I/O-Efficient Algorithms for Bounded Treewidth Graphs

Anil Maheshwari Norbert Zeh

August 6, 2000

Abstract

We present an I/O-efficient algorithm for the single source shortest path (SSSP) problem for graphs of bounded treewidth. For sparse graphs in general SSSP seems to be extremely hard to solve. We show how to solve SSSP in $O(sort(N))$ I/Os for the above class of graphs. An important ingredient to our solution is an $O(sort(N))$ algorithm to compute a tree-decomposition of width k of the given graph. Given this decomposition, the algorithm for solving SSSP is rather simple. In order to construct the tree-decomposition, we present an I/O-efficient algorithm for finding a maximal matching of a graph, and introduce flippable DAGs as an interesting new concept that may be useful for other applications as well. We also show how to realize the well-known time-forward processing technique in $O(scan(N))$ I/Os if the processed graph is a tree whose vertices are sorted in preorder. This does not give any asymptotic improvements, but is interesting from a practical point of view, as the used techniques are extremely simple.

1 Introduction

1.1 Background and Motivation

In [32], finding I/O-optimal algorithms for the single source shortest path problem has been identified as an important open problem. This paper can be seen as part of our endeavour to identify classes of graphs that exhibit sufficient structure, in order to beat the $\Omega(|V|)$ bound exhibited by all known single source shortest path algorithms [26, 22].

Also, although many problems have been shown to be NP-hard, some of them still have to be solved (at least approximately), in order to answer certain real life problems. A long list of publications on graphs of bounded treewidth [25, 23, 28, 5, 8, 11, 21, 4, 9, 15, 10] shows how to deal with NP-hard graph problems, provided that the graph has bounded treewidth,[1] which is true for many well-studied classes of graphs including trees, partial k-trees, series parallel graphs, k-outerplanar graphs, Halin graphs, control flow graphs of goto-free programs, chordal, interval, and circular arc graphs with maximum clique size k. For graphs having this property, these papers provide polynomial or even linear time solutions to problems that would be intractable on general graphs.

An issue that these algorithms do not address is the question of I/O-efficiency. If the problem instances become too large to fit into the internal memory of current state-of-the-art computers, the internal memory algorithms, which often exhibit fairly random memory access patterns, may be forced to perform one I/O-operation per data access. The result is an immense slow-down of the algorithm, and from a practical point of view, the problem is intractable again, using these algorithms for huge data sets.

Our paper tries to bridge this gap by providing an $O(sort(N))$ algorithm to construct a tree-decomposition of a given input graph of bounded treewidth. Most internal memory algorithms for these graphs proceed

[1] See Bodlaender's web page (`http://www.cs.ruu.nl/~hansb`) for excellent surveys on this topic.

using a traversal of the tree-decomposition and almost trivially translate into efficient external memory algorithms, once the tree-decomposition is given.

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. Unfortunately, this I/O-bottleneck is becoming more significant as parallel computing gains popularity and CPU speeds increase, since disk speeds are not keeping pace [30]. 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) [33], as well as a number of other external memory models [13, 34]. 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. The complexity of an algorithm is the number of I/O operations it performs.

In [33] it has been shown that sorting an data set of size N takes $sort(N) = \Theta((N/DB)\log_{M/B}(N/B))$; permuting the data set takes $perm(N) = \Theta(\min\{N, sort(N)\})$ I/Os; scanning the data set takes $scan(N) = \Theta(N/DB)$ I/Os.

1.3 Previous Work

Internal memory algorithms for bounded treewidth graphs have been well-studied [25, 23, 28, 5, 8, 4, 9, 15, 10]. Bodlaender [8] gives a linear-time algorithm for constructing a tree-decomposition of width at most k for a graph of treewidth at most k, using the algorithm of [11] to compute a decomposition of width at most k from a decomposition of width at most $2k$. Several authors give improved algorithms for graphs of treewidth at most two [9], and treewidth at most four [5]. In [6, 4, 7], linear time algorithms for problems on bounded treewidth graphs are given, which are NP-hard for general graphs. These algorithms assume that the tree-decomposition is given as part of the input. Several authors deal with the problem of computing tree-decompositions of bounded treewidth graphs in parallel using the PRAM model of computation [9, 15, 10, 23]. Bodlaender and Hagerup [10] give PRAM algorithms for general k, while Bodlaender and de Fluiter [9] give more practical solutions for graphs of treewidth at most two. We are not aware of any external memory results on computing tree-decompositions of graphs or NP-hard problems. However, the embedding algorithm in [24] can be used to obtain a tree-decomposition of width 2 for outerplanar graphs in $O(sort(N))$ I/Os.

Shortest path problems are well-studied in internal memory. Dijkstra's algorithm [14] when implemented using Fibonacci heaps [18] is the best known algorithm for the single source shortest path problem in general graphs and runs in $O(|V|\log|V| + |E|)$ time. In [16], an $O(n\sqrt{\log n})$ algorithm for single source shortest paths and an $O(n^2)$ algorithm for all pairs shortest paths in planar graphs is given. In [17], the all pairs shortest path algorithm was improved to $O(pn)$, where p is the size of the minimum face-on-vertex cover for the given graph. Of course, in order to beat the $\Omega(n^2)$ time bound for all pairs shortest paths, Frederickson needs to use an implicit representation of the all pairs shortest path information. In [19], a linear time single source shortest path algorithm for planar graphs is given.

In external memory, the best known single source shortest path algorithm is due to Kumar and Schwabe [22] and takes $O(V + (E/B)\log_{M/B}(E/B))$ I/Os. 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 general graphs is due to Munagala and Ranade [26] and takes $O(V + sort(E))$ I/Os. In [24], Maheshwari and Zeh give

$O(sort(N))$ algorithms for BFS and DFS in outerplanar graphs. They also prove $\Omega(perm(N))$ lower bounds for embedding, BFS, and DFS. Recently, Arge et al. [3] provided an $O(sort(N))$ single source shortest path algorithm for planar graphs, provided that a BFS tree of the graph is given.

In internal memory, a simple greedy algorithm computes a maximal matching for any graph in $O(|V| + |E|)$ time. The best known external memory results are $O(|E|/|V| sort(|V|) \log_2(V/M))$ I/Os deterministically and $O(sort(|E|))$ I/Os [1].

A perfect elimination ordering of a chordal graph can be found in linear time using algorithms by [27, 29, 31]. In the PRAM model, Klein [20] shows how to compute a perfect elimination ordering in $O(\log^2 n)$ time using $O((n+m)/\log n)$ 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

Our results are the following:

1. An $O(sort(N))$ algorithm to test whether a given graph has treewidth at most k. This algorithm constructs a tree-decomposition of width at most k if the testing algorithm confirmed that the treewidth of the given graph is at most k. (Section 5)

2. An $O(scan(N+I))$ algorithm for time-forward processing in rooted trees, provided that the input tree is sorted in preorder. Here, I is the total amount of data sent along the edges of T. As preorder numbering takes $O(sort(N))$ I/Os, this result gives an asymptotic improvement over the previous $O(sort(N+I))$ result [12] only if $I = \omega(N)$. However, the techniques used in our solution are extremely simple, and the constants hidden in the big-Oh are extremely small, which makes this technique very attractive from a practical point of view, especially if several passes over the tree are required. (Section 3)

3. An $O(sort(|V|+|E|))$ algorithm to compute a maximal matching of a given graph G. Our algorithm is deterministic and matches the I/O-complexity of the best previously known randomized algorithm for this problem. (Section 5.2)

4. An $O(sort(N))$ algorithm to compute a perfect elimination ordering for the chordal extension of a partial k-tree. (Section 5.3.4)

5. An $O(scan(N))$ single source shortest path algorithm for directed graphs whose underlying undirected graphs have treewidth bounded by some constant k, provided that the tree-decomposition is given as part of the input, and the tree nodes are sorted in preorder. (Section 6)

6. $O(scan(N))$ algorithms for the following problems on graphs of bounded treewidth: maximum clique, minimum vertex colver, maximum independent set, Hamiltonian cycle, and chromatic number. Again the algorithm assumes that the tree-decomposition is given as part of the input, and the tree nodes are sorted in preorder. (Section 6)

In Section 4, we develop *flippable DAGs* as a concept to solve the follwing problem: Assume we are given two *st*-graphs G_1 and G_2 that we want to join into a new DAG G either by series or parallel composition of graphs G_1 and G_2. When performing a series composition, we may choose to join the sink of G_1 to the sink of G_2 and turn all edges in G_2 around. During a parallel composition, we may join the sink of one graph to the source of the other, which again requires flipping all edges in one of the graphs. Graph

G may itself be involved in further series or parallel compositions. Now it is easy to construct an example where $\Omega(N^2)$ edge flips are required, if we always perform the required edge flips explicitly. Using flippable DAGs, we perform only $O(1)$ updates per composition; once the final DAG has been constructed, every edge is examined exactly once, in order to determine whether or not it has to be flipped to give it the proper direction.

2 Preliminaries

Basic concepts An *(undirected) graph* is a pair $G = (V, E)$ of sets V and E; V is the *vertex set* of G; E is the *edge set* of G and contains unordered pairs $\{v, w\} \subseteq V$. A graph G is *simple* if for all $v \in V$, $\{v, v\} \notin E$. In this paper we consider only simple graphs. A *directed graph* is a pair $G = (V, E)$ of sets V and $E \subseteq V \times V$. That is, the edges in G are *ordered* pairs (v, w). We call v the *source* and w the *target* of (v, w) and say that edge (v, w) is directed from v to w. For an undirected graph $G = (V, E)$, let the *directed graph corresponding to* G be the graph $D(G) = (V, \{(v, w), (w, v) : \{v, w\} \in E\})$. For a directed graph $G = (V, E)$, let the *underlying undirected graph* be the graph $U(G) = (V, \{\{v, w\} : (v, w) \in E\})$. Note that $U(D(G)) = G$, for every undirected graph G, but not necessarily $D(U(G)) = G$ for a directed graph G. For every vertex v in an undirected graph G, the *neighborhood* of v is the set $N_G(v) = \{w \in V : \{v, w\} \in E\}$. The *degree* of v is defined as $d_G(v) = |N_G(v)|$. Sometimes we just write $d(v)$ if the graph G is understood. For every vertex v in a directed graph G, the *in-neighborhood* and *out-neighborhood* of v are the sets $N_G^-(v) = \{w \in V : (w, v) \in E\}$, and $N_G^+(v) = \{w \in V : (v, w) \in E\}$, respectively. The neighborhood of v is the set $N_G(v) = N_G^-(v) \cup N_G^+(v)$. We define the *in-degree*, *out-degree*, and *degree* of v as $d_G^-(v) = |N_G^-(v)|$, $d_G^+(v) = |N_G^+(v)|$, and $d_G(v) = d_G^-(v) + d_G^+(v)$, respectively.

A graph $H = (W, F)$ is called a *subgraph* of $G = (V, E)$ if $W \subseteq V$ and $F \subseteq E$. Given a vertex set $X \subseteq V$, let $G[X]$ denote the subgraph of G *induced by* X defined as $G[X] = (X, \{\{v, w\} \in E : \{v, w\} \subseteq X\})$. Let $G - X$ denote the graph $G[V \setminus X]$.

