leftmost and rightmost such subgraphs. As the concatenation of path decompositions $Y_p^\circ, \ldots, Y_q^\circ$ is itself a path decomposition, x is contained in all sets of path decompositions Y_f°, $p < f < q$. Hence, if σ_f and τ_f are the source and sink of graph $\mathcal{G}(y_f^\circ)$, then $x \in L_{\sigma_f}$, for all $p < f \leq q$, and $x \in R_{\tau_f}$, for all $p \leq f < q$. In addition, there is a unique vertex $\mu(x) \in \mathcal{G}(y_p^\circ)$ such that $x \in L_{\mu(x)}$ and a unique vertex $\nu(x) \in \mathcal{G}(y_q^\circ)$ such that $x \in R_{\nu(x)}$. By the induction hypothesis, these are the only vertices α in $\mathcal{G}(y_p^\circ), \ldots, \mathcal{G}(y_q^\circ)$ such that $x \in L_\alpha$ or $x \in R_\alpha$. Our construction procedure removes x from all adjacent sink and source vertices in this set of vertices, so that $\mu(x) \in \mathcal{G}(y_p^\circ)$ and $\nu(x) \in \mathcal{G}(y_q^\circ)$ are the only remaining vertices with $x \in L_{\mu(x)}$ and $x \in R_{\nu(x)}$.

The proof of Property G5 distinguishes the same two cases as the proof of Property G4. If x is contained in only one graph $\mathcal{G}(y_f^\circ)$, then the property follows immediately from the induction hypothesis and the fact that Y_{y_h} is the concatenation of $Y_{y_k^\circ}, \ldots, Y_{y_l^\circ}$ and $\mathcal{G}(y_h)$ is the concatenation of graphs $\mathcal{G}(y_k^\circ), \ldots, \mathcal{G}(y_l^\circ)$. If x is contained in more than one graph $\mathcal{G}(y_f^\circ)$, let $I_{\mathcal{G}(y_p^\circ)}(x) = [c, d]$ and $I_{\mathcal{G}(y_q^\circ)}(x) = [c', d']$. The leftmost interval $I(\alpha)$, $\alpha \in \mathcal{G}(y_p^\circ)$, contained in $I_{\mathcal{G}(y_p^\circ)}(x)$ is $I(\mu(x))$; the rightmost interval $I(\beta)$, $\beta \in \mathcal{G}(y_q^\circ)$, contained in $I_{\mathcal{G}(y_q^\circ)}(x)$ is $I(\nu(x))$. We have already observed that x must be contained in all sets between Y_c and $Y_{d'}$. Hence, the interval $I_{\mathcal{G}(y_h)}(x) = [c, d']$ as defined by the two nodes $\mu(x)$ and $\nu(x)$ is correct.

If $\mathcal{T} \neq \mathcal{T}^*$, there is a leaf $\mathsf{a} \in \mathcal{T}$ that is not in $\mathcal{T}^*$. If the neighbor, b, of a is a node of $\mathcal{T}^*$, the path decomposition for every edge $e \in \mathcal{T}^*$ remain the same. Hence, by leaving the graph associated with every entry $y_h \in \tau\left(y_e^{(q)}\right)$ unchanged, Properties G1–G5 are preserved.

If b is not a node of $\mathcal{T}^*$, we have to merge path decompositions $Y_{(\mathsf{c},\mathsf{b})}$ and $Y_{(\mathsf{b},\mathsf{d})}$ into a path decomposition $Y_{(\mathsf{c},\mathsf{d})}$, where c and d are the other two neighbors of b in $\mathcal{T}$. Once we have guaranteed that the directions of typical lists $\tau[y_{(\mathsf{c},\mathsf{b})}]$ and $\tau[y_{(\mathsf{b},\mathsf{d})}]$ and of the corresponding graphs $\mathcal{G}(y_h)$ match, the lemma can be shown similarly to the argument for the case $\mathcal{T} = \mathcal{T}^*$, as the same operations are involved in computing path decompositions Y_h and graphs $\mathcal{G}(y_h)$.

In order to show that we flip graph $\mathcal{G}(y_h)$ correctly, for every $y_h \in \tau\left(y_{(\mathsf{b},\mathsf{c})}^{(q)}\right)$, observe that by the induction hypothesis, coloring its source σ with color $c(\sigma)$ directs the edges of $\mathcal{G}(y_h)$ from σ to τ and colors the sink τ of $\mathcal{G}(y_h)$ with color $c(\tau)$. This implies that coloring τ with color $\bar{c}(\tau)$ colors σ with color $\bar{c}(\sigma)$ and directs the edges of $\mathcal{G}(y_h)$ from τ to σ. In order to complete the flip, we exchange the roles of σ and τ as source and sink vertices, and exchange the roles of L_α and R_α, for every vertex $\alpha \in \mathcal{G}(y_h)$.

Introduce node: We discuss the different possible cases. First assume that $\mathcal{T} = \mathcal{T}^*$. In this case, we have to augment path decomposition Y_e to path decomposition Y_e^*, for every edge $e \in \mathcal{T}$, by adding the introduced vertex x to the appropriate sets in Y_e. If $|Z_e| = |Z_e^*|$, the typical list $\tau[y_e]$ remains structurally unchanged. The only change is the increase of all values in typical sequences $\tau\left(y_e^{*(i)}\right)$ by one, for all sets $Z_e^{*(i)}$ containing the introduced vertex x. This means that we have to introduce vertex x into every path decomposition Y_{y_h} represented by an element $y_h \in \tau\left(y_e^{*(i)}\right)$.

This is done by adding x to L_σ and R_τ, where σ and τ are the source and sink of $\mathcal{G}(y_h)$. Note that none of the graphs $\mathcal{G}(y_h)$ changes structurally. Therefore, Properties G1–G5 are readily verified. So consider the case when $|Z_e^*| = |Z_e| + 1$ and $Z_e^{*(a-1)} = Z_e^{*(a)} \setminus \{x\}$, for some a. (The other two cases are similar.)

Again, we write $\tau[y_e] = \left(\tau\left(y_e^{(1)}\right), \ldots, \tau\left(y_e^{(a-2)}\right), \tau\left(y_e^{(a)}\right), \ldots, \tau\left(y_e^{(s)}\right)\right)$. Then every typical sequence $\tau\left(y_e^{*(q)}\right)$, except $\tau\left(y_e^{*(a-1)}\right)$ and $\tau\left(y_e^{*(a)}\right)$, is derived from the corresponding typical sequence $\tau\left(y_e^{(q)}\right)$ as in the case $|Z_e| = |Z_e^*|$. Also, the information stored with every element in such a sequence is computed in the same way as in the case $|Z_e| = |Z_e^*|$, so that Properties G1–G5 are easily verified. So consider the computation for $Z_e^{*(a-1)}$ and $Z_e^{*(a)}$. For these two intervals, $\tau\left(y_e^{*(a-1)}\right) = \tau_1$ and $\tau\left(y_e^{*(a)}\right) = 1 + \tau_2$, where (τ_1, τ_2) is a split of sequence $\tau\left(y_e^{(a)}\right)$. If the split is of type two, Properties G1–G5 are easily verified for all graphs $\mathcal{G}(y_h)$ associated with entries $y_h \in \tau\left(y_e^{*(a-1)}\right)$ or $y_h \in \tau\left(y_e^{*(a)}\right)$ because again these graphs are just copies of the graphs associated with the corresponding entries in $\tau\left(y_e^{(a)}\right)$, possibly augmented with the new vertex x. If the split is of type one, we have to consider the last entry y_f in τ_1. For this entry, we create a new one-vertex graph $\mathcal{G}(y_f)$. In path decomposition Y_e, this type-one split corresponds to duplicating the first set in the path decomposition corresponding to $(y_f, \ldots, y_r)$, which is just what we want. It is straightforward to verify Properties G1–G5 for $\mathcal{G}(y_h)$.

If $\mathcal{T} \neq \mathcal{T}^*$, we possibly split an edge (c, d) of $\mathcal{T}$ into two new edges (c, b) and (b, d) and then attach a new edge (a, b). Properties G1–G5 can be verified for graphs $\mathcal{G}(y_h)$ along edge (a, b) just as for start-nodes and for the graphs along all other edges as for the case $\mathcal{T} = \mathcal{T}^*$.

Join node: Finally, consider a join node i with children j and k. Note that the tree models of characteristics C_i, C_j, and C_k are the same. So we fix an edge e and discuss the computation for edge e.

The correctness of the flip possibly performed for some of the lists $\tau[a_e]$ or $\tau[b_e]$ can be established using the same arguments as for forget nodes. Properties G1–G5 are easily verified for graphs $\mathcal{G}_a(y_f^\circ)$ and $\mathcal{G}_b(y_f^\circ)$. So we prove Properties G1–G5 for graphs $\mathcal{G}(y_f^\circ)$. Once this is done, the lemma can be shown for graphs $\mathcal{G}(y_h)$ as for a forget node.