A *walk* in a directed graph $G = (V, E)$ is a sequence $p = (v_1, \ldots, v_n)$ such that $(v_i, v_{i+1}) \in E$, for $1 \leq i < n$; p is a *path* if all vertices in p are distinct; p is a *cycle* if $v_1 = v_n$ and a *simple cycle* if $v_1 = v_n$ and all vertices in p except v_1 and v_n are distinct. We call v_1 and v_n the *source* and *target* of p. A *directed acyclic graph* (DAG) is a directed graph $G = (V, E)$ without cycles. Denote all vertices v with $d^-(v) = 0$ as *sources* in G and all vertices with $d^+(v) = 0$ as *sinks* in G.

Given a weight function $\omega : E \to \mathbb{R}$, the *weight* of a walk $p = (v_1, \ldots, v_n)$ is as $\omega(p) = \sum_{i=1}^{n-1} \omega((v_i, v_{i+1}))$. We call a walk negative or positive if its weight is negative or positive. If we are given a weight function ω for a directed graph G, we call G *weighted*. Given a weighted directed graph G without negative cycles, a shortest path from a vertex v to a vertex w is a path $\pi(v, w)$ with minimal weight among all paths with source v and target w. Walks, paths, cycles, simple cycles, and shortest paths in an undirected graph G are the same as walks, paths, cycles, simple cycles, and shortest paths in $D(G)$.

An undirected graph $G = (V, E)$ is *connected* if there is a path from $v \in V$ to any other vertex $w \in V$. A directed graph $G = (V, E)$ is connected if $U(G)$ is connected. The *connected components* of a graph G are its maximal connected subgraphs. A tree is a connected graph T that does not contain cycles. For a connected graph $G = (V, E)$, a *spanning tree* is a tree $T = (V, F)$. For a graph $G = (V, E)$, a *spanning forest* contains a spanning tree for every connected component of G. A *rooted tree* is a tree $T = (V, F)$ with a distinguished root vertex $r \in V$. We call a vertex $v \in V$ and *ancestor* of another vertex $w \in V$ and w a *descendant* of v if the unique path from r to w in T contains v; v is the *parent* of w and w is a *child* fo v if v is an ancestor of w and $\{v, w\} \in F$. For rooted trees, we define the *degree* of a vertex v as the number of its children. (Note that this definition differs from the degree definition for undirected graphs.)

Given an undirected graph $G = (V, E)$, a *matching* $\mathcal{M}$ is a subset of edges in E such that no two edges in $\mathcal{M}$ share an endpoint. A matching is *maximal* if every edge in $E \setminus \mathcal{M}$ shares at least one endpoint with an

edge in $\mathcal{M}$. A *clique* is a subgraph H of G such that for all vertices $v, w \in H$, $\{v,w\} \in E$. A vertex $v \in V$ is *simplicial* if the neighborhood $N_G(v)$ of v in G forms a clique. In particular, $N_G(v) \cup \{v\}$ is a clique.

Given a cycle $c = (v_1, \ldots, v_n)$ in a graph G, a *chord* of c is an edge $\{v_i, v_j\}$ such that $j - i \geq 2$ and either $i > 1$ or $j < n-1$. An undirected graph $G = (V,E)$ is *triangulated* 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 for every vertex v, the set $\{w \in N_G(v) : v \prec w\}$ is a clique.

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 can be obtained from two st-graphs G_1 and G_2 with sources s_1 and s_2 and sinks t_1 and t_2 by identifying t_1 and s_2 (series composition) or by identifying s_1 with s_2 and t_1 with t_2. In this paper, we also consider the graph consisting of a single vertex as series parallel. Series parallel graphs do not contain the complete graph with 4 vertices as a minor. Hence, they are planar.

Tree-decompositions Given an undirected graph $G = (V,E)$, a *tree-decomposition* $\mathcal{D} = (\mathcal{X}, T)$ consists of a tree $T = (I,F)$ and a collection $\mathcal{X}$ of sets $\{X_\mathsf{i} : \mathsf{i} \in I\}$ 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, $X_\mathsf{i} \cap X_\mathsf{k} \subseteq X_\mathsf{j}$.

To avoid confusion we will refer to the vertices of graph G as *vertices* and use small intalic letters to denote these vertices, while we use small sans serif letters to denote the nodes of the tree T, which we denote as *nodes*. 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 over all possible tree-decompositions of G. In particular, the tree-width of G is 1 if and only if G is a forest of trees. We define the tree-width of a directed graph G to be the same as the tree-width of $U(G)$. A *rooted tree-decomposition* is a tree-decomposition with a distinguished root vertex. Given a rooted tree-decomposition $\mathcal{D} = (\mathcal{X}, T)$, let $desc(\mathsf{i})$ be the set of descendants of node i in T, including i. Let $T_\mathsf{i} = T[desc(\mathsf{i})]$. Let $G_\mathsf{i} = G[\bigcup_{\mathsf{j} \in desc(\mathsf{i})} X_\mathsf{j}]$, and $\mathcal{D}_\mathsf{i}$ be the tree-decomposition of G_i obtained from $\mathcal{D}$ by pruning T to T_i and removing all sets X_j, $\mathsf{j} \notin desc(\mathsf{i})$, from $\mathcal{X}$.

A rooted tree-decomposition is *nice* if all nodes of T have at most two children and the following conditions hold: (1) If node i has degree 2 and j and k are the children of i, then $X_\mathsf{i} = X_\mathsf{j} = X_\mathsf{k}$; (2) if node i has degree 1 and j is the child of i, then either $X_\mathsf{j} \subseteq X_\mathsf{i}$ and $|X_\mathsf{i}| = |X_\mathsf{j}| + 1$ or $X_\mathsf{i} \subseteq X_\mathsf{j}$ and $|X_\mathsf{j}| = |X_\mathsf{i}| + 1$. We call a leaf of T a *start node*, a degree-2 node a *join node*, and a degree-1 node a *forget node* if $X_\mathsf{i} = X_\mathsf{j} \setminus \{x\}$ and an *introduce node* if $X_\mathsf{i} = X_\mathsf{j} \cup \{x\}$.

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 can root a path-decomposition by distinguishing one of its two leaves as the root of the path. The *pathwidth* of a graph G is the minimum width over all possible path-decompositions of G. Given two path-decompositions Y_1 and Y_2, denote by $Y_1 \circ Y_2$ the concatenation of Y_1 and Y_2. This operation is allowed only if $Y_1 \circ Y_2$ satisfies Properties T1–T3.

Typical sequences and typical lists Given an integer sequence $a = (a_1, \ldots, a_n)$, let the *length* of a be $|a| = n$ and $\max(a) = \max\{a_i : 1 \leq i \leq n\}$. 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 $c_i = a_i + b_i$, $1 \leq i \leq n$. 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 $a_i = a_{i+1}$ in a, then a_{i+1} is removed 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 $a_i \geq a_j \geq a_k$, then elements $a_{i+1}, \ldots, a_{k-1}$ are removed from a.

Lemma 1 [11] *The typical sequence $\tau(a)$ of a sequence a is well-defined.*

Lemma 2 [11] *If the elements in 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 is a sequence $a^* = (a_1^*, \ldots, a_m^*)$ such that there are indices $t_1 < t_2 < \cdots < t_{n+1} = m+1$ such that for all $1 \leq i \leq n$ and $t_i \leq j < t_{i+1}$, $a_{t_i} = a_j^*$. Let $E(a)$ be the set of all extensions of a.

Lemma 3 [11] *If $a^* \in E(a)$, then $\tau(a^*) = \tau(a)$.*

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

Lemma 4 [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 $b = (b_1, \ldots, b_m)$, let the *concatenation* of a and b be the sequence $a \circ b = (a_1, \ldots, a_m, b_1, \ldots, b_n)$.

Lemma 5 [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 (b, c) of sequences 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 first case we say that the split is of *type one*; otherwise, it is of *type two*.

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

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

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

```
 1: Let δ(r) be some dummy input for the root r of T.
 2: PUSH(S, δ(r))
 3: for every node v of T in preorder do
 4:    δ(v) ← POP(S)
 5:    Evaluate v using δ(v) and φ(v)
 6:    Let w_1, ..., w_k be the children of v sorted by increasing preorder numbers
 7:    for i = k, k−1, ..., 1 do
 8:       PUSH(S, δ(w_i))
 9:    end for
10: end for
```

Algorithm 1: Top-down time-forward processing in a rooted tree.

3 Time-Forward Processing for Rooted Trees

Time-forward processing was introduced in [12] as a technique to evaluate directed acyclic graphs. That is, every vertex is initially labeled with a predicate $\phi(v)$. In the end, we want to compute a predicate $\psi(v)$ for every vertex, which may only depend on the predicate $\phi(v)$ and some input received from the in-neighbors of v. In the general setting, this technique requires $O(sort(N+I))$ I/Os, where N is the number of vertices in G, and I is the total amount of information sent along all edges of G. In this section, we show the following result.

Theorem 1 *Given a rooted tree $T = (V,E)$ whose edges are directed and whose vertices are sorted in preorder, T can be evaluated in $O(scan(N+I))$ I/Os using $O((N+I)/B)$ blocks of external memory, where I is the total amount of information sent along all edges of T.*

In order to prove this theorem, we will need the following two lemmas.

Lemma 6 *Given a rooted tree $T = (V,E)$ whose edges are directed from parents to children and whose vertices are sorted in preorder, T can be evaluated in $O(scan(N+I))$ I/Os using $O((N+I)/B)$ blocks of external memory, where I is the total amount of information sent along all edges of T.*

Proof. Denote the data sent from the parent of a node v to node v by $\delta(v)$. We use a stack S to implement the time-forward processing. Algorithm 1 shows the code. In order to prove this algorithm correct, we have to show that $\delta(v)$ is indeed the top element of S in Line 4 when vertex v is being evaluated. We will need the following claim.

Claim 1 *Let m be the number of elements on stack S before evaluating a vertex v of T, and let T_v be the subtree of T rooted at v. Then the stack level never falls below $m-1$ during the evaluation of T_v. After the evaluation of T_v, the stack level is $m-1$.*

Proof. We prove the claim by induction on the size $|T_v|$ of T_v. If $|T_v| = 1$ (i.e., v is a leaf), v removes the top entry from the stack in Line 4 and does not put any new entries onto the stack, as the loop in Lines 7–9 is never executed. Hence, the claim is true.

So assume that $|T_v| > 1$. Then v has at least one child. Let $w_1, \ldots, w_k$ be the children of v. The evaluation of the T_v is done by evaluating v followed by the evaluation of subtrees $T_{w_1}, \ldots, T_{w_k}$. Note that each subtree T_{w_i} has size $|T_{w_i}| < |T_v|$; so the claim holds for T_{w_i} by induction hypothesis.

Let $S = (s_1, \ldots, s_m)$ before the evaluation of v. After the evaluation of v and before the evaluation of w_1, $S = (\delta(w_1), \ldots, \delta(w_k), s_2, \ldots, s_m)$. Inductively, we claim that $S = (\delta(w_i), \ldots, \delta(w_k), s_2, \ldots, s_m)$ before the evaluation of node w_i. As Claim 1 holds for T_{w_i}, the evaluation of T_{w_i} never touches the elements $\delta(w_{i+1}), \ldots, \delta(w_k), s_2, \ldots, s_m$ and removes the top element $\delta(w_i)$ from the stack. Hence, after the evaluation of T_{w_i}, $S = (\delta(w_{i+1}), \ldots, \delta(w_k), s_2, \ldots, s_m)$. In particular, after the evaluation of T_{w_k}, $S = (s_2, \ldots, s_m)$. But the evaluation of T_{w_k} finishes the evaluation of T_v. Hence, after the evaluation of T_v, $S = (s_2, \ldots, s_m)$, as desired. Also, before the evaluation of every T_{w_i}, $|S| \geq m$. By induction hypothesis, the stack level never falls below $m-1$ during the evaluation of T_{w_i}. Hence, the same is true for the evaluation of T_v. □

Claim 1 immediately implies the correctness of the lemma: If node v is the root of T, then $S = (\delta(v))$. Otherwise, v has a parent u with children $w_1, \ldots, w_k$ such that $v = w_i$. Immediately, after the evaluation of u, $S = (\delta(w_1), \ldots, \delta(w_k), \ldots)$. In the proof of Claim 1 it is shown that before the evaluation of $v = w_i$, $S = (\delta(w_i), \ldots, \delta(w_k), \ldots)$. Hence, in any case, $\delta(v)$ is the top of the stack immediately before the evaluation of node v.