First we prove Property G1 and that coloring σ with color $c(\sigma)$ colors τ with color $c(\tau)$ and directs edges from σ to τ. Consider a spanning tree H of $\mathcal{G}(y_f^\circ)$. H consists of spanning trees for $\mathcal{G}_a(y_f^\circ)$ and $\mathcal{G}_b(y_f^\circ)$ as well as three of the four edges $\{\sigma, \sigma_a\}$, $\{\sigma, \sigma_b\}$, $\{\tau, \tau_a\}$, and $\{\tau, \tau_b\}$. First assume that both edges $\{\sigma, \sigma_a\}$ and $\{\sigma, \sigma_b\}$ are included in H. Then coloring σ with color $c(\sigma)$ colors σ_a and σ_b with colors $c(\sigma_a)$ and $c(\sigma_b)$, by the choice of the colors of edges (σ, σ_a) and (σ, σ_b). Hence, τ_a and τ_b receive colors $c(\tau_a)$ and $c(\tau_b)$, by the induction hypothesis. Regardless of whether $\{\tau, \tau_a\} \in H$ or $\{\tau, \tau_b\} \in H$, the coloring of edges (τ_a, τ) and (τ_b, τ) guarantees that τ receives color $c(\tau)$. As the coloring of all vertices in $\mathcal{G}_a(y_f^\circ)$ and $\mathcal{G}_b(y_f^\circ)$ is uniquely determined by the colors of σ_a and σ_b, all spanning trees containing edges $\{\sigma, \sigma_a\}$ and $\{\sigma, \sigma_b\}$ give the same coloring. If w.l.o.g. $\{\sigma, \sigma_b\} \notin H$, we can argue as above that vertices σ_a, τ_a, and τ are colored with colors $c(\sigma_a)$, $c(\tau_a)$, and $c(\tau)$, respectively. This implies that τ_b receives color $c(\tau_b)$, and, thus, σ_b receives color $c(\sigma_b)$. Hence, all spanning trees H give the same coloring, and $\mathcal{G}(y_f^\circ)$ is a flippable DAG. Also, coloring σ with color $c(\sigma)$ chooses edges (σ, σ_a) and (σ, σ_b) from the possible edges between σ and σ_a and σ_b. As σ_a and σ_b are colored with colors $c(\sigma_a)$ and $c(\sigma_b)$, all edges in $\mathcal{G}_a(y_f^\circ)$ and $\mathcal{G}_b(y_f^\circ)$ are directed from σ_a to τ_a and from σ_b to τ_b, respectively. Finally, as τ_a and τ_b are colored with colors $c(\tau_a)$ and

$c(\tau_b)$, edges (τ_a, τ) and (τ_b, τ) are chosen from the possible edges between τ and τ_a and τ_b. This proves that coloring σ with color $c(\sigma)$ directs all edges in $\mathcal{G}(y_f^\circ)$ from σ to τ.

Property G2 is easily verified. To prove Property G3, we argue as follows: By the induction hypothesis, the interval $I(\alpha)$ assigned to every node $\alpha \in \mathcal{G}_a(y_f^\circ)$ or $\alpha \in \mathcal{G}_b(y_f^\circ)$ is independent of the path chosen to compute this interval. This immediately implies that all nodes in $\mathcal{G}(y_f^\circ)$, except τ, have this property. To see that node τ has this property, observe that values $\rho(\sigma_a)$ and $\rho(\sigma_b)$ are adjusted so that for any $\sigma_a\tau_a$-path P_a in $\mathcal{G}_a(y_f^\circ)$ and any $\sigma_b\tau_b$-path P_b in $\mathcal{G}_b(y_f^\circ)$, $\sum_{\alpha \in P_a} \rho(\alpha) = \sum_{\alpha \in P_b} \rho(\alpha)$. This implies that $I(\tau)$ is independent of the choice of the $\sigma\tau$-path chosen to compute $I(\tau)$. The adjustment of $\rho(\sigma_a)$ or $\rho(\sigma_b)$ during the computation $\mathcal{G}(y_f^\circ)$ corresponds to the duplication of initial elements in path decomposition $A_{a_f^\circ}$ or $B_{b_f^\circ}$ before "overlaying" these two path decompositions to obtain path decomposition $Y_{y_f^\circ}$. Hence, for any $\sigma\tau$-path P in $\mathcal{G}(y_f^\circ)$, $\sum_{\alpha \in P} \rho(\alpha) = |Y_{y_f^\circ}|$, which shows that $\mathcal{G}(y_f^\circ)$ has Property G3. Property G5 follows immediately from the observation just made, that the increase of $\rho(\sigma_a)$ or $\rho(\sigma_b)$ reflects the "stretching" of the corresponding path decomposition $A_{a_f^\circ}$ or $B_{b_f^\circ}$.

In order to prove Property G4, observe that graphs $\mathcal{G}_a(y_f^\circ)$ and $\mathcal{G}_b(y_f^\circ)$ have this property, by the induction hypothesis. Since $\mathcal{G}_a(y_f^\circ)$ corresponds to a path decomposition $\bar{A}_f^\circ$ containing only vertices from G_j, and $\mathcal{G}_b(y_f^\circ)$ corresponds to a path decomposition $\bar{B}_f^\circ$ containing only vertices from G_k, every vertex that is shared by two sets in $\mathcal{G}_a(y_f^\circ)$ and $\mathcal{G}_b(y_f^\circ)$ is in X_i. So Property G4 holds for all vertices in path decomposition Y_f°, except those that are in X_i. Now observe that y_f° corresponds to a set $Z_e^{(q)}$ in the interval model of edge $e \in \mathcal{T}$, so that exactly the vertices in $Z_e^{(q)}$ are shared between G_j and $\mathcal{G}_\mathsf{k}$. Moreover, as path decompositions $\bar{A}_f^\circ$ and $\bar{B}_f^\circ$ are completely contained in the interval corresponding to $Z_e^{(q)}$, the vertices in $Z_e^{(q)}$ are contained in *every* set of $\bar{A}_f^\circ$ and $\bar{B}_f^\circ$. Hence, $Z_e^{(q)} \subseteq L_{\sigma_a}$, $Z_e^{(q)} \subseteq R_{\tau_a}$, $Z_e^{(q)} \subseteq L_{\sigma_b}$, and $Z_e^{(q)} \subseteq R_{\tau_a}$; and by the induction hypothesis, sets L_{σ_a}, R_{τ_a}, L_{σ_b}, and L_{τ_b} are the only sets in $\mathcal{G}_a(y_f^\circ)$ and $\mathcal{G}_b(y_f^\circ)$ containing vertices from $Z_e^{(q)}$. In order to obtain $\mathcal{G}(y_f^\circ)$ from $\mathcal{G}_a(y_f^\circ)$ and $\mathcal{G}_b(y_f^\circ)$, we define $L_\sigma \leftarrow L_{\sigma_a} \cup L_{\sigma_b}$ and $L_{\sigma_a} \leftarrow L_{\sigma_b} \leftarrow \emptyset$. Thus, every vertex in $Z_e^{(q)} \subseteq L_{\sigma_a} \cup L_{\sigma_b}$ occurs in exactly one set L_α, namely L_σ. We argue similarly that every vertex in $Z_e^{(q)}$ is contained only in set R_τ, which finishes the proof of Property G4. □

The following lemma bounds the size of the constructed graph $\mathcal{G}$.

Lemma 18 *Graph $\mathcal{G}$ has size $\mathcal{O}(N)$.*

Proof. It is easily verified that we introduce only a constant number of vertices into graph $\mathcal{G}$ at every node of T. As $|T| \leq 4N$, $\mathcal{G}$ has $\mathcal{O}(N)$ vertices. Also, it is easy to see that the in-degree and out-degree of every vertex are at most two. Hence, $\mathcal{G}$ has $\mathcal{O}(N)$ edges. □

6.2.5 Phase 4: Constructing the Tree Decomposition Explicitly

In this section, we show how to extract path decomposition Y_e from graph $\mathcal{G}(e)$, for every edge e removed from the trunk; we also show how to use the information stored in the link list $\mathcal{L}$ to construct tree U by joining these path decompositions. We start with the description of the method for extracting path decomposition Y_e from graph $\mathcal{G}(e)$.

First we compute the connected components $\mathcal{G}(e)$ of $\mathcal{G}$ and use Algorithm 1 to replace each such graph them with a DAG $\mathcal{G}'(e)$ that represents path decomposition Y_e. Let $\mathcal{G}'$ be the union of all these DAGs. We sort $\mathcal{G}'$ topologically.