In order to see the I/O-bound, observe that we scan the vertex list of T once, which takes $O(scan(N))$ I/Os. Data is sent along the tree edges using the stack S. Every data item is pushed once and popped once, so that we perform $O(I)$ stack operations, which cost $O(scan(I))$ I/Os. In total, we spend $O(scan(N+I))$ I/Os. The space requirements are clearly bounded by $O((N+I)/B)$ blocks of external memory, as this is the maximal number of blocks accessible in the given number of I/Os. □

Lemma 7 *Given a rooted tree $T = (V, E)$ whose edges are directed from children to parents and whose vertices are sorted in preorder, T can be evaluated in $O(scan(N+I))$ I/Os using $O((N+I)/B)$ blocks of external memory, where I is the total amount of information sent along all edges of T.*

Proof. The proof is similar to that of Lemma 6. We simply reverse the order of processing the nodes of T. □

Proof (Theorem 1). The crucial observation is the following: A vertex v with an edge directed from v to v's parent in T can only receive input from its children. More formally, consider the subgraph T' of T induced by all edges of T that are directed from children to parents. T' is a forest of rooted trees. Then the following claim holds.

Claim 2 *For every non-root vertex v in T', $N^-_{T'}(v) = N^-_T(v)$.*

Proof. Assume that $N^-_{T'}(v) \neq N^-_T(v)$. All edges in $N^-_T(v)$ are directed from children of v towards v except the edge from v's parent to v (if it is in $N^-_T(v)$). Thus, this is the only edge that may not be in $N^-_{T'}(v)$. However, if the edge from v's parent to v is directed from v's parent to v, then v is a root in T'. □

Claim 2 implies that we can use the algorithm of Lemma 7 to evaluate all non-root vertices of T'. Moreover, for every root in T', all children have been fully evaluated, so that they can provide their input to v. Thus, every node that has not been evaluated yet is waiting only for the input from its parent. Thus, we consider the subgraph T'' of T induced by all edges directed from parents to children. Again T'' is a forest of rooted trees. We apply the algorithm of Lemma 6 to T'' to evaluate the remaining vertices.

By Lemmas 6 and 7 both phases of the algorithm take $O(scan(N+I))$ I/Os using $O((N+I)/B)$ blocks of external memory. □

Given the fact that a preorder numbering of T can be computed in $O(sort(N))$ I/Os using $O(N/B)$ blocks of external memory (e.g., using the Euler tour technique and list ranking [12]), we obtain the following corollary.

Corollary 1 *Given a rooted tree $T = (V,E)$ whose edges are directed, T can be evaluated in $O(sort(N) + scan(I))$ I/Os using $O((N+I)/B)$ blocks of external memory, where I is the total amount of information sent along all edges of T.*

Note that, if $I = O(N)$, Corollary 1 does not give an asymptotic improvement over the standard time-forward processing technique for general graphs [2, 12]. However, even then it is worthwhile to use our algorithm, in particular if an algorithm requires several passes over the tree, because the tree needs to be sorted only once; once the tree has been sorted, our algorithm is very efficient with extremely small constants hidden in the big-Oh.

4 Flippable DAGs

In this section we introduce the concept of flippable DAGs. The motivation is that we will later use series parallel *st*-graphs representing a path Y, which we want to be able to flip. That is, we want to decide whether we process Y from head to tail or from tail to head. This corresponds to processing G from s to t or from t to s. Thus, we will need an efficient way to turn the edges in G around. If all edges of G were given directed from s to t, this would require a single scan. However, G may be composed of two smaller DAGs G_1 and G_2, and we may have joined G_1 and G_2 at their sink vertices instead of joining the sink of G_1 with the source of G_2. Hence, the edges of G_2 need to be flipped. G_1 and G_2 are recursively composed of smaller DAGs. So we are facing the problem of deciding for each edge whether or not to flip it. This section gives a solution for this problem.

A *flippable DAG* is a directed graph $G = (V, E_1 \cup E_2)$ such that (V,E_1) is a DAG and $E_2 = \{(v,w) : (w,v) \in E_1\}$. Every vertex $v \in V$ will later be assigned a color $c(v) \in \{\mathsf{red}, \mathsf{blue}\}$. This defines a *coloring* of v, which is a function $c : V \to \{\mathsf{red}, \mathsf{blue}\}$. For a color $c \in \{\mathsf{red}, \mathsf{blue}\}$, let $\bar{c}$ denote the opposite color. That is, if $c = \mathsf{red}$, then $\bar{c} = \mathsf{blue}$ and vice versa. Given a coloring $c : V \to \{\mathsf{red}, \mathsf{blue}\}$, the coloring $\bar{c} : V \to \{\mathsf{red}, \mathsf{blue}\}$ is defined as $\bar{c}(v) = \overline{c(v)}$, for all $v \in V$. Every edge $e = (v,w)$ in $E = E_1 \cup E_2$ is assigned a pair of colors (c_1, c_2), where $c_1 \in \{\mathsf{red}, \mathsf{blue}\}$ and $c_2 \in \{\mathsf{red}, \mathsf{blue}\}$; for an edge $(v,w) \in E_1$ with color $c((v,w)) = (c_1, c_2)$, the color of edge $(w,v) \in E_2$ is $c((w,v)) = (\bar{c}_2, \bar{c}_1)$.

We represent G as an undirected graph $G' = (V, E')$, $E' = \{\{v,w\} : (v,w) \in E_1\}$, storing with every edge $e \in E'$ the color information about its two corresponding edges $e_1 \in E_1$ and $e_2 \in E_2$. Given an edge $e \in G'$ with corresponding edge $e_1 \in E_1$, edge e is *color-preserving* if $c(e_1) = (c_1, c_2)$ and $c_1 = c_2$. Otherwise, edge e is *color-changing*. Given a vertex $v \in V$, a color c, and a spanning tree T', we define a coloring $c = c(v, c, T')$ of G with respect to v, c, and T' as follows: Root T' at v and assign color $c(v) = c$ to v. For every other vertex $w \in V$ with parent $p(w)$ in T', let $c(w) = c(p(w))$ if edge $\{w, p(w)\}$ is color-preserving and $c(w) = \overline{c(p(w))}$ if edge $\{w, p(w)\}$ is color-changing. It is easy to see that $c(v, \bar{c}, T') = \overline{c(v, c, T')}$. G is a *valid* flippable DAG if the following two conditions hold:

(F1) Given any two spanning trees T' and T'' and any two vertices v and w such that coloring $c(v, c, T')$ assigns color c' to w, the colorings $c(v, c, T')$ and $c(w, c', T'')$ are the same. As all these colorings are the same, we write $c(v, c)$ for the coloring defined by coloring v with color c.[2]

(F2) Consider the edge set $E_{v,c}$ defined as $\{(v,w) \in E_1 \cup E_2 : c((v,w)) = (c(v), c(w))\}$. Then either $E_{v,c} = E_1$ or $E_{v,c} = E_2$.

As for a valid flippable DAG $E_{v,c} = E_1$ and $E_{v,\bar{c}} = E_2$ or vice versa, we uniquely choose a DAG $G_{v,c} = (V, E_{v,c})$ such that either $E_{v,c} = E_1$ or $E_{v,c} = E_2$, only by choosing the color of a vertex v. We call the construction of $G_{v,c}$ from G the *untangling* of G.

[2] This implies in particular that for every edge $\{v,w\} \in E'$, either $c((v,w)) = (c(v), c(w))$ or $c((w,v)) = (c(w), c(v))$, but not both.

Property F1 immediately gives us an algorithm to untangle G: (Remember that we represent G as the undirected graph G' defined above.) We compute a spanning tree T' of G', choose a root v of T' and a color c for the root vertex v. Then apply time-forward processing in T' to color all vertices of G. Now use sorting and scanning to extract all edges $(v,w) \in E_1 \cup E_2$ such that $c((v,w)) = (c(v), c(w))$.

If G has more than one connected component, we have to choose one vertex v_i for each connected component of G_i along with a color c_i for v_i. Then we compute a spanning tree for each of the components, topologically sort the spanning forest, and apply time-forward processing to the spanning forest as for the connected case. This gives us the following result.

Lemma 8 *A valid flippable DAG G of size N can be untangled in $O(sort(N))$ I/Os using $O(N/B)$ blocks of external memory, provided that G belongs to a class of graphs that is sparse under edge-contraction.*

5 Computing a Tree-Decomposition of Width k

Our algorithm for computing a tree-decomposition of width k for a graph $G = (V,E)$ of treewidth k follows the framework of [8]. We first describe this algorithm and then show how to perform the important subprocedures in $O(sort(N))$ I/Os, which will result in an $O(sort(N))$ algorithm to find the tree-decomposition. We want to emphasize that the result is of purely theoretical interest and does not give practical algorithms, as already for small treewidths the constants involved go beyond the memory size of current state-of-the-art computers. However, we believe that the result is still interesting as we will use it in Section 6.1 to extend the class of graphs for which $o(|V|)$ I/Os suffice to solve the single source shortest path problem.

5.1 The Main Algorithm

In order to describe the main algorithm we have to introduce some terminology. We choose two constants $c_1, c_2 \in \mathbb{R}^+$ with $0 < c_1, c_2 < 1$ such that

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

Constant c_1 is an upper bound on the fraction of vertices that is of "high degree" as defined below; c_2 is a lower bound on the fraction of vertices that is removed from G before calling the algorithm recursively. Let $d = \max(k^2 + 4k + 4, \lceil 2k/c_1 \rceil)$. A vertex of degree at most d is said to be of *low degree*; otherwise, the vertex is of *high degree*. A vertex is *friendly* if it is of low degree and adjacent to at least one other low-degree vertex. Given a tree-decomposition $\mathcal{D} = (\mathcal{X}, T)$ of G, a vertex v is said to be *pretty* w.r.t. $\mathcal{D}$ if it is of low degree, not friendly, and there exists a node $i \in I$ such that all neighbors of v are in X_i.

For a graph $G = (V,E)$, let the *improved graph* $G' = (V,E')$ of G be the graph obtained by adding an edge $\{v,w\}$ to E for all pairs of vertices $v,w \in V$ such that v and w have at least $k+1$ common neighbors of low degree in G. A vertex $v \in G$ is I-simplicial if it is simplicial in G', of low degree in G, and not friendly in G.

Algorithm 2 gives the main algorithm as described in [8], where its correctness is shown. The lemmas cited in parentheses after each step in the algorithm show how to realize this particular step in $O(sort(N))$ I/Os. All other steps take $O(sort(N))$ 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 αN for some $\alpha < 1$. Hence, the I/O-complexity of our algorithm is $I(N) = I(\alpha N) + O(sort(N)) = O(sort(N))$, and we obtain the following result.