Now consider a single path decomposition Y_e. We compute for every node $\alpha \in \mathcal{G}'(e)$, its interval $I(\alpha)$. This is easily done using time-forward processing [14]: given the interval $I(\alpha') = [a', b']$ of one of the in-neighbors of a node $\alpha \in \mathcal{G}'(e)$, the interval $I(\alpha)$ is defined as $I(\alpha) = [b'+1, b'+\rho(\alpha)]$.

Next we compute for every vertex $x \in G$, its interval $I(x)$. Recall that this is the smallest interval containing $I(\mu(x))$ and $I(\nu(x))$. Thus, for every node $\alpha \in \mathcal{G}'(e)$ and every vertex $x \in L_\alpha \cup R_\alpha$, we write a triple $(e, x, I(\alpha))$ to a list L. (This step can be incorporated into the time-forward processing step.) Then we sort this list by the first two components of its entries. This stores triples with the same first two components consecutively. For every pair (e, x), there are at most two triples $(e, x, I(\alpha))$ and $(e, x, I(\beta))$ that have e and x as their first two components. $I(x)$ is the smallest interval containing $I(\alpha)$ and $I(\beta)$. If there is only one such interval $I(\alpha)$, then $x \in L_\alpha \cap R_\alpha$; that is, x appears only in the sets associated with the nodes of the path decomposition corresponding to node α. Hence $I(x) = I(\alpha)$. Note that sets L_α and R_α are in fact handled as one set $S_\alpha = L_\alpha \cup R_\alpha$. This is the reason why we did not have to exchange sets L_α and R_α for vertices α involved in a graph flip in Phase 3.

Given intervals $I(x)$, for all vertices in the graph G_e represented by path decomposition Y_e, we create a list $\mathcal{S}$ of triples (e, a, x), for all $x \in G_e$ and $a \in I(x)$. This can be done in a single scan of the list of intervals $I(x)$. We sort list $\mathcal{S}$ by the first two components of its entries, thereby storing all entries with the same two components e and a consecutively. Pairs $(e, 1), \ldots, (e, |Y_e|)$ represent the nodes of path decomposition Y_e. We scan list $\mathcal{S}$ to construct vertex sets $X_{(e,a)} = \{x : (e, a, x) \in \mathcal{S}\}$. For every such vertex set with $a > 1$, we also add an edge $((e, a-1), (e, a))$ to the edge set of U.

Observe that we can perform this construction for all graphs $\mathcal{G}(e)$ at the same time, as we label every record in L and $\mathcal{S}$ that represents a node or vertex in path decomposition Y_e with the ID of edge e.

It remains to describe how to link path decompositions Y_e, in order to obtain the desired tree U. To do this, we need to translate the links $(\alpha, \beta) \in \mathcal{L}$ linking two graph $\mathcal{G}(e_1)$ and $\mathcal{G}(e_2)$ into edges between appropriate nodes of path decompositions Y_{e_1} and Y_{e_2}.

During the initial time-forward processing step computing intervals $I(\alpha) = [a, b]$, for all nodes $\alpha \in \mathcal{G}'(e)$, we add an entry (α, e, a) to a link translation table LT. We sort LT and the link list $\mathcal{L}$ by the first components of their entries. In a single scan of LT and $\mathcal{L}$, we translate entries (α, β) in $\mathcal{L}$ into entries $((e_1, a), \beta)$. We sort $\mathcal{L}$ by the second components of its entries and scan LT and $\mathcal{L}$ again, to translate entries $((e_1, a), \beta)$ into entries $((e_1, a), (e_2, b))$; we add these entries to the edge set of U. We summarize this section in the following theorem.

Theorem 6 *Given a graph $G = (V, E)$, two constants $k, \ell \in \mathbb{N}$, and a tree decomposition $\mathcal{D} = (\mathcal{X}, T)$ of width at most k and size $\mathcal{O}(N)$ for G, it is possible to decide in $\mathcal{O}(\mathrm{sort}(N))$ I/Os whether G has treewidth at most ℓ and, if so, compute a tree decomposition $\mathcal{E} = (\mathcal{Y}, U)$ of width at most ℓ and size $\mathcal{O}(N)$ for G.*

Proof. Phase 1 of Algorithm 5 takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os, by Theorem 5. Phase 2 is trivial. Phase 3 takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os: We send $\mathcal{O}(1)$ information along every edge of T when processing T bottom-up; the construction of graph $\mathcal{G}$ and link list $\mathcal{L}$ takes $\mathcal{O}(\mathrm{scan}(|\mathcal{G}|)) = \mathcal{O}(\mathrm{scan}(N))$ I/Os because we sequentially write the vertices of $\mathcal{G}$ and the link records in $\mathcal{L}$ to disk in the order they are created, which can be done in a blockwise fashion. To see that Phase 4 takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os, observe that the untangling of all subgraphs $\mathcal{G}(e)$ of $\mathcal{G}$ takes $\mathcal{O}(\mathrm{sort}(\sum |\mathcal{G}(e)|)) = \mathcal{O}(\mathrm{sort}(|\mathcal{G}|))$ I/Os, by Lemma 7. By Lemma 18, $|\mathcal{G}| = \mathcal{O}(N)$, so that all subgraphs of $\mathcal{G}$ can be untangled in $\mathcal{O}(\mathrm{sort}(N))$ I/Os. Also, observe that every DAG $\mathcal{G}'(e)$ is in fact a planar st-graph, because it is series-parallel; hence, it can be topologically sorted in $\mathcal{O}(\text{sort}(|\mathcal{G}'(e)|))$ I/Os [14], and topologically sorting $\mathcal{G}'$ takes $\mathcal{O}(\text{sort}(N))$ I/Os. The remainder of Phase 4 takes $\mathcal{O}(\text{sort}(N))$ I/Os, as it involves sorting and scanning linear size lists a constant number of times. Hence, the I/O-complexity of the algorithm is $\mathcal{O}(\text{sort}(N))$.

The correctness of the algorithm follows from the correctness of each of its phases: The correctness of Phase 1 is shown in [11]. Phase 2 is trivial. Lemma 17 establishes that the graphs $\mathcal{G}(y_h)$ constructed in Phase 3 have Properties G1–G5, which implies that Phase 4 constructs the path decompositions of the tree decomposition $\mathcal{E}$ correctly because Phase 4 precisely implements the rules for deriving these path decompositions from the corresponding graphs $\mathcal{G}(e)$. The correctness of the linking in Phase 4 follows from the observation that a correct tree decomposition is obtained by translating every link (α, β) in the link table into an edge (a, b) between any two nodes a and b that correspond to α and β, because all these nodes contain the relevant vertices of G. $\square$

Remark. We have not included an explicit description of the (simpler) algorithm for computing a path decomposition of minimal width for a graph $G = (V, E)$, but the algorithm is given implicitly in the description of the procedures for constructing the path decompositions corresponding to trunk-edges.

7 Constructing a Nice Tree Decomposition

In this section, we consider the following problem: Given a tree decomposition $\mathcal{D} = (\mathcal{X}, T)$ of width k and linear size for a graph $G = (V, E)$, construct a *nice* tree decomposition $\mathcal{E} = (\mathcal{Y}, \mathcal{U})$ of linear size and width at most k for G. We first sketch the internal-memory algorithm of [11] for this problem and then show how to make this algorithm I/O-efficient.

The algorithm of [11] first constructs a chordal graph $G' \supseteq G$ with tree decomposition $\mathcal{D}$ and then computes a perfect elimination ordering (PEO) of the vertices of G'. Then it processes the vertices of G' in reverse elimination order; initially tree decomposition $\mathcal{E}$ consists of a single node r with Y_{r} containing the last $k+1$ vertices in the PEO; the remaining vertices are processed one by one.

Let v be the next vertex to be inserted. By the chordality of G', vertex v is simplicial in the subgraph of G' induced by all vertices following and including v in the PEO. Hence, the neighbors of v form a clique C of size k in this graph and must be stored at some node in the part of tree decomposition $\mathcal{E}$ constructed so far. Moreover, there exists such a node i_v that is either a leaf or has only one child, because $\mathcal{E}$ is nice.

If i_v is a leaf, the algorithm adds one or two vertices below i_v: If $Y_{\mathsf{i}_v} = C$, only one child j_v of i_v with $Y_{\mathsf{j}_v} = C \cup \{v\}$ is added. Otherwise, two nodes j_v and k are added, with j_v being a child of k and k being a child of i_v. The sets associated with these two nodes are $Y_{\mathsf{k}} = Y_{\mathsf{i}_v} \setminus \{y\}$, for some vertex $y \in Y_{\mathsf{i}_v} \setminus C$, and $Y_{\mathsf{j}_v} = Y_{\mathsf{k}} \cup \{v\}$.

If i_v has one child j, a node k with $Y_{\mathsf{k}} = Y_{\mathsf{i}_v}$ is inserted between i and j and another child l of i_v with $Y_{\mathsf{l}} = Y_{\mathsf{i}}$ is added. Node l is a leaf, and the insertion procedure of the previous paragraph can be applied with l playing the role of i_v.

In order to make this algorithm I/O-efficient, we have to show how to compute G', a PEO of G', and tree decomposition $\mathcal{E}$ in $\mathcal{O}(\text{sort}(N))$ I/Os.

Given tree decomposition $\mathcal{D} = (\mathcal{X}, T)$ of G, graph G' can be computed as follows: For every node $\mathsf{i} \in T$ and every pair of vertices $\{v, w\} \subseteq X_{\mathsf{i}}$, add an edge $\{v, w\}$ to E'. In [22, Lemma 2.2.3], it is proved that G' is chordal. We show in Section 8 how to compute a PEO of G', using tree-decompositon $\mathcal{D}$. The rest of this section deals with the construction of the nice tree decomposition $\mathcal{E}$.

In order to construct $\mathcal{E}$, we have to solve two problems: First we have to show how to find the node i_v, for each vertex v to be inserted. Then we have to show how to update the structure of $U = (I, F)$ as we keep inserting the vertices of G. The following modification of the algorithm helps to solve both problems efficiently: Instead of computing $\mathcal{E}$ right away, we first construct the tree decomposition $\mathcal{E}' = (\mathcal{X}', U')$ that is obtained from $\mathcal{E}$ by contracting all edges $\{\mathsf{i}, \mathsf{j}\} \in U$ such that $X_\mathsf{i} = X_\mathsf{j}$. The nodes in U' may have many children. Given $\mathcal{E}'$, we expand every node $\mathsf{i} \in U'$ with more than one child into a binary tree U_i whose number of leaves equals the number of children of i in U' and then make every child of i in U' the child of a different leaf in U_i.

Now the procedure for inserting vertex v into the current tree decomposition simplifies to the following two steps: Let C be the set of neighbors of v that succeed v in the PEO. Then we have to find a node i_v such that $C \subseteq Y_{\mathsf{i}_v}$, and we have to add one or two descendants of i_v, depending on whether $C = Y_{\mathsf{i}_v}$ or $C \subset Y_{\mathsf{i}_v}$. The following observation tells us how to find such a node i_v, for every vertex v: Let w be the last vertex in v's neighborhood that has been inserted before v. Since the neighbors of v that succeed v in the PEO form a clique, they must all be neighbors of w. Hence, we can choose i_v to be the leaf created when inserting w into the tree decomposition. (Note that i_v may no longer be a leaf when v is inserted).

Assuming that every vertex $v \in G'$ is labeled with its number $\nu(v)$ in the PEO, vertex w is the neighbor of v with the smallest number $\nu(w) > \nu(v)$ in the PEO. A single scan of the adjacency lists of all vertices in G' is sufficient to determine this vertex, for all $v \in G'$, and to create a list L containing one pair (w, v), for every vertex $v \in G'$. We sort list L lexicographically, thereby storing all pairs $(w, v_1), \ldots, (w, v_k)$ consecutively.

Now we process the vertices of G' in reverse elimination order, first creating a node r in $\mathcal{E}$ that contains the last $k+1$ vertices of G' and then adding the vertices one by one. For every vertex $w \in Y_\mathsf{r}$, we process all its entries in L and insert a pair (v, r) into a max-priority queue[2] Q, for every processed entry $(w, v) \in L$. When inserting vertex $v \in G'$ into $\mathcal{E}$, we perform DeleteMax operations until we retrieve the unique entry (v, i_v) from Q. Then we perform the insertion procedure for vertex v, as described above, and insert an entry (u, j_v) into priority queue Q, for every pair $(v, u) \in L$.

Once we have processed all vertices of G' in this manner, we obtain tree decomposition $\mathcal{E}'$. The replacement of high-degree nodes in U' by binary trees, as described above, is straightforward. Hence, we obtain the following lemma.

Lemma 19 *Given a graph $G = (V, E)$ and a tree decomposition $\mathcal{D} = (\mathcal{X}, T)$ of width k and size $O(N)$ for G, a nice tree decomposition $\mathcal{E} = (\mathcal{Y}, U)$ of width at most k and size $\mathcal{O}(N)$ for G can be computed in $\mathcal{O}(\mathrm{sort}(N))$ I/Os.*

Proof. Given tree decomposition $\mathcal{D}$, the construction of G' takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os, as we only have to scan the nodes of the tree decomposition to create a multi-set containing the edges of G'; then we sort and scan the resulting multiset to remove duplicate edges. The construction of the PEO takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os, by Lemma 20. The computation of list L takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os, as it only requires scanning the adjacency lists of the vertices of G' and sorting list L. Given list L, the construction of $\mathcal{E}$ takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os: We scan list L as well as the vertex set of G', sorted

[2] A priority queue which supports DeleteMax instead of DeleteMin operations.

in reverse elimination order; and we perform $\mathcal{O}(N)$ priority queue operations, because every entry in L causes one INSERT and one DELETEMAX operation to be performed on Q. To see that the algorithm is correct, we have to show that every vertex v, except the last $k+1$ vertices of G', has a neighbor w with $\nu(w) > \nu(v)$. This, however, follows from the chordality of G' and the fact that G' is connected. □

8 Finding a Perfect Elimination Ordering of a k-Tree

In this section, we consider the following problem: Given an undirected graph $G = (V, E)$ and a tree decomposition $\mathcal{D} = (\mathcal{X}, T)$ of width k for G, assume that we have computed a chordal supergraph $G' = (V, E')$ of G, by making sure that every set X_i, $\mathsf{i} \in T$, is a clique in G'. We want to find a perfect elimination ordering (PEO) of the vertices of G'.

The algorithm is simple: We traverse the tree T in preorder. At the root r of T we "process" all vertices in X_r, where "processing" means that we assign to each of these vertices the highest possible number in the PEO that has not been used yet. At any other vertex j with parent i, we "process" all vertices in $X_\mathsf{j} \setminus X_\mathsf{i}$.

Lemma 20 *Given a graph $G = (V, E)$, a chordal supergraph $G' = (V, E')$ of G, and a tree decomposition $\mathcal{D} = (\mathcal{X}, T)$ of G and G', as defined above, it takes $\mathcal{O}(\text{sort}(N))$ I/Os to compute a PEO of G'.*

Proof. Clearly, the above method takes $\mathcal{O}(\text{sort}(N))$ I/Os and produces an ordering of the vertices of G. We have to prove that this ordering is a PEO.

Assume that there are three vertices $u < v < w$ such that there are edges $\{u, v\}$ and $\{u, w\}$ in G', where "$<$" is the computed PEO. We show that edge $\{v, w\}$ must also be in G', which proves that the computed ordering is indeed a PEO. Let i, j, and k be the first nodes in a preorder traversal of T that contain u, v, and w, respectively. As $u < v < w$, we have $\mathsf{i} \geq \mathsf{j} \geq \mathsf{k}$. First observe that k must be an ancestor of i. Indeed, if k is not an ancestor of i, let a be the lowest common ancestor of i and k; then any path from u to w has to contain a vertex from X_a, so that edge $\{u, w\}$ cannot exist. For the same reason, j must be an ancestor of i. However, no proper ancestor of i contains u. Hence, as edges $\{u, v\}$ and $\{u, w\}$ are in G', $\{v, w\} \subseteq X_\mathsf{i}$, and edge $\{v, w\}$ is in G'. □

9 Applications

In this section, we present three applications of our tree-decomposition algorithm. All the algorithms in this section assume that a nice tree decomposition of the graph is given. In Section 9.1, we show how to solve the single-source shortest path problem on directed graphs of bounded treewidth in $\mathcal{O}(\text{scan}(N))$ I/Os. The algorithm assumes that the given graph does not contain any negative cycles. In Section 9.2, we show how to compute optimal separators for graphs of bounded treewidth and how to exploit the information provided by these separators to compute DFS-trees for graphs of bounded treewidth. In Section 9.3, we argue that the linear-time solutions for many NP-hard problems on graphs of bounded treewidth of [5, 6, 7] can be combined with the linear-I/O time-forward processing procedure of [42] to solve these problems in $\mathcal{O}(\text{scan}(N))$ I/Os.