Theorem 2 *Given a graph $G = (V,E)$ of treewidth at most k it takes $O(sort(N))$ I/Os and $O(N/B)$ blocks of external memory to compute a tree-decomposition $\mathcal{D} = (\mathcal{X}, T)$ of width at most k and size $O(N)$ for G.*

if $|G| \leq M$ **then**
  Compute a tree-decomposition of G using the algorithm in [8].
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 $M \subseteq E$ of G. (Lemma 11)
      Contract the edges in 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 $O(N)$ 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 M. The width of $\mathcal{D}''$ is at most $2k+1$.
        Compute a tree-decomposition $\mathcal{D} = (\mathcal{X}, T)$ of width at most k and size $O(N)$ for G from $\mathcal{D}''$ using Algorithm 3. (Theorem 4)
      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 9)
      if there is an I-simplicial vertex of degree at least $k+1$ (Lemma 9) **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 $O(N)$ for G^*.
          if the tree-width 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 $\mathrm{i}_v \in I^*$ such that $N_G(v) \subseteq X^*_{\mathrm{i}_v}$. Add a node j_v to I^*, make it adjacent to i_v and let $X^*_{\mathrm{j}_v} = N_G(v) \cup \{v\}$. Let $\mathcal{D} = (\mathcal{X}, T)$ be the resulting tree-decomposition of width at most k. (Lemma 10)
          end if
        end if
      end if
    end if
  end if
end if

Algorithm 2: Computing a tree-decomposition.

The next lemmas show how to realize Steps 18, 19, and 29 of Algorithm 2.

Lemma 9 *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 $O(sort(N))$ I/Os using $O(N/B)$ blocks of external memory.*

Proof. The proof is a straightforward adaptation of the internal memory algorithm for this problem given in [8]. We include it for completeness. We use sorting and scanning to compute the list of low-degree vertices and their adjacencies. Note that a low-degree vertex has constant degree for fixed k. Assume that there exists an ordering $<$ defined on the vertices of G (e.g., a numbering of the vertices). For each low-degree vertex u with neighborhood $N_G(u)$, we create a list $L(u) = \{(v,w,u) : \{v,w\} \subseteq N_G(u) \wedge v < w\}$. From the edge set of G we create a list $L' = \{(v,w,—) : \{v,w\} \in E \wedge v < w\}$. We create a list $L = L' \cup \bigcup L(u)$, where u iterates over all low-degree vertices of G. Note that $|L| = O(N)$ for fixed k. We sort L lexicographically, where the symbol "—" is assumed to be less than any vertex of G. In order to obtain E' we have to add an edge $\{v,w\}$ to E for all pairs of vertices $v < w$ such that there is no triple $(v,w,—) \in L$ and there are at least $k+1$ triples $(v,w,u_1),\ldots,(v,w,u_{k'}) \in L$. Also, for every triple $(v,w,u) \in L$, add an entry (u,v,w) to a new list S if the entry $(v,w,—) \in L$ or there are at least $k+1$ triples $(v,w,u_1),\ldots,(v,w,u_{k'}) \in L$. This can easily be tested in a single scan of L. List S now contains for every low-degree vertex u the edges connecting neighbors of u. We sort S lexicographically. As every such vertex u is of low-degree, the neighborhood of u is of constant size. Hence, it now takes a single scan of S to decide whether the neighborhood of u in G' forms a clique. We add each vertex u such that $N_{G'}(u)$ is a clique to the list SL of I-simplicial vertices. □

Given the lists S and SL in the proof of Lemma 9, we can decide whether there exists an I-simplicial vertex of degree at least $k+1$ only by scanning these two lists (Step 19).

Lemma 10 *Given a tree-decomposition $\mathcal{D}^* = (\mathcal{X}^*, T^*)$ of size $O(N)$ for the graph $G^* = G' - SL$, it takes $O(sort(N))$ I/Os and $O(N/B)$ blocks of external memory to compute a tree-decomposition $\mathcal{D} = (\mathcal{X}, T)$ of G' and thus G.*

Proof. Again the solution given in [8] can easily be adapted in order to obtain an I/O-efficient solution for this problem. For each I-simplicial vertex u with $N_G(u) = \{v_1,\ldots,v_l\}$, $v_1 < v_2 < \cdots < v_l$, create the tuple $(v_1,\ldots,v_l,u)$ and add it to a list L. For every node $\mathsf{i} \in I^*$ and every subset $\{x_1,\ldots,x_l\} \subseteq X_\mathsf{i}$ create a tuple $(x_1,\ldots,x_l,\mathsf{i})$, $x_1 < x_2 < \cdots < x_l$ and add it to L. As there are at most N I-simplicial vertices, the tree T^* has $O(N)$ nodes, and $|X_\mathsf{i}| = O(1)$ for all $\mathsf{i} \in I^*$, the list L has size $O(N)$. We sort this list lexicographically, where we assume that $\mathsf{i} < v$ for every $\mathsf{i} \in I^*$ and $v \in V$. As a result, every set of I-simplicial vertices with the same neighborhood is preceded by a tuple corresponding to a tree node containing this neighborhood. In a single scan of list L we can 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 $N_G(v) \subseteq X_{\mathsf{i}_v}$. Now we add a new node j_v to I^*, a set $X^*_{\mathsf{j}_v} = N_G(v) \cup \{v\}$ to $\mathcal{X}^*$, and the edge $\{\mathsf{i}_v, \mathsf{j}_v\}$ to T^*, for every I-simplicial vertex v. The result is the desired tree-decomposition $\mathcal{D} = (\mathcal{X}, T)$. □

5.2 Finding a Maximal Matching Greedily

Given an undirected graph $G = (V,E)$, we want to compute a maximal matching $\mathcal{M} \subseteq E$ of G. We start by numbering the vertices in V. We represent every edge $\{v,w\} \in E$ by the pair (v,w), $v < w$. We define an ordering on the edges as $(v,w) < (v',w')$ if $v < v'$ or $v = v'$ and $w < w'$. For every vertex, we sort the incident edges by this ordering and link them to form a list. This defines a DAG $G' = (E,E')$, where the edges in E' are the links defined by these sorted incidence lists.

Now we use time-forward processing in G' to find the maximal matching $\mathcal{M}$. We process the edges in E according to the above ordering. This guarantees that all edges in G' are directed from left to right. Note

that every node e of G' has at most two in and two out-edges, one per endpoint of e in G. Let $e = \{v, w\}$. If e does not receive any information along either of its in-edges, this means that no edge incident to v or w has been chosen for the matching yet. Thus, edge e can be chosen for the matching, and e sends this information to its two successors around v and w. If e receives some input along one of its in-edges, e cannot be chosen for the matching, as at least one of its endpoints is already matched. If e received this information from its predecessor around v, e passes the same information on to its successor around v. The same rule applies to w. This way we guarantee that we choose at most one edge incident to every vertex of G. On the other hand we choose an edge if we have not already added an edge incident to at least one of its endpoints. Hence, the matching we compute is indeed maximal. Constructing G' from G takes $O(sort(|V| + |E|))$ I/Os. The size of G' is $O(|E|)$. Hence, the time-forward processing phase takes $O(sort(|E|))$ I/Os, and we obtain the following result.

Lemma 11 *Given an undirected graph $G = (V, E)$, a maximal matching $M \subseteq E$ of G can be computed in $O(sort(|V| + |E|))$ I/Os.*

5.3 Improving the Tree-decomposition

In this section, we present our algorithm to compute a tree-decomposition $\mathcal{E} = (\mathcal{Y}, U)$ of width at most ℓ for a graph of treewidth at most ℓ, given a tree-decomposition of width k, where k and ℓ are constants. First we show how to test a given graph $G = (V, E)$ with tree-decomposition $\mathcal{D} = (\mathcal{X}, T)$ whether it has tree-width at most ℓ. Then we show how to augment this algorithm in order to obtain an actual tree-decomposition. The testing part is a straightforward simulation of the algorithm of [11] using the techniques described in Section 3. The construction of the corresponding tree-decomposition is non-trivial, as the algorithm described in [11] makes heavy use of the random access capability of the RAM model and cannot easily be adapted to access data in a block by block fashion.

5.3.1 Bounded Treewidth Testing

The algorithm we describe assumes that the given tree-decomposition $\mathcal{D} = (\mathcal{X}, T)$ is nice. In Section 5.3.3 we show how to transform a tree-decomposition into an equivalent nice tree-decomposition in $O(sort(N))$ I/Os. So this assumption is justified. Recall that the nodes in a nice tree-decomposition can be of four different types: *start*, *join*, *forget*, and *introduce* nodes. Given a node i of T, we will show how to compute a *full set of characteristics* $FS(\mathsf{i})$ from the full sets of characteristics of its children. This set has the property that for every characteristic in $FS(\mathsf{i})$, there is a tree-decomposition of G_i with this characteristic, and for every tree-decomposition $\mathcal{E}$ of G_i, there is another tree-decomposition $\mathcal{E}'$ of G_i whose characteristic is in $FS(\mathsf{i})$ and which is better than $\mathcal{E}$ in a sense not to be formalized here (see [11] for details). From this it follows immediately that the graph G has treewidth at most ℓ if and only if the set $FS(\mathsf{r})$ for the root r of T is not empty.

Let us define these concepts formally. We first have to define some concepts related to path-decompositions of subgraphs of G. Such path-decompositions will be part of the tree-decompositions we are constructing. 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 $Y = (Y_1, \ldots, Y_r)$ rooted at node i, the *restriction* of Y is the path-decomposition $Z = (Z_1, \ldots, Z_r)$ 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 $Z' = (Z_{q_1}, \ldots, Z_{q_t})$ 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 $q_{t+1} = |Y| + 1$.)

Given a partial path-decomposition $Y = (Y_1, \ldots, Y_r)$ with interval model $Z = (Z_{q_1}, \ldots, Z_{q_t})$, the *list representation* of Y is the pair $(Z, [Y])$, where $[Y] = (Y^{(1)}, Y^{(2)}, \ldots, Y^{(t)})$ and $Y^{(s)} = (Y_{q_s}, \ldots, Y_{q_{s+1}-1})$. Given the

list representation $(Z,[Y])$ of Y, we define the *list* of Y as $[y]=(y^{(1)},y^{(2)},\ldots,y^{(t)})$, $y^{(s)}=(|Y_{q_s}|,\ldots,|Y_{q_{s+1}-1}|)$. The *characteristic* of Y is the pair $C(Y)=(Z,\tau[y])$.

A tree-decomposition is *non-trivial* if for two adjacent nodes i and j in the tree, $X_\mathsf{i}\neq X_\mathsf{j}$. Obviously, every tree-decomposition can be transformed into a non-trivial tree-decomposition, so that it is sufficient if our algorithm considers non-trivial partial tree-decompositions as candidates to be represented by the full set of characteristics.

A leaf i of the tree-decomposition is *maximal* if there is a vertex $v\in X_\mathsf{i}$ that is not contained in any other set X_j. That is, a leaf is maximal if and only if there is a vertex v that is not contained in i's neighbor in T. We call a tree-decomposition *minimal* if it is non-trivial and all its leaves are maximal. It is easily verified that each graph of treewidth ℓ has a minimal tree-decomposition of width ℓ.

Lemma 12 [11] *The number of nodes in a minimal tree-decomposition of an N-vertex graph is at most $(2N-1)^2$.*

Lemma 13 [11] *The number of nodes in a minimal path-decomposition of an N-vertex graph is at most $2N+1$.*

We define the *restriction* of a partial tree-decomposition $\mathcal{E}$ rooted at node i analogous to the restriction of a partial path-decomposition. We define the *trunk* of a partial tree-decomposition as the tree obtained from the restriction by recursively removing all non-maximal leaves and compressing each path of degree-2 vertices to a single edge.