9.1 Single Source Shortest Paths

The input to our algorithm is a nice tree decomposition $\mathcal{D} = (\mathcal{X}, T)$ of the given graph $G = (V, E)$ represented as follows: The nodes of the tree $T = (I, F)$ are stored in preorder. Every node $\mathsf{i} \in I$ is represented by the set of at most $k+1$ vertices in X_i. Every edge $(v, w) \in E$ is stored at every node $\mathsf{i} \in I$ such that $\{v, w\} \subseteq X_\mathsf{i}$.

Our algorithm uses dynamic programming in T and is divided into three phases. The first phase processes the nodes of T bottom-up, computing for every pair of vertices $\{v, w\} \subseteq X_\mathsf{i}$ the distances $d_\mathsf{i}(v, w)$ and $d_\mathsf{i}(w, v)$ from v to w and from w to v in G_i. The second phase processes the nodes of T top-down and computes for all pairs of vertices $\{v, w\} \subseteq X_\mathsf{i}$ the distances $d(v, w)$ and $d(w, v)$ from v to w and from w to v in G. The third phase uses the distance information computed by the first two phases to build a shortest path tree for the desired source vertex s.

During the first phase of the algorithm $(k+1) \times (k+1)$-matrices M'_i are computed for all nodes $\mathsf{i} \in I$. Let the vertices in X_i be $v_1, \ldots, v_{k+1}$. Then position (j, k) of matrix M'_i stores $d_\mathsf{i}(v_j, v_k)$. The second phase uses matrices M'_i to compute a matrix M_i, for every node $\mathsf{i} \in T$, which stores distance $d(v_j, v_k)$ at position (j, k). The third phase uses the distances $d(v_j, v_k)$ computed in the second phase to compute distances $d(s, v)$, $v \in G$.

Phase 1. For a start node i, computing M'_i is straightforward as $G_\mathsf{i} = G[X_\mathsf{i}]$. Any other node has either one or two children. If i has one child, it is either an introduce or a forget node. For a forget node i with child j, $G_\mathsf{i} = G_\mathsf{j}$ and $X_\mathsf{j} = X_\mathsf{i} \cup \{x\}$. Hence, we only delete the row and column corresponding to x from M'_j to obtain M'_i. For an introduce node i with child j, $X_\mathsf{i} = X_\mathsf{j} \cup \{x\}$. Let v and w be two nodes in X_i with shortest path $P = \langle v = u_0, u_1, \ldots, u_s = w\rangle$ in G_i. If x is not contained in this path, then P is also a shortest path from v to w in G_j, and we just copy the corresponding entry from M'_j to M'_i. Otherwise, let $u_l = x$. The only edges in G_i that are not in G_j are edges with endpoint x. Hence, the paths $P_1 = \langle u_0, \ldots, u_{l-1}\rangle$ and $P_2 = \langle u_{l+1}, \ldots, u_s\rangle$ exist in G_j and must be shortest paths from u_0 to u_{l-1} and from u_{l+1} to u_s, respectively, in G_j. Thus, the distance between v and w in G_i is the same as the distance between v and w in the following graph $\hat{G}_\mathsf{i}$. The vertex set of $\hat{G}_\mathsf{i}$ is X_i. For every finite entry in M'_j, there is an edge of that weight between the corresponding vertices in X_j. Finally, we add all edges incident to x in G_i to $\hat{G}_\mathsf{i}$. Now we compute M'_i by running the Floyd-Warshall algorithm [19, 41] on $\hat{G}_\mathsf{i}$ in internal memory, as $|\hat{G}_\mathsf{i}| = \mathcal{O}(1)$.

If i is a join node, i has children j and k with $X_\mathsf{i} = X_\mathsf{j} = X_\mathsf{k}$. Also, if $G_\mathsf{j} = (V_\mathsf{j}, E_\mathsf{j})$ and $G_\mathsf{k} = (V_\mathsf{k}, E_\mathsf{k})$, then $G_\mathsf{i} = (V_\mathsf{j} \cup V_\mathsf{k}, E_\mathsf{j} \cup E_\mathsf{k})$. We construct a graph $\hat{G}_\mathsf{i}$ from M'_j and M'_k as follows: The vertex set of $\hat{G}_\mathsf{i}$ is again X_i; the edge set contains an edge (v_j, v_k) if position (j, k) is less than infinity in at least one of M'_j and M'_k. The weight of the edge is $\min\{M'_\mathsf{j}(j, k), M'_\mathsf{k}(j, k)\}$. Again, we compute M'_i by running the Floyd-Warshall algorithm on $\hat{G}_\mathsf{i}$. We have to prove that this gives the right result.

Consider a shortest path P in G_i from a vertex $v \in X_\mathsf{i}$ to another $w \in X_\mathsf{i}$. We cut P into maximal subpaths $P_1, \ldots, P_q$ such that none of these paths has an interior vertex in X_i. Such a path P_k stays completely inside one of the graphs G_j and G_k because the vertices in X_i form a separator of G_i cutting G_i into two pieces: one contains all vertices in $V_\mathsf{j} - X_\mathsf{i}$; the other contains all vertices in $V_\mathsf{k} - X_\mathsf{i}$. Hence, if u and z are the endpoints of P_k with $u, z \in X_\mathsf{i}$, then P_k must be the shortest path between u and z in either G_j or G_k, so that we have assigned the length of P_k as the weight of edge (u, z) in $\hat{G}_\mathsf{i}$. As we do this for all subpaths P_k, the length of the shortest path between any pair of vertices in $\hat{G}_{\mathsf{i}}$ is just the length of the shortest path between these two vertices in G_{i}.

Phase 2. Having computed matrices M'_{i}, for all nodes $\mathsf{i} \in T$, which store distances $d_{\mathsf{i}}(v,w)$, for all pairs of vertices $v, w \in X_{\mathsf{i}}$, we now use these matrices to compute matrices M_{i}, for all nodes $\mathsf{i} \in T$, which store distances $d(v,w)$, for all pairs of vertices $v, w \in X_{\mathsf{i}}$.

Consider a node j with parent i. Depending on the type of i, there are two cases to consider. If i is a join or introduce node, then $X_{\mathsf{j}} \subseteq X_{\mathsf{i}}$. Otherwise, $X_{\mathsf{j}} = X_{\mathsf{i}} \cup \{x\}$. The root r of the tree T does not have any parent, and $G_{\mathsf{r}} = G$. Hence, $M_{\mathsf{r}} = M'_{\mathsf{r}}$. That is, the distances between vertices in X_{r} stored in M'_{r} are the distances between these vertices in G.

Now consider the case of a node j with a join or introduce node i as parent. As already noted $X_{\mathsf{j}} \subseteq X_{\mathsf{i}}$. By induction, matrix M_{i} already stores all the distances in G between vertices in X_{i}. As $X_{\mathsf{j}} \subseteq X_{\mathsf{i}}$, we just copy the relevant entries from M_{i} to M_{j}.

If node j's parent i is a forget node, $X_{\mathsf{j}} = X_{\mathsf{i}} \cup \{x\}$. That is, matrix M_{i} already stores the distance in G between any pair of vertices $\{v, w\} \subseteq X_{\mathsf{j}} \setminus \{x\}$. We have to compute the distances between x and all other vertices $v \in X_{\mathsf{i}}$. Consider a shortest path P from x to v. Let w be the first vertex in X_{i} suceeding x on path P and consider the subpaths P_1 from x to w and P_2 from w to v. Both paths have to be shortest paths as well. Hence, the length of P_2 is stored in M_{i}. We claim that P_1 stays within G_{j}, which implies that the length of P_1 is just the distance $d_{\mathsf{i}}(x,w)$ from x to w stored in M'_{j}. Again, this claim is easy to prove, as the vertices in X_{i} form a separator of the vertices in G_{j} from the rest of G. That is, in order to reach a vertex not in G_{j}, a path starting at x must cross some vertex in X_{i}, but w is the first such vertex in P, so that P_1 cannot contain any vertex not in G_{j}. A similar argument shows that a shortest path from a vertex $v \in X_{\mathsf{i}}$ to vertex x can be divided into a shortest path from v to another vertex $w \in X_{\mathsf{i}}$ and a shortest path from w to x which stays completely inside G_{j}.

Hence, we build a graph $\tilde{G}_{\mathsf{j}}$ with vertex set X_{j}. An edge (v,w) in $\tilde{G}_{\mathsf{j}}$ has the weight given in M_{i} if $x \notin \{v,w\}$. For edges incident to x, we take the appropriate edge weight from M'_{j}. Matrix M_{j} will now be filled with the distances in $\tilde{G}_{\mathsf{j}}$ between the vertices in X_{j}. Again, we compute these distances in internal memory, using the Floyd-Warshall algorithm.

Phase 3. The third and final phase of the algorithm uses the distances $d(v,w)$, $v, w \in X_{\mathsf{i}}$, computed by the second phase to compute the distances $d(s,v)$ from the source s to all other vertices $v \in G$. In particular, we compute for every node i a vector Δ_{i} storing the distances from node s to the nodes in X_{i}. We do this as follows: First we extract all nodes i such that $s \in X_{\mathsf{i}}$. For all these nodes, the matrices M_{i} already give us the distances from s to the nodes in X_{i}. After removing the subtree T_s of T induced by these nodes, we obtain a set of subtrees $T_1, T_2, \ldots, T_q$ of T. We root these subtrees at the nodes that are adjacent to nodes in T_s. Now we process each of these trees top-down as follows.

Consider such a tree T_l. Let j be a node in T_l with parent i. Let $v \in X_{\mathsf{j}} \setminus X_{\mathsf{i}}$. Then any path from s to v must contain at least one vertex in $X_{\mathsf{i}} \cap X_{\mathsf{j}}$. Hence, $d(s,v) = \min\{d(s,w) + d(w,v) : w \in X_{\mathsf{i}} \cap X_{\mathsf{j}}\}$. Distances $d(s,w)$ are already provided by the vector Δ_{i}, as $X_{\mathsf{i}} \cap X_{\mathsf{j}} \subseteq X_{\mathsf{i}}$. Distances $d(w,v)$ are provided by the matrix M_{j}.

It is an exercise to augment the three phases of this algorithm to compute a shortest path-tree with root s instead of only the distances from s to all other vertices in G.

Theorem 7 *Given a nice tree decomposition $\mathcal{D} = (\mathcal{X}, T)$ of width at most k and size $\mathcal{O}(N)$ for a directed graph $G = (V, E)$, the single-source shortest path problem in G can be solved in $\mathcal{O}(\mathrm{scan}(N))$ I/Os and linear space, provided that the nodes of T are stored in preorder.*

Proof. Given that the nodes of T are stored in preorder, Phases 1 and 2 of the above algorithm take $\mathcal{O}(\mathrm{scan}(N))$ i/Os using the linear-I/O time-forward processing technique for trees of [42] to realize the bottom-up or to-down processing of T. This is true because we send only $\mathcal{O}(1)$ information along every edge of T, and the computation at every node is carried out in internal memory. In order to realize Phase 3 in $\mathcal{O}(\mathrm{scan}(N))$ I/Os, the nodes in each subtree T_l need to be rearranged in preorder w.r.t. its new root. For all of these subtrees, except the tree T_r containing the root r of T, this preorder numbering is consistent with the preorder numbering of T, so that no computation is required. In order to change the preorder numbers of the nodes of T_r and rearrange the nodes according to the new preorder numbering, it is sufficient to compute an Euler tour of T_r and then use this Euler tour to derive the desired preorder numbering. This can be done in $\mathcal{O}(\mathrm{scan}(N))$ I/Os using ideas similar to those used in the time-forward processing technique of [42]. The details are straightforward. The correctness of the above algorithm follows from the discussion included with the description of the algorithm. □

By Theorem 4 and Lemma 19, a nice tree decomposition $\mathcal{D} = (\mathcal{X}, T)$ of G can be computed in $\mathcal{O}(\mathrm{sort}(N))$ I/Os. Computing a preorder numbering of the nodes of T takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os using the Euler tour technique and list-ranking [14]. Hence, we obtain the following corollary.

Corollary 2 *Given a directed graph $G = (V, E)$ of treewidth at most k, the single-source shortest path problem in G can be solved in $O(\mathrm{sort}(N))$ I/Os and linear space.*

9.2 Depth-First Search

In this section, we address an important fundamental problem on graphs of bounded treewidth that remains entirely elusive on general graphs: computing a DFS-tree of a directed graph. Our algorithm takes $\mathcal{O}(N/B)$ I/Os, once a tree decomposition of the graph is given.

Let $\mathcal{D} = (\mathcal{X}, T)$ be a nice tree decomposition of G with root r. Our DFS-algorithm is the standard internal-memory DFS-procedure [15]; but it uses the tree decomposition to guide the order in which the out-edges of a given vertex v are explored. As we show, combined with an appropriate blocking of tree T, this ensures that the total number of I/Os performed by the algorithm is $\mathcal{O}(N/B)$.

Blocking. We partition T into $\mathcal{O}(N/B)$ subtrees of size B. Since T is binary, we can use the partition procedure described in [**?**, 42] to achieve this. We store every such subtree in a separate block. Every (copy of) a vertex $v \in G$ is augmented with a pointer to the root of the subtree T_v of T that corresponds to node v. Every vertex in every set X_i has room to store whether it has been explored and to store a mark to be used in the depth-first search.

Choosing out-neighbours. When the DFS visits a node v for the first time, it initiates the following procedure: It locates the root r_v of T_v and defines i_v to be the root of T_v. Every time the DFS visits node v, we repeat the following procedure: If there is an unexplored out-neighbour w of v in X_{i_v}, we choose this out-neighbour to be the next vertex to be visited and push the pair

(v, i_v) onto the DFS-stack, to be recovered when the DFS backtracks to v from w. If there is no unexplored out-neighbour of v in X_{i_v}, we check whether $v \in X_{\mathsf{j}}$, where j is the left child of i_v, and whether v is unmarked in X_{j}. If so, we set $\mathsf{i}_v = \mathsf{j}$ and repeat the procedure. Otherwise, we determine whether X_{k}, where k is the right child of i_v, contains an unmarked copy of v. If so, we set $\mathsf{i}_v = \mathsf{k}$ and repeat the procedure. If neither X_{j} nor X_{k} contain an unmarked copy of v, we mark v in $\mathcal{X}_{\mathsf{i}_v}$; if v is contained in the set of i_v's parent p, we set $\mathsf{i}_v = \mathsf{p}$. Otherwise, the exploration of v is finished, and we backtrack to v's parent in the DFS-tree.

In essence, our algorithm performs a depth-first traversal of T_v to locate the out-neighbours of v. We argue next that this costs only $\mathcal{O}(N/B)$ I/Os for constructing the DFS-tree of G.

First observe that, except for the traversal of the tree decomposition, the DFS-procedure takes $\mathcal{O}(N/B)$ I/Os. Indeed, we perform $\mathcal{O}(N)$ operations on the stack of vertices representing the path from the root of the DFS-tree to the current vertex. These operations take $\mathcal{O}(N/B)$ I/Os. No random accesses are required to determine whether a given out-neighbour w of a node v is explored. This is true because, at the time when we explore edge (v, w), we are at a node i_v such that $w \in X_{\mathsf{i}_v}$. Thus, we only have to query the local copy of w to determine w's status. This, of course, requires the updating of this status when w is explored. The easy way to do this is to traverse all of T_w and update the status of all copies of w when w is explored. A more clever argument, which we omit here, shows that it is sufficient to copy the status between copies of w residing in adjacent tree nodes as we move between these nodes.

It remains to show that loading tree nodes as we move in the tree decomposition incurs no more than $\mathcal{O}(N/B)$ I/Os. We split the I/Os into two types. A *jump* is an I/O that loads the root of a subtree T_v into memory when v is visited for the first time and an I/O that loads the node i_w of the parent w of the current node into memory when we backtrack from v to w. A *step* is an I/O that loads a node of T into memory as a result of moving between adjacent nodes in T_v.

Lemma 21 *The total number of I/Os incurred by steps is $\mathcal{O}(N/B)$.*

Proof. We partition steps further into downward steps and upward steps, depending on which direction we are moving in as we take the step. Since we perform a DFS-traversal of T_v, the total number of upward steps equals the number of downward steps. Hence, it suffices to bound the number of downward steps. Assume that a step is taken from node i to node j during a traversal of T_v. Then $v \in X_{\mathsf{j}}$, and j is the root of one of the $\mathcal{O}(N/B)$ subtrees into which T has been partitioned by our blocking. Hence, the total number of downward steps is bounded by the number of nodes in the roots of these subtrees. There are $\mathcal{O}(N/B)$ subtrees, each containing k nodes. Hence, the number of downward steps is $\mathcal{O}(kN/B) = \mathcal{O}(N/B)$. □

Note that Lemma 21 implies that the marking of all copies of an explored node w mentioned above incurs only $\mathcal{O}(N/B)$ extra I/Os because this marking process can be implemented by traversing T_w in the same order as the subsequent steps we take.

Lemma 22 *The total number of I/Os incurred by jumps is $\mathcal{O}(N/B)$.*

Proof. Similar to the pairing of downward and upward steps, we can divide jumps into explore jumps, which are the result of exploring a new vertex, and backtrack jumps, which are the result of backtracking to a vertex's parent. Every backtrack jump must have been preceded by a corresponding explore jump, so that it suffices to count explore jumps.