Lemma 14 [11] *The size of the trunk is at most $2k$.*

Each edge e of the trunk represents a path of edges of 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 by its corresponding path-decomposition Y_e. Let Z_e be the interval model of this path-decomposition. 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. 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 $\tau[y_e]$ is the typical list of Y_e.

Lemma 15 [11] *The characteristic of a partial tree-decomposition rooted at a node i has constant size.*

Lemma 16 [11] *The full set of characteristics $FS(\mathsf{i})$ of partial tree-decompositions rooted at a node i has constant size.*

The linear time algorithm for testing whether a given graph with nice tree-decomposition $\mathcal{D}=(\mathcal{X},T)$ has treewidth at most ℓ given in [11] processes the tree T bottom-up and computes for each node its full set of characteristics from the full sets computed for its children; for a leaf, the full set of characteristics is generated by brute force, which takes constant time, as there is only a constant number of minimal tree-decompositions to be generated and tested whether they have width at most ℓ. In external memory, this algorithm takes $O(sort(N))$ I/Os using the techniques of Section 3 and the fact that we send only a constant amount of information from each node to its parent (Lemmas 15 and 16). After the next theorem, we describe the computation for each type of nodes in order to describe the subsequent construction of a tree-decomposition of width at most ℓ.

Theorem 3 *Given a graph $G=(V,E)$, two constants k,ℓ, and a tree-decomposition $\mathcal{D}=(\mathcal{X},T)$ of width k for G, it takes $O(sort(N))$ I/Os and $O(N/B)$ blocks of external memory to decide whether G has tree-width at most ℓ.*

Proof. This follows from [11], Lemmas 15 and 16, and Corollary 1. □

Start nodes: We 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 $FS(\mathsf{i})$. This takes constant time as there are only a constant number of these tree-decompositions by Lemma 12. The trunk $\mathcal{T}$ of $\mathcal{F}$ is obtained by removing all nodes of degree 2 from H. For each 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 $|Y_e^{(i)}|$. That is $\tau[y_e] = [y_e]$.

Join nodes: If i is a join node with children j and k, we compute $FS(\mathsf{i})$ from $FS(\mathsf{j})$ and $FS(\mathsf{k})$ as follows. First observe that $X_\mathsf{i} = X_\mathsf{j} = X_\mathsf{k}$. We take all pairs of characteristics in $FS(\mathsf{j}) \times FS(\mathsf{k})$ with the same tree model. That is 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 $e \in \mathcal{T}$, we compute a list $[a_e^*] = (\tau(a_e^{(1)}) - |Z_e^{(1)}|, \tau(a_e^{(2)}) - |Z_e^{(2)}|, \dots)$. We compute all lists $\tau[c_e]$, where $[c_e] \in [a_e^*] \oplus \tau[b_e]$ and $\max[c_e] \leq \ell + 1$, and add $(\mathcal{T}, (Z_e)_{e\in\mathcal{T}}, (\tau[c_e])_{e\in\mathcal{T}})$ to the set $FS(\mathsf{i})$.

Forget nodes: If i is a forget node with child j and $X_\mathsf{i} = X_\mathsf{j} \setminus \{x\}$, we compute the full set $FS(\mathsf{i})$ from $FS(\mathsf{j})$ as follows. From each characteristic $(\mathcal{T}, (Z_e)_{e\in\mathcal{T}}, (\tau[y_e])_{e\in\mathcal{T}}) \in FS(\mathsf{j})$, we generate a characteristic $(\mathcal{T}^*, (Z_e^*)_{e\in\mathcal{T}^*}, (\tau[y_e^*])_{e\in\mathcal{T}^*}) \in FS(\mathsf{i})$. First we remove the vertex x from all sets $Z_e^{(q)}$ and compute the new trunk $\mathcal{T}^*$. We remove repetitions from the interval models Z_e and let the resulting interval model be Z_e^* for each edge $e \in \mathcal{T}^*$. Finally, we change the typical list $\tau[y_e]$ into $\tau[y_e^*]$ as described next.

Consider an interval model $Z_e = (Z_e^{(1)}, \dots, Z_e^{(s)})$. If the vertex x is contained in some sets of Z_e, then these sets have to be consecutive. That is, x is contained in sets $Z_e^{(a)}, \dots, Z_e^{(b)}$. As all sets in Z_e are different, the removal of x can cause at most two pairs of sets to become equal: $Z_e^{(a-1)} = Z_e^{(a)} \setminus \{x\}$ and $Z_e^{(b)} \setminus \{x\} = Z_e^{(b+1)}$. In this case, we would remove $Z_e^{(a)}$ or $Z_e^{(b)}$ or both from Z_e to obtain Z_e^*. In total, these are four different scenarios, which we discuss next:

1. If $|Z_e^*| = |Z_e|$, let $\tau[y_e^*] = \tau[y_e]$.
2. If $Z_e^{(a-1)} = Z_e^{(a)} \setminus \{x\}$, let
$$\tau^* = \tau(\tau(y_e^{(a-1)}) \circ \tau(y_e^{(a)}))$$
and
$$\tau[y_e^*] = (\tau(y_e^{(1)}), \dots, \tau(y_e^{(a-2)}), \tau^*, \tau(y_e^{(a+1)}), \dots, \tau(y_e^{(s)})).$$
3. If $Z_e^{(b)} \setminus \{x\} = Z_e^{(b+1)}$, we 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
$$\tau_1 = \tau(\tau(y_e^{(a-1)}) \circ \tau(y_e^{(a)})) \text{ and } \tau_2 = \tau(\tau(y_e^{(b)}) \circ \tau(y_e^{(b+1)})),$$
and
$$\tau[y_e^*] = (\tau(y_e^{(1)}), \dots, \tau(y_e^{(a-2)}), \tau_1, \tau(y_e^{(a+1)}), \dots, \tau(y_e^{(b-1)}), \tau_2, \tau(y_e^{(b+2)}), \dots, \tau(y_e^{(s)})).$$

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

Introduce nodes: If i is an introduce node with child j and $X_\mathsf{i} = X_\mathsf{j} \cup \{x\}$, we compute $FS(\mathsf{i})$ by first computing all minimal tree-decompositions of $G[X_\mathsf{i}]$. As for a start node, this can be done by brute force, as there are only a constant number of such tree-decompositions, each of which has a constant size characteristic. For each such tree-decomposition, we remove vertex x from all subsets, resulting in a tree-decomposition for $G[X_\mathsf{j}]$. We compute the tree-model of this decomposition. We check if there is a characteristic with the same tree-model in $FS(\mathsf{j})$. If this is the case, we change this characteristic into a new characteristic for $FS(\mathsf{i})$.

Let $\mathcal{F}^*$ be a tree-decompostion of $G[X_\mathsf{i}]$, and let $\mathcal{F}$ be the resulting tree-decomposition of $G[X_\mathsf{j}]$. Let $C(\mathcal{F}) = (\mathcal{T}, (Z_e)_{e\in\mathcal{T}}, (\tau[y_e])_{e\in\mathcal{T}})$ and $C(\mathcal{F}^*) = (\mathcal{T}^*, (Z_e)_{e\in\mathcal{T}}, (\tau[y_e])_{e\in\mathcal{T}})$. We show how to compute characteristics $(\mathcal{T}^*, (Z_e^*)_{e\in\mathcal{T}^*}, (\tau[y_e^\circ])_{e\in\mathcal{T}^*})$ for $FS(\mathsf{i})$:

1. If $\mathcal{T} = \mathcal{T}^*$, we change the interval model and typical list for each edge of the trunk as follows: Let $Z_e = (Z_e^{(1)}, \ldots, Z_e^{(s)})$ and $\tau[y_e] = (\tau(y_e^{(1)}), \ldots, \tau(y_e^{(s)}))$ be the interval model and typical list for edge e in $\mathcal{T}$. Analogously, let $Z_e^* = (Z_e^{*(1)}, \ldots, Z_e^{*(t)})$ be the interval model for edge e in $\mathcal{T}^*$. Let $a \le b$ be such that $Z_e^{*(a)}$ and $Z_e^{*(b)}$ are the first and last sets in Z_e^*, respectively, containing the introduced vertex x. Analogous to the discussion for a forget node, $s \le t \le s+2$. Hence, we distinguish the following four cases:

   (a) If $s = t$, we change the typical list $\tau[y_e]$ into a list $\tau[y_e^\circ] = (\tau(y_e^{(1)}), \ldots, \tau(y_e^{(a-1)}), 1 + \tau(y_e^{(a)}), \ldots, 1 + \tau(y_e^{(b)}), \tau(y_e^{(b+1)}), \ldots, \tau(y_e^{(s)}))$.

   (b) If $t = s+1$ and $Z_e^{*(a-1)} = Z_e^{*(a)} \setminus \{x\}$, write the typical list of Z_e as follows: $\tau[y_e] = (\tau(y_e^{(1)}), \ldots, \tau(y_e^{(a-2)}), \tau(y_e^{(a)}), \ldots, \tau(y_e^{(t)}))$. Consider all possible splits of the typical sequence $\tau(y_e^{(a)}) = (y_1, \ldots, y_r)$ 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 $\tau_2 = (y_{f+1}, \ldots, y_r)$. For each such split change the typical list $\tau[y_e]$ into a list

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

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

   (d) If $t = s+2$, write

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

   In this case, we consider all possible splits of $\tau(y_e^{(a)})$ into two sequences τ_1 and τ_2 as in Case (b) and all possible splits of $\tau(y_e^{(b)})$ into two sequences τ_3 and τ_4, as in Case (c). For each such pair of splits, we change the typical sequence $\tau[y_e]$ into the sequence

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

1: Run the testing algorithm of Section 5.3.1.
2: **if** G has width at most ℓ **then**
3: Build a tree of characteristics:
For every node i of T choose a characteristic $C_\mathsf{i} \in FS(\mathsf{i})$. For a forget or introduce node i with child j, C_i is chosen 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, then C_i has been produced from C_j and C_k during the construction of $FS(\mathsf{i})$.
4: Compute an implicit representation of $\mathcal{E}$:
This implicit representation consists of two parts: (1) a flippable DAG $\mathcal{G}$ whose connected components represent the path-decompositions defined by the maximal paths in U having only degree-2 nodes as internal nodes; (2) a link list $\mathcal{L}$ connecting these path-decompositions into the tree U.
5: Compute $\mathcal{E}$ explicitly:
Perform time-forward processing in $\mathcal{G}$ to compute all path-decompositions mentioned in the previous step and link them together to form U.
6: **end if**

Algorithm 3: Improving the tree-decomposition.

Add all characteristics $(\mathcal{T}^*, (Z^*_e)_{e\in\mathcal{T}^*}, (\tau[y^\circ_e])_{e\in\mathcal{T}^*})$ with $\max(\tau[y^\circ_e]) \le \ell+1$ for all edges $e \in \mathcal{T}^*$ to $FS(\mathsf{i})$. Note that in the last case, if $a = b$, we split $\tau(y^{(a)}_e)$ into three parts τ_1, τ_2, τ_3 and replace $\tau(y^{(a)}_e)$ by the sequence $\tau_1, 1+\tau_2, \tau_3$.