To do so, we make the simple observation that an explore jump always jumps up tree T. Indeed, since w is contained in the set X_{i_v} of the current node i_v when w is explored, the root of T_w must be an ancestor of i_v. Hence, w must be contained in the set X_{j} of the root j of the tree in the blocking that contains i_v. This limits the number of jumps out of any block to the number of vertices in the root of this subtree, which is constant. Since there are $\mathcal{O}(N/B)$ subtrees, the total number of jumps is also $\mathcal{O}(N/B)$. □

Since a nice tree decomposition and the required blocking of the decomposition can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os, we obtain the following result.

Theorem 8 *Depth-first search can be solved in $\mathcal{O}(\text{sort}(N))$ I/OS on a directed or undirected graph of constant treewidth.*

9.3 Solving NP-Hard Problems on Graphs of Bounded Treewidth

Arnborg, Lagergren, and Seese [5] present polynomial-time solutions for NP-hard problems on bounded treewidth graphs which are describable in monadic second order logic. Arnborg and Proskurowski [6] present linear time algorithms for graphs of bounded treewidth by processing a k-tree embedding of the graph. It is straightforward to rewrite their algorithms to use a rooted tree decomposition instead of a k-tree, so that their algorithms compute a solution by traversing the tree decomposition bottom-up. Bodlaender [7] defines two classes of graph problems that are decidable in polynomial time on graphs of bounded treewidth and NP-hard in general. As the algorithms in [6], his algorithms compute an answer to the problem by traversing the tree decomposition bottom-up. He proves that several of these problems are decidable in linear time on graphs of bounded treewidth or graphs of bounded treewidth and bounded degree. The linear time tree-traversal algorithms of [6, 7] together with the linear-I/O time-forward processing technique of [42] give the following results.

Theorem 9 *Given a graph $G = (V, E)$ with treewidth bounded by some constant k, the following optimization problems can be solved in $\mathcal{O}(\text{sort}(N))$ or $\mathcal{O}(\text{scan}(N))$ I/Os and linear space, depending on whether a nice tree decomposition $\mathcal{D} = (\mathcal{X}, T)$ for G of width at most k and size $\mathcal{O}(N)$ is given and the nodes of T are stored in preorder: vertex cover, chromatic number, independent set, dominating set, and Hamiltonian cycle.*

Proof. Simulate the algorithms from [6] using the time-forward processing technique for trees proposed in [42]. If the tree decomposition is not part of the input, a nice tree decomposition can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os by Theorem 4 and Lemma 19. Using results from [14], it takes $\mathcal{O}(\text{sort}(N))$ I/Os to arrange the nice tree decomposition in preorder. □

Theorem 10 *Given a graph $G = (V, E)$ with treewidth bounded by some constant k, the following decision problems can be solved in $\mathcal{O}(\text{sort}(N))$ I/Os or $\mathcal{O}(\text{scan}(N))$ I/Os and linear space, depending on whether a tree decomposition $\mathcal{D} = (\mathcal{X}, T)$ for G of width at most k and size $O(N)$ is given and the nodes of T are stored in preorder: vertex cover, chromatic number, independent set, bipartite subgraph, k-closure, and max cut.*

Proof. Analogous to the previous result, using the algorithms from [7] for graph problems in 1-ECC (see [7] for a definition of this complexity class). □

For completeness, we illustrate the approach for Theorem 9 using the vertex cover problem as an example. The other solutions are similar. Details can be found in [6]. Also, the solutions in [6] are easily extended to find minimum or maximum weight solutions instead of minimum or maximum cardinality solutions. The weighted vertex cover problem is defined as follows:

> Given a graph $G = (V, E)$ and a weight function $\omega : V \to \mathbb{R}$, find a vertex set $C \subseteq V$ of minimum weight such that every edge $e \in E$ has at least one endpoint in C.

The algorithm proceeds in two phases. The first phase processes the tree T bottom-up, i.e., starting at the leaves and computing compact representations of partial solutions for internal nodes from representations computed for their children. The second phase processes the tree top-down to compute the final solution.

In the bottom-up phase a *candidate set* $\mathcal{C}_\mathsf{i}$ is computed, for every node i. This set $\mathcal{C}_\mathsf{i}$ contains all pairs (S, ω), where $S \subseteq X_\mathsf{i}$ and there is a vertex cover C of G_i containing all vertices in S, but no vertex in $X_\mathsf{i} \setminus S$. Let $Cov(S)$ be the set of all such vertex covers of G_i. Then $\omega = \min\{\omega(C) : C \in Cov(S)\}$. After the bottom-up phase, we choose the pair $(S, \omega) \in \mathcal{C}_\mathsf{r}$, where r is the root of T, such that ω is minimized. The top-down phase constructs a vertex cover $C \in Cov(S)$ with $\omega(C) = \omega$.

Bottom-up phase. The set $\mathcal{C}_\mathsf{i}$ is easy to compute for a start node because $Cov(S) = \emptyset$ if S is not a vertex cover of $G[X_\mathsf{i}]$, and $Cov(S) = \{S\}$ if S is a vertex cover of $G[X_\mathsf{i}]$. In the latter case, we put $(S, \omega(S))$ into $\mathcal{C}_\mathsf{i}$.

At a forget node i with child j, observe that any vertex cover of G_i is also a vertex cover of G_j and vice versa. Given a vertex cover C for G_j, then $C \cap X_\mathsf{i} = S$ if and only if $C \cap X_\mathsf{j} = S$ or $C \cap X_\mathsf{j} = S \cup \{x\}$. Hence, we put a pair (S, ω) into $\mathcal{C}_\mathsf{i}$ if at least one of (S, ω_1) and $(S \cup \{x\}, \omega_2)$ is in $\mathcal{C}_\mathsf{j}$. We define $\omega = \min\{\omega_1, \omega_2\}$.

At an introduce node i with child j, observe that any vertex cover C of G_i must cover all edges in G_j and all edges incident to x. Moreover, x cannot cover any of the edges in G_j. Hence, either $x \in C$ and $C \setminus \{x\}$ is a vertex cover for G_j, or $x \notin C$, C is a vertex cover for G_j, and $\Gamma_{G_\mathsf{i}}(x) \subseteq C$. As $\Gamma_{G_\mathsf{i}}(x) \subseteq X_\mathsf{i}$, $\Gamma_{G_\mathsf{i}}(x) \subseteq S$ in the latter case. Hence, we construct $\mathcal{C}_\mathsf{i} = \mathcal{C}' \cup \mathcal{C}''$ from two sets $\mathcal{C}'$ and $\mathcal{C}''$: $\mathcal{C}' = \{(S \cup \{x\}, \omega + \omega(x)) : (S, \omega) \in \mathcal{C}_\mathsf{j}\}$ and $\mathcal{C}'' = \{(S, \omega) : (S, \omega) \in \mathcal{C}_\mathsf{j} \wedge \Gamma_{G_\mathsf{i}}(x) \subseteq S\}$.

Finally, at a join node i with children j and k, observe that any vertex cover C of G_i must be a vertex cover for $G_\mathsf{j} = (V_\mathsf{j}, E_\mathsf{j})$ and $G_\mathsf{k} = (V_\mathsf{k}, E_\mathsf{k})$. Let $C_1 = C \cap V_\mathsf{j}$, $C_2 = C \cap V_\mathsf{k}$, and $C \cap X_\mathsf{i} = S$. Then $C_1 \cap X_\mathsf{j} = S$ and $C_2 \cap X_\mathsf{k} = S$. Hence, there is a vertex cover C with $C \cap X_\mathsf{i} = S$ if and only if $(S, \omega_1) \in \mathcal{C}_\mathsf{j}$ and $(S, \omega_2) \in \mathcal{C}_\mathsf{k}$, where $\omega_1 = \omega(C_1)$ and $\omega_2 = \omega(C_2)$. As the vertices in S are counted in ω_1 and ω_2, we have to compute $\omega = \omega_1 + \omega_2 - \omega(S)$ and add (S, ω) to $\mathcal{C}_\mathsf{i}$.

Top-down phase. Once we have reached the root r of T, we can immediately report the weight of the minimum weight vertex cover by examining pairs $(S, \omega) \in \mathcal{C}_\mathsf{r}$ and reporting the weight $\omega_{\min} = \min\{\omega : (S, \omega) \in \mathcal{C}_\mathsf{r}\}$. We construct a vertex cover C with weight $\omega(C) = \omega_{\min}$ as follows:

At the root r of T, we mark the element $(S_{\min}, \omega_{\min}) \in \mathcal{C}_\mathsf{r}$ as selected and add the vertices in $S_{\min}$ to C. At any other node j with parent i, the computation depends on the type of its parent.