2. If $\mathcal{T} \neq \mathcal{T}^*$, the trunk $\mathcal{T}^*$ contains a leaf a which is not a leaf of $\mathcal{T}$. Let b be the neighbor of a in $\mathcal{T}^*$. If b is of degree three in $\mathcal{T}^*$, b is not a node of $\mathcal{T}$. In this case, we do the following: Let c and d be the other neighbors of b in $\mathcal{T}^*$. Note that c and d are adjacent in $\mathcal{T}$. let Z_b be the set corresponding to node b in T^*, and $Z_e = (Z^{(1)}_e, \ldots, Z^{(q)}_e, \ldots, Z^{(s)}_e)$ be the interval model for edge $e = (\mathsf{c}, \mathsf{d})$ in $\mathcal{T}$, where $Z^{(q)}_e = Z_\mathsf{b}$. In $\mathcal{T}^*$ we split this interval model into two parts $Z^*_{e_1} = (Z^{(1)}_{e_1}, \ldots, Z^{(q)}_{e_1})$ and $Z^*_{e_2} = (Z^{(q)}_{e_1}, \ldots, Z^{(s)}_{e_1})$, 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 $\tau[y_e] = (\tau(y^{(1)}_e), \ldots, \tau(y^{(q)}_e), \ldots, \tau(y^{(s)}_e))$. We split the typical sequence $\tau(y^{(q)}_e) = (y_1, \ldots, y_s)$ for interval $Z^{(q)}_e$ in all possible ways to obtain two sequences $\tau_1 = (y_1, \ldots, y_f)$ and $\tau_2 = (y_f, \ldots, y_s)$. Then the typical lists for e_1 and e_2 in $\mathcal{T}^*$ are $\tau[y^*_{e_1}] = (\tau(y^{(1)}_e), \ldots, \tau(y^{(q-1)}_e), \tau_1)$ and $\tau[y^*_{e_2}] = (\tau_2, \tau(y^{(q+1)}_e), \ldots, \tau(y^{(s)}_e))$, respectively. Each of these splits produces one new element in $FS(\mathsf{i})$. If node b exists in $\mathcal{T}$, we do not need to split any interval model or typical list. It remains to reattach edge $f = (\mathsf{a}, \mathsf{b})$ to $\mathcal{T}^*$. The interval model Z_f is some list $Z_f = (Z^{(1)}_f, \ldots, Z^{(t)}_f)$ with $Z^{(1)}_f = Z_\mathsf{b}$, as computed as part of the minimal tree-decomposition of $G[X_\mathsf{i}]$. The typical sequence for each set $Z^{(q)}_f \in Z_f$ is $\tau(y^{\circ(q)}_f) = (|Z^{(q)}_f|)$.

5.3.2 Constructing a Tree-decomposition

In this section we describe how to construct a tree-decomposition $\mathcal{E} = (\mathcal{Y}, U)$ of width at most ℓ for a graph G, given a nice tree-decomposition $\mathcal{D} = (\mathcal{X}, T)$ of width $k \ge \ell$ of G. Our algorithm proceeds in four phases, as sketched in Algorithm 3. Next we describe each of these phases in detail.

Phase 1 — An augmented test algorithm In order to facilitate subsequent phases of our algorithm, we need to augment the algorithm of Section 5.3.1 to compute additional information.

Let i be a node of T with child j (i.e., a forget or introduce node). For every characteristic $C \in FS(\mathsf{i})$, we store a "pointer" to the characteristic $C' \in FS(\mathsf{j})$ from which C has been produced by applying the rules described in Section 5.3.1. 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 two pointers to the two characteristics in the full sets of its two children from which C has been produced.

For a join node i with children j and k, we computed a characteristic $(\mathcal{T},(Z_e)_{e\in\mathcal{T}},(\tau[y_e])_{e\in\mathcal{T}}) \in FS(\mathsf{i})$ from two characteristics $(\mathcal{T},(Z_e)_{e\in\mathcal{T}},(\tau[a_e])_{e\in\mathcal{T}}) \in FS(\mathsf{j})$ and $(\mathcal{T},(Z_e)_{e\in\mathcal{T}},(\tau[b_e])_{e\in\mathcal{T}}) \in FS(\mathsf{k})$. Consider the computation for a particular edge $e \in \mathcal{T}$. Let $Z_e = (Z_e^{(1)},\ldots,Z_e^{(s)})$ and $\tau[y_e] = (\tau(y_e^{(1)}),\ldots,\tau(y_e^{(s)}))$. Each sequence $\tau(y_e^{(q)})$ is computed from two sequences a^* and $\tau(b_e^{(q)})$, where $a^* = \tau(a_e^{(q)}) - |Z_e^{(q)}|$. Then $\tau(y_e^{(q)}) = \tau(y^\circ)$, where $y^\circ \in a^* \oplus \tau(b_e^{(q)})$, i.e., $y^\circ = a^\circ + b^\circ$, where $a^\circ \in E(a^*)$ and $b^\circ \in E(\tau(b_e^{(q)}))$. Let $a^* = (a_1,\ldots,a_n)$ and $\tau(b_e^{(q)}) = (b_1,\ldots,b_{n'})$. 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 $b_{q(b_f^\circ)} \in \tau(b_e^{(q)})$ of which b_f° is a copy. As $y_f^\circ = a_f^\circ + b_f^\circ$, we define $p(y_f^\circ) = p(a_f^\circ)$ and $q(y_f^\circ) = q(b_f^\circ)$. Finally, every element $y_h \in \tau(y_e^{(q)})$ corresponds to an interval of elements $y_{f_h}^\circ,\ldots,y_{g_h}^\circ$ in y° such that $y_f^\circ \le y_h$, for $f_h \le f \le g_h$. Let $r(y_h) = f_h$. This information can easily be computed during the computation of all typical lists when constructing the full set for a join node. Note that $|y^\circ| \le |\tau(a_e^{(q)})| + |\tau(b_e^{(q)})| - 1 = O(1)$.

Similarly, we will need for every characteristic in the full set of an introduce node or forget node, some information about how the characteristic has been produced from the corresponding characteristic at the child. We do not formalize this information here, as it is easy to see that it can be maintained using $O(1)$ additional space per characteristic.

Phase 2 — Building a tree of characteristics If G has treewidth at most ℓ, the full set $FS(\mathsf{r})$ of the root r of T is non-empty. We pick a characteristic $C_\mathsf{r} \in FS(\mathsf{r})$. Note that C_r has been computed recursively from a set of characteristics at the leaves of T. We discard all characteristics in all full sets except one per node. These characteristics are chosen in a top-down fashion: For the root r, we pick C_r arbitrarily. Now consider a node i for which we have chosen a characteristic $C_\mathsf{i} \in FS(\mathsf{i})$. If i is a join node with children j and k, C_i stores two pointers to characteristics $C_\mathsf{j} \in FS(\mathsf{j})$ and $C_\mathsf{k} \in FS(\mathsf{k})$. These are the characteristics we choose for nodes j and k. If i is a forget or introduce node with child j, C_i stores a pointer to a characteristic $C_\mathsf{j} \in FS(\mathsf{j})$. This is the characteristic we choose for node j. If i is a start node, it does not have any children.

The structure of a tree-decomposition for G Before we start describing Phases 3 and 4, we first show 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 at every node i a partial tree-decomposition $\mathcal{E}_\mathsf{i}$ rooted at i which has width at most ℓ. The bound on the width of $\mathcal{E}_\mathsf{i}$ follows immediately, if we can associate each set Y_l in the tree-decomposition 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 $Z_e = (Z_e^{(1)},\ldots,Z_e^{(s)})$. Recall that the path-decomposition Y_e corresponding to edge e is just the list Z_e. That is, a tree-decomposition $\mathcal{E}_\mathsf{i}$ of G_i is given by taking the trunk and replacing every edge e by the path-decomposition Z_e. Note that every typical sequence $\tau(y_e^{(q)})$ consists of a single element y_1 and that this element y_1 corresponds to a single set Y_1 in the path-decomposition Y_e.

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, $\mathcal{E}_\mathsf{i}$ may contain one more node than $\mathcal{E}_\mathsf{j}$ as discussed below.

Although tree-decompositions $\mathcal{E}_\mathsf{i}$ and $\mathcal{E}_\mathsf{j}$ are the same if $\mathcal{T} = \mathcal{T}^*$, the way this tree-decomposition corresponds to entries in the characteristics changes. The parts of $\mathcal{E}_\mathsf{j}$ represented in C_j are also represented in C_i. In particular, 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 $y_h \in \tau(y_e^{(q)})$, then for all sets Y_l in the interval Y_{y_h} corresponding to y_h, $X_i \cap Y_l = Z_e^{(q)}$ and $|Y_l| \leq y_h$. 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 can be handled analogously.) Then we first computed a list $y^\circ = \tau(y_e^{(a-1)}) \circ \tau(y_e^{(a)})$, and every entry $y_f^\circ \in y^\circ$ represents the same interval of Y_e as its corresponding entry y_h in $\tau(y_e^{(a-1)})$ or $\tau(y_e^{(a)})$. That is, $Y_{y_f^\circ} = Y_{y_h}$. In particular, for every set $Y_l \in Y_{y_f^\circ}$, $|Y_l| \leq y_f^\circ$. Now $\tau(y_e^{*(a)}) = \tau(y^\circ)$. That is, $\tau(y_e^{*(a)})$ is derived from y° by the application of typical operations and the removal of consecutive duplicates. This way, every element y_k in $\tau(y_e^{*(a)})$ corresponds to an interval $I(y_k)$ of elements in y°, such that for every element $y_f^\circ \in I(y_k)$, $y_f^\circ \leq y_k$. We let $Y_{y_k} = \bigcirc_{y_f^\circ \in I(y_k)} Y_{y_f^\circ}$. As for each element $y_f^\circ \leq y_k$ and for every set $Y_l \in Y_{y_f^\circ}$, $|Y_l| \leq y_f^\circ$, it follows that for every set $Y_l \in Y_{y_k}$, $|Y_l| \leq y_k$.

If $\mathcal{T} \neq \mathcal{T}^*$, a whole path of the partial tree-decomposition represented in C_j is no longer represented in C_i. This means that this path is no longer involved in any modifications, and it stays a valid path-decomposition. If the non-leaf endpoint b of the removed edge is of degree 3 in $\mathcal{T}$, it is not a node of $\mathcal{T}^*$, and the two other neighbors c and d of b in $\mathcal{T}$ are linked by an edge (c,d) in $\mathcal{T}^*$. We computed the characteristic of the path-decomposition $Y_{(\mathsf{c},\mathsf{d})}$ by concatenating the characteristics of path-decompositions $Y_{(\mathsf{c},\mathsf{b})}$ and $Y_{(\mathsf{b},\mathsf{d})}$ and then applying typical operations and duplicate removals to the necessary sets. Now we compute $Y_{(\mathsf{c},\mathsf{d})} = Y_{(\mathsf{c},\mathsf{b})} \circ Y_{(\mathsf{b},\mathsf{d})}$ and then compute for every element $y_h \in \tau[y_{(\mathsf{c},\mathsf{d})}]$ its corresponding interval of $Y_{(\mathsf{c},\mathsf{d})}$ as described above. Note that this operation corresponds to duplicating the vertex of U corresponding to b. This is not necessary; but we treat this case this way, as it simplifies Phase 3 of our algorithm. 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 edges in $\mathcal{T}^*$, the representation of Y_e by the elements of $\tau[y_e^*]$ is defined as in the case $\mathcal{T} = \mathcal{T}^*$ and $|Z_e| = |Z_e^*|$. Hence, in all cases, it remains true that every element y_h in a typical sequence of the characteristic represents an interval of a path-decomposition Y_e in the filled trunk and every set in this interval has size at most 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 any more and thus remain valid. So let us see how the "stretching" is done:

Consider 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, we have to duplicate certain entries in A_e and B_e such 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 $\tau(a_e^{(q)})$ and $\tau(b_e^{(q)})$ in C_j and C_k, respectively. Let $a^* = (a_1, \ldots, a_n)$, $\tau(b_e^{(q)}) = (b_1, \ldots, b_{n'})$, $\tau(y_e^{(q)}) = (y_1, \ldots, y_m)$, and $y^\circ = (y_1^\circ, \ldots, y_c^\circ)$. Recall that $y^\circ = a^\circ + b^\circ$, where a° and b° are extensions of a^* and $\tau(b_e^{(q)})$, respectively. Initally, there are paths A_{a_h}

and B_{b_h} associated with each entry $a_h \in a^*$ and $b_h \in \tau(b_e^{(q)})$.

We first 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 $\bar{B}_{b_f^\circ}$. We first assign $\bar{A}_{a_f^\circ} = A_{a_f^\circ}$ and then 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 $\bar{B}_{b_f^\circ} = (\bar{B}_1, \ldots, \bar{B}_r)$. Then we define $Y_{y_f^\circ} = (Y_1, \ldots, Y_r)$, where $Y_l = \bar{A}_l \cup \bar{B}_l$. Finally, we obtain the path-decomposition associated with every element y_h as

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

Clearly, the resulting tree-decomposition is valid. It remains to show that for every set $Y_l \in Y_{y_h}$, $|Y_l| \leq y_h$. However, it is easily verified that for each a_f°, every set in $\bar{A}_{a_f^\circ}$ has size at most $a_f^\circ + |Z_e^{(q)}|$, and for each b_f°, 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 handle the two different cases for the trunks separately. If $\mathcal{T} = \mathcal{T}^*$, we have to update the path-decompositions for all edges $e \in \mathcal{T}$. If $|Z_e| = |Z_e^*|$, let $\tau[y_e] = (\tau(y_e^{(1)}), \ldots, \tau(y_e^{(q)}))$ and $\tau[y_e^*] = (\tau(y_e^{(1)}, \ldots, \tau(y_e^{(a-1)}), 1 + \tau(y_e^{(a)}), \ldots, 1 + \tau(y_e^{(b)}), \tau(y_e^{(b+1)}), \ldots, \tau(y_e^{(q)}))$. 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 analogous.) In this case, we split the typical sequence $\tau(y_e^{(a)})$ into two sequences τ_1 and τ_2, associating with every element in τ_1 and τ_2 the same subsequence of Y_e as with the corresponding element in $\tau(y_e^{(a)})$. If the split of sequence $\tau(y_e^{(a)})$ is of type one, 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 $\tau(y_e^{(a)})$ 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 $\mathcal{T}$ is all that has to be done. Otherwise, we need to split the path-decomposition $Y_{(\mathsf{c},\mathsf{d})}$ to obtain two path-decompositions $Y_{(\mathsf{c},\mathsf{b})}$ and $Y_{(\mathsf{b},\mathsf{c})}$. Note that the split is of type one, so that we have to perform the 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 17 *The tree-decomposition $\mathcal{E}_\mathsf{i}$ defined for node i is a tree-decomposition for G_i.*

Proof. We have to verify Properties T1–T3, which is easily done for Properties T1 and T3. To prove Property T2, we distinguish the four different node types of $\mathcal{D}$.

For a start node i, Property T2 is obviously satisfied, as $\mathcal{E}_\mathsf{i} = \mathcal{F}$, where $\mathcal{F}$ is the tree-decomposition of $G_\mathsf{i} = G[X_\mathsf{i}]$ from which C_i has been derived.

For a forget node i with child j, we start with a tree-decomposition $\mathcal{E}_\mathsf{j}$ for G_j and possibly augment it by duplicating a node. 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. All edges of 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 $\mathcal{F}$, 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 $\mathcal{F}$ although possibly not characteristic C_i. 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 viewed as an extension of $\mathcal{E}_\mathsf{j}$ as well as $\mathcal{E}_\mathsf{k}$. As $G_\mathsf{i} = G_\mathsf{j} \cup G_\mathsf{k}$, 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 18 *The tree-decomposition $\mathcal{E} = \mathcal{E}_\mathsf{r}$ has size $O(N)$.*

Proof. We begin with some observations:

As we start by constructing a minimal tree-decomposition at every start node i, $|\mathcal{E}_\mathsf{i}| \leq (2k+1)^2$ by Lemma 12.

At a forget node i with child j, $|\mathcal{E}_\mathsf{i}| \leq |\mathcal{E}_\mathsf{j}| + 1$. 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, normally the tree-decompositions would be the same; but when merging path-decompositions $Y_{(\mathsf{c},\mathsf{b})}$ and $Y_{(\mathsf{b},\mathsf{d})}$, the node corresponding to b is represented once in $Y_{(\mathsf{c},\mathsf{b})}$ and once in $Y_{(\mathsf{b},\mathsf{d})}$, and we simply concatenate path-decompositions $Y_{(\mathsf{c},\mathsf{b})}$ and $Y_{(\mathsf{b},\mathsf{d})}$ without removing this duplicate. In all other cases, the tree-decomposition does not change.

At an introduce node i with child j, $|\mathcal{E}_\mathsf{i}| \leq |\mathcal{E}_\mathsf{j}| + 4k$. Indeed, if $\mathcal{T} = \mathcal{T}^*$, the modifications made for every edge do not alter the size of the tree-decomposition except the type-one splits we perform. A type-one split leads to the duplication of the vertex corresponding to the shared element y_f of the two sequences resulting from the split. As we perform at most two type-one splits per edge of $\mathcal{T}$ and $|\mathcal{T}| \leq 2k$ by Lemma 14, the size of the tree-decomposition increases by at most $4k$ in this case. If $\mathcal{T} \neq \mathcal{T}^*$, we possibly perform a type-one split of an edge of $\mathcal{T}$ and then attach the path-decomposition corresponding to the attached edge of $\mathcal{T}^*$. The type-one split increases the size of the path-decomposition by one; the attached path-decomposition has length at most $2k+3$ by Lemma 13.

At a join node i with children j and k, $|\mathcal{E}_\mathsf{i}| \leq |\mathcal{E}_\mathsf{j}| + |\mathcal{E}_\mathsf{k}|$. 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 $|Y| \leq |A| + |B| - 1$. Applying this claim to all intervals in the tree-model gives the above claim. Let a° and b° be the usual extensions of a^* and $\tau(b_e^{(q)})$ and $y^\circ = a^\circ + b^\circ$. By Lemma 4, $|y^\circ| \leq |a^*| + |\tau(b_e^{(q)})| - 1$. For every element $a_h \in a^*$ ($b_h \in \tau(b_e^{(q)})$), let A_h (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 $a_f^\circ \in a^\circ$ and $b_f^\circ \in b^\circ$ let A_f° and B_f° be the corresponding path-decompositions. Observe that all path-decompositions A_h appear exactly once in the list of path-decompositions A_f°, and all other path-decompositions A_f° have size one. Hence, $\sum_{f=1}^{c} |A_f^\circ| \leq |A| + (c-n)$. Analogously, $\sum_{f=1}^{c} |B_f^\circ| \leq |B| + (c-n')$. Finally, for every path-decomposition

Y_f° corresponding to element y_f°, $|Y_f^\circ| = \max\{|A_f^\circ|, |B_f^\circ|\} \leq |A_f^\circ| + |B_f^\circ| - 1$. That is,

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

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 $|\mathcal{E}| = |\mathcal{E}_\mathsf{r}| \leq (2k+1)^2|T_\mathsf{r}| \leq 4(2k+1)^2 N = O(N)$, which proves the lemma.

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

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

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

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

Phase 3 — Constructing the tree-decomposition implicitly The goal of the next two phases is to construct for every node i of T exactly the tree-decomposition $\mathcal{E}_\mathsf{i}$ of G_i. Before we start describing the third phase of our algorithm, we define the implicit representation of these tree-decompositions that we are going to use in this phase. The fourth phase will use the implicit representation derived for $\mathcal{E}_\mathsf{r} = \mathcal{E}$ to construct $\mathcal{E}$ explicitly.

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 $Z_e^{(q)} \in Z_e$. Then an entry $y_h \in \tau(y_e^{(q)})$ corresponds to a path-decomposition Y_{y_h} which is part of $\mathcal{E}_\mathsf{i}$. We represent each such path-decomposition Y_{y_h} by a flippable DAG $\mathcal{G}(y_h)$, from which we can reconstruct Y_{y_h} easily. For the parts of $\mathcal{E}_\mathsf{i}$ that are not in the filled trunk of C_i, a similar representation has been computed at the descendants of i, and they are linked to the nodes of the trunk nodes by *link records* (α, β) stored in a link list $\mathcal{L}$. Every graph $\mathcal{G}(y_h)$ has the following properties:

(G1) $\mathcal{G}(y_h)$ is a valid flippable DAG representing a series parallel *st*-graph.

(G2) Every node $\alpha \in \mathcal{G}(y_h)$ is labeled with a triple $(L_\alpha, R_\alpha, \rho_\alpha)$.

(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 $1 \leq l \leq r$.

The source and sink of $\mathcal{G}(y_h)$ are stored with y_h and are only written to disk when $\mathcal{G}(y_h)$ is merged with other graphs $\mathcal{G}(y_k)$ to form a larger graph $\mathcal{G}(y'_l)$. Moreover, y_h stores a label $\kappa(y_h) = |Y_{y_h}|$ and colors $c(\sigma)$ and $c(\tau)$ that help us to extend $\mathcal{G}(y_h)$ properly. We now describe how to construct this information bottom-up in T. Once we are at the root, we will have constructed the necessary information to be fed into the next phase of the algorithm, which will construct the desired tree-decomposition explicitly. As before, we deal with the four different node types separately:

Start nodes: Consider a start node i, For this node, the above information can easily be set up. For the single entry $y_1 \in \tau(y_e^{(q)})$, the graph $\mathcal{G}(y_1)$ consists of a single node $\sigma = \tau$ with $L_\sigma = R_\tau = Z_e^{(q)}$ and $\rho_\sigma = 1$; $\kappa(y_1) = 1$, $c(\sigma) = c(\tau) = \mathsf{red}$.

Forget nodes: If i is a forget node with child j, we distinguish two cases again, depending on whether or not $\mathcal{T} = \mathcal{T}^*$. Let $C_\mathsf{i} = (\mathcal{T}^*, (Z_e^*)_{e\in\mathcal{T}^*}, (\tau[y_e^*])_{e\in\mathcal{T}^*})$ and $C_\mathsf{j} = (\mathcal{T}, (Z_e)_{e\in\mathcal{T}}, (\tau[y_e])_{e\in\mathcal{T}})$. If $\mathcal{T} = \mathcal{T}^*$, consider a path-decomposition Y_e for 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 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 can be handled analogously.) For the sake of description, write

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

For $q < a-1$ and $q > a$, $\tau(y_e^{*(q)}) = \tau(y_e^{(q)})$. For the entries in these sequences, we just copy the information. For $q = a$, recall that $\tau(y_e^{*(a)}) = \tau(\tau(y_e^{(a-1)}) \circ \tau(y_e^{(a)}))$. In particular, we first constructed a sequence $y^\circ = \tau(y_e^{(a)}) \circ \tau(y_e^{(a+1)})$ and then computed $\tau(y_e^{*(a)}) = \tau(y^\circ)$. Every entry in y° inherits its labels from its corresponding entry in $\tau(y_e^{(a)})$ or $\tau(y_e^{(a+1)})$, respectively. Now $\tau(y_e^{*(a)})$ is computed from y° by means of two operations: duplicate removal and typical operations. Finally, every entry $y_h \in \tau(y_e^{*(a)})$ corresponds to an interval $y_k^\circ, \ldots, y_l^\circ$ of entries in y°. We compute $\kappa(y_h) = \sum_{f=k}^{l} \kappa(y_f^\circ)$ and store with y_h the source node of $\mathcal{G}(y_k^\circ)$ and the sink node of $\mathcal{G}(y_l^\circ)$. Moreover, we have to concatenate graphs $\mathcal{G}(y_f^\circ)$, $k \leq f \leq l$, to form graph $\mathcal{G}(y_h)$. 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 the source of $\mathcal{G}(y_{f+1}^\circ)$, respectively. Then we add edges (τ,σ) and (σ,τ) to $\mathcal{G}(y_h)$ and compute $R_\tau \leftarrow R_\tau \setminus L_\sigma$ and $L_\sigma \leftarrow L_\sigma \setminus R_\tau$. Edge (τ,σ) is assigned color $(c(\tau), c(\sigma))$; edge (σ,τ) is assigned color $(\overline{c}(\sigma), \overline{c}(\tau))$.