If i is a join node with selected pair (S, ω), we mark the pair $(S, \omega') \in \mathcal{C}_\mathsf{j}$ as selected, but add no vertices to C.

If i is an introduce node with selected pair (S, ω), then either $x \in S$ or $x \notin S$. If $x \in S$, we mark the pair $(S \setminus \{x\}, \omega - \omega(x)) \in \mathcal{C}_\mathsf{j}$ as selected. Otherwise, we mark the pair $(S, \omega) \in \mathcal{C}_\mathsf{j}$ as selected. Again, we do not add any vertices to C.

If i is a forget node with selected pair (S, ω), we mark one of the pairs (S, ω_1) or $(S \cup \{x\}, \omega_2)$ as selected, depending on which one has less weight. It may also be that $(S, \omega_1) \notin \mathcal{C}_\mathsf{j}$, in which case the only possible choice is $(S \cup \{x\}, \omega_2)$. If we mark pair $(S \cup \{x\}, \omega_2)$ as selected, we add vertex x to C.

The correctness of the bottom-up phase follows from the discussion given in the description of that phase. The correctness of the top-down phase follows from the observation that we mark one pair (S, ω) as selected in every candidate set $\mathcal{C}_\mathsf{i}$ and we guarantee that $C \cap X_\mathsf{i} = S$. Hence, C is a vertex cover for all graphs $G[X_\mathsf{i}]$, where i is a leaf of T, and an inductive argument shows that C is a vertex cover for $G_\mathsf{i} = (V_\mathsf{i}, E_\mathsf{i})$. A similar inductive argument shows that if (S, ω) is the selected pair in $\mathcal{C}_\mathsf{i}$, then $\omega(C \cap V_\mathsf{i}) = \omega$, so that $\omega(C) = \omega_{\min}$.

10 Conclusions and Open Problems

Even though our algorithms exploit the constant treewidth of the input graphs to obtain asymptotically more efficient algorithms than those for general graphs, they suffer from the same drawbacks as existing internal memory algorithms for graphs of bounded treewidth, namely large constants hidden in the big-Oh notation which are super-exponential in the treewidth of the graph. Thus, the contribution of our paper is of rather theoretical interest, and the results in this paper should be seen as a step towards understanding the I/O-complexity of fundamental graph problems, while practitioners cannot benefit from the results presented here.

A fact that is worth mentioning is that all classes of sparse graphs with I/O-efficient solutions for the single-source shortest path problem have small balanced separators: Outerplanar graphs have $\frac{2}{3}$-separators of size 2, planar graphs have $\frac{2}{3}$-separators of size $\mathcal{O}(\sqrt{N})$, and graphs of bounded treewidth have $\frac{2}{3}$-separators of size k. Thus, the results presented in this paper seem to suggest that there exist I/O-efficient algorithms for the SSSP problem on outerplanar and planar graphs not so much because they are planar, but rather because they have small separators. Still, the planarity of outerplanar and planar graphs can be exploited to obtain much more efficient and possibly practical solutions for the single-source shortest path problem in these graphs than the shortest path algorithm for graphs of bounded treewidth presented here.

References

1. J. Abello, A. L. Buchsbaum, and J. Westbrook. A functional approach to external graph algorithms. *Algorithmica*, 32(3):437–458, 2002.

2. L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proceedings of the 4th Workshop on Algorithms and Data Structures*, volume 955 of *Lecture Notes in Computer Science*, pages 334–345. Springer-Verlag, 1995.

3. L. Arge, G. S. Brodal, and L. Toma. On external memory MST, SSSP, and multi-way planar separators. In *Proceedings of the 7th Scandinavian Workshop on Algorithm Theory*, volume 1851 of *Lecture Notes in Computer Science*, pages 433–447. Springer-Verlag, 2000.

4. S. Arnborg, B. Courcelle, A. Proskurowski, and D. Seese. An algebraic theory of graph reduction. *Journal of the ACM*, 40(5):1134–1164, November 1993.

5. S. Arnborg, J. Lagergren, and D. Seese. Problems easy for tree-decomposable graphs. In *Proceedings of the 15th International Colloquium on Automata, Languages and Programming*, volume 317 of *Lecture Notes in Computer Science*, pages 38–51. Springer-Verlag, 1988.

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

7. H. L. Bodlaender. Dynamic programming on graphs with bounded treewidth. In *Proceedings of the 15th International Colloquium on Automata, Languages and Programming*, volume 317 of *Lecture Notes in Computer Science*, pages 105–118. Springer-Verlag, 1988.

8. H. L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, 25:1305–1317, 1996.

9. H. L. Bodlaender and B. de Fluiter. Parallel algorithms for series parallel graphs. Technical Report UU-CS-1997-21, Utrecht University, Utrecht, the Netherlands, 1997.

10. H. L. Bodlaender and T. Hagerup. Parallel algorithms with optimal speedup for bounded treewidth. *SIAM Journal on Computing*, 27:1725–1746, 1998.

11. H. L. Bodlaender and T. Kloks. Efficient and constructive algorithms for the pathwidth and treewidth of graphs. *Journal of Algorithms*, 21:358–402, 1996.

12. A. Broder, R. Kumar, F. Manghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. In *Proceedings of the 9th International World-Wide Web Conference*, 2000. `http://www9.org`.

13. A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. On external memory graph traversal. In *Proceedings of the 11th ACM-SIAM Symposium on Discrete Algorithms*, pages 859–860, 2000.

14. Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149, January 1995.

15. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, second edition, 2001.

16. B. de Fluiter and H. L. Bodlaender. Parallel algorithms for treewidth two. In *Proceedings of the 23rd International Workshop on Graph-Theoretic Concepts in Computer Science*, volume 1335 of *Lecture Notes in Computer Science*, pages 157–170. Springer-Verlag, 1997.

17. F. Dehne, W. Dittrich, and D. Hutchinson. Efficient external memory algorithms by simulating coarse-grained parallel algorithms. In *Proceedings of the 9th ACM Symposium on Parallel Algorithms and Architectures*, pages 106–115, 1997.

18. G. A. Dirac. On rigid circuit graphs. *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*, 25:71–76, 1961.

19. R. W. Floyd. Algorithm 97 (SHORTEST PATHS). *Communications of the ACM*, 5(6):345, 1962.

20. F. Harary. *Graph Theory*. Addison-Wesley, 1969.

21. P. N. Klein. Efficient parallel algorithms for chordal graphs. In *Proceedings of the 29th Symposium on foundations of Computer Science*, pages 150–161, 1989.

22. T. Kloks. *Treewidth: Computations and Approximations*, volume 842 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1994.

23. V. Kumar and E. J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proceedings of the 8th IEEE Sumposium on Parallel and Distributed Processing*, pages 169–176, October 1996.

24. J. Lagergren. Efficient parallel algorithms for tree-decomposition and related problems. In *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science*, pages 173–182, 1990.

25. A. Maheshwari and N. Zeh. External memory algorithms for outerplanar graphs. In *Proceedings of the 10th International Symposium on Algorithms and Computation*, volume 1741 of *Lecture Notes in Computer Science*, pages 307–316. Springer-Verlag, December 1999.

26. A. Maheshwari and N. Zeh. I/O-efficient algorithms for graphs of bounded treewidth. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 89–90, 2001.

27. A. Maheshwari and N. Zeh. I/O-optimal algorithms for planar graphs using separators. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 372–381, 2002.

28. J. Matoušek and R. Thomas. Algorithms finding tree-decompositions of graphs. *Journal of Algorithms*, 12:1–22, 1991.

29. K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In *Proceedings of the 10th Annual European Symposium on Algorithms*, volume 2461 of *Lecture Notes in Computer Science*, pages 723–735. Springer-Verlag, 2002.

30. U. Meyer. External memory BFS on undirected graphs with bounded degree. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 87–88, 2001.

31. U. Meyer and N. Zeh. I/O-efficient undirected shortest paths. In *Proceedings of the 11th European Symposium on Algorithms*, 2003.

32. R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, December 1987.

33. B. Reed. Finding approximate separators and computing treewidth quickly. In *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing*, pages 221–228, 1992.

34. D. J. Rose, R. E. Tarjan, and G. S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on Computing*, 5:266–283, 1976.

35. C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, 1994.

36. R. E. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing*, 13(3):566–579, August 1984.

37. W. T. Tutte. *Graph Theory.* Cambridge University Press, 2001. First published by Addison-Wesley, 1984.

38. J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.

39. J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2–3):110–147, 1994.

40. J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory II: Hierarchical multilevel memories. *Algorithmica*, 12(2–3):148–169, 1994.

41. S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

42. N. Zeh. *I/O-Efficient Algorithms for Shortest Path Related Problems.* PhD thesis, School of Computer Science, Carleton University, 2002.