If $\mathcal{T} \neq \mathcal{T}^*$, there is a leaf a of $\mathcal{T}$ which 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 to detach the edge (a,b) from $\mathcal{T}$. Otherwise, let c and d be the other two neighbors of b in $\mathcal{T}$. Then (c,d) is an edge of $\mathcal{T}^*$, and the characteristic of edge (c,d) has been computed by concatenating the characteristics of edges (c,b) and (b,d) and then applying the same operations as for the case $\mathcal{T} = \mathcal{T}^*$ to edge (c,d). Once the concatenation has been performed, we proceed as in the case $\mathcal{T} = \mathcal{T}^*$.

The concatenation of characteristics, however, requires some closer consideration. The problem we are facing 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 instead of from c to b, while $\tau[y_{(\mathsf{b},\mathsf{d})}]$ is sorted from b to d. This means that the path-decomposition $Y_{(\mathsf{b},\mathsf{c})}$ is currently sorted from b to c; but we need to sort it from c to b in order to concatenate it with path-decomposition $Y_{(\mathsf{b},\mathsf{d})}$.

Let $\tau[y_{(\mathsf{b},\mathsf{c})}] = (\tau(y^{(1)}_{(\mathsf{b},\mathsf{c})}),\dots,\tau(y^{(s)}_{(\mathsf{b},\mathsf{c})}))$ and $\tau(y^{(q)}_{(\mathsf{b},\mathsf{c})}) = (y_1,\dots,y_r)$. Then we define $\tau[y_{(\mathsf{c},\mathsf{b})}]$ as $\tau[y_{(\mathsf{c},\mathsf{b})}] = (\tau(y^{(1)}_{(\mathsf{c},\mathsf{b})}),\dots,\tau(y^{(s)}_{(\mathsf{c},\mathsf{b})}))$, where $\tau(y^{(s-q+1)}_{(\mathsf{c},\mathsf{b})}) = (y_r, y_{r-1},\dots,y_1)$.

In order to properly flip the path-decomposition $Y_{(\mathsf{b},\mathsf{c})}$, we also have to flip all graphs $\mathcal{G}(y_h)$ along edge (b,c). Let the source and sink of $\mathcal{G}(y_h)$ be σ and τ. Then we flip $\mathcal{G}(y_h)$ by changing the colors of both vertices. To keep track of the flip, we also let the source pointer of y_h point to τ and the sink pointer of y_h point to σ. We also have to exchange sets L_α and R_α for all vertices $\alpha \in \mathcal{G}(y_h)$. We perform this exchange only for the source σ and the sink τ of $\mathcal{G}(y_h)$. We will see in the description of Phase 4 that an explicit exchange of the other sets is not required. In order to prove the correctness of this Phase, however, we assume that we exchange all these sets explicitly. Now we can proceed as above to concatenate graphs as needed.

It remains to describe how to "detach" an edge from $\mathcal{T}$. Take any other edge e incident to node b. W.l.o.g. assume that e is directed away from b and (a,b) is directed from b to a. Let y_1 be the first entry of $\tau(y^{(1)}_e)$, and let y'_1 be the first entry of $\tau(y^{(1)}_{(\mathsf{b},\mathsf{a})})$. Let σ and σ' be the sources of $\mathcal{G}(y_1)$ and $\mathcal{G}(y'_1)$, respectively. Then we add a link (σ,σ') to the link list $\mathcal{L}$. In addition, we concatenate all graphs $\mathcal{G}(y_h)$, $y_h \in \tau(y^{(q)}_{(\mathsf{b},\mathsf{a})})$ and $\tau(y^{(q)}_{(\mathsf{b},\mathsf{a})}) \in \tau[y_{(\mathsf{b},\mathsf{a})}]$, to a single graph $\mathcal{G}((\mathsf{b},\mathsf{a}))$ in the same way as we concatenated graphs $\mathcal{G}(y^\circ_f)$ to a graph $\mathcal{G}(y_h)$ above.

Introduce nodes: 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. As for forget nodes, we have two cases. If $\mathcal{T} = \mathcal{T}^*$, consider an edge $e \in \mathcal{T}$. If $|Z_e| = |Z^*_e|$, let $\tau[y_e] = (\tau(y^{(1)}_e),\dots,\tau(y^{(s)}_e))$ and $\tau[y^*_e] = (\tau(y^{(1)}_e),\dots,\tau(y^{(a-1)}_e), 1+\tau(y^{(a)}_e),\dots,1+\tau(y^{(b)}_e), \tau(y^{(b+1)}_e)),\dots,\tau(y^{(s)}_e))$. First we copy the information from each entry in $\tau[y_e]$ to the corresponding entry in $\tau[y^*_e]$. Now let $y_h \in \tau(y^{(q)}_e)$, $a \le q \le b$, and let σ and τ be the source and sink nodes stored with y_h. Then we set $L_\sigma \leftarrow L_\sigma \cup \{x\}$ and $R_\tau \leftarrow R_\tau \cup \{x\}$.

Now consider the case when $|Z^*_e| = |Z_e|+1$ and $Z^{*(a-1)}_e = Z^{*(a)}_e \setminus \{x\}$. (The two other cases can be handled analogously.) In this case, $\tau[y_e] = (\tau(y^{(1)}_e),\dots,\tau(y^{(s)}_e)$ and $\tau[y^*_e] = (\tau(y^{(1)}_e),\dots,\tau(y^{(a-1)}_e), \tau_1, 1+\tau_2, 1+\tau(y^{(a+1)}_e),\dots,1+\tau(y^{(b)}_e), \tau(y^{(b+1)}_e),\dots,\tau(y^{(s)}_e))$, where (τ_1,τ_2) is a split of $\tau(y^{(a)}_e)$ of either of the two types. For all typical lists except $\tau(y^{(a)}_e)$ 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 the elements in $\tau(y^{(a)}_e)$. We copy the necessary information and add the node x to sets L_σ and R_τ in τ_2 as above. If (τ_1,τ_2) is a type-one split, $\tau(y^{(a)}_e) = (y_1,\dots,y_r)$, $\tau_1 = (y_1,\dots,y_f)$, and $\tau_2 = (y_f,\dots,y_r)$. For $1 \le h < f$, we simply copy the information for y_h from $\tau(y^{(a)}_e)$ to τ_1. For all elements in τ_2, we copy the information from $\tau(y^{(a)}_e)$ to τ_2 and then add x to sets L_σ and R_τ as described above. Finally, for y_f in τ_1, we create a new graph $\mathcal{G}(y_f)$ with a single node $\sigma = \tau$ and set $L_\sigma \leftarrow R_\tau \leftarrow L_{\sigma'}$, $\rho_\sigma \leftarrow 1$, and $c(\sigma) = c(\tau) = \mathsf{red}$, where σ' is the sink node stored with y_f in τ_2 (before adding x). Also, we set $\kappa(y_f) \leftarrow 1$ in τ_1.

If $\mathcal{T}^* \neq \mathcal{T}$, $\mathcal{T}^*$ contains one edge from a leaf a to its neighbor b which 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, and we simply copy it. If b is not in $\mathcal{T}$, let c and d be the other two neighbors of b in $\mathcal{T}^*$. We performed a type-one split to obtain characteristics for edges (c,b) and (b,d) in $\mathcal{T}^*$. The necessary information stored with these edges can be computed similar to the procedure for type-one splits described above. Finally, let $e = (\mathsf{a},\mathsf{b})$ and $\tau[y_e] = (\tau(y^{(1)}_e),\dots,\tau(y^{(s)}_e))$. Then $\tau(y^{(q)}_e) = (|Z^{(q)}_e|)$, and we assign to the unique element $y_1 \in \tau(y^{(q)}_e)$ the graph $\mathcal{G}(y_1)$ consisting of a single node $\sigma = \tau$ with $L_\sigma = R_\tau = Z^{(q)}_e$, $\rho_\sigma = 1$, and $c(\sigma) = c(\tau) = \mathsf{red}$; $\kappa(y_1) = 1$.

Join nodes: Finally, 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 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.

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

Given the pointers $r(y_h)$ for all elements $y_h \in \tau(y_e^{(q)})$ as computed in Phase 1, the information stored with every element $y_h \in \tau(y_e^{(q)})$ 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, let $\kappa_a(y_f^\circ) = 1$. 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. Now we define $\kappa(y_f^\circ) = \max\{\kappa_a(y_f^\circ), \kappa_b(y_f^\circ)\}$. 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 define $\rho_{\sigma_a} \leftarrow \rho_{\sigma_a} + (\kappa(y_f^\circ) - \kappa_a(y_f^\circ))$ and $\rho_{\sigma_b} \leftarrow \rho_{\sigma_b} + (\kappa(y_f^\circ) - \kappa_b(y_f^\circ))$. We create two new vertices σ and τ and add edges between σ and σ_a and σ_b as well as between τ_a and τ_b and τ, 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 set $L_\sigma \leftarrow L_{\sigma_a} \cup L_{\sigma_b}$, $R_\sigma \leftarrow \emptyset$, $L_{\sigma_a} \leftarrow L_{\sigma_b} \leftarrow \emptyset$, $L_\tau \leftarrow \emptyset$, $R_\tau \leftarrow R_{\tau_a} \cup R_{\tau_b}$, $R_{\tau_a} \leftarrow R_{\tau_b} \leftarrow \emptyset$, $\rho_\sigma \leftarrow 0$, $\rho_\tau \leftarrow 0$, and $c(\sigma) = c(\tau) = \mathsf{red}$. The new edges are colored as follows:

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

Note that all parts of the constructed tree-decomposition are now represented as a collection of path-decompositions, except the filled trunk of the characteristic C_r, where r is the root of T. In order to simplify the description of the next phase, we decompose this filled trunk into a collection of path-decompositions as well. This can be done by detaching edges bottom-up until $\mathcal{T}$ consists of only a single node. The detachment of edges is done by the procedure used for detaching edge (a,b) from the trunk of a forget node. The following lemma shows that Phase 3 computes the correct input for Phase 4.

Lemma 19 *Let i be a node in T, $C_\mathsf{i} = (\mathcal{T}, (Z_e)_{e\in\mathcal{T}}, (\tau[y_e])_{e\in\mathcal{T}})$ be the characteristic stored at i, e an edge of $\mathcal{T}$, $\tau(y_e^{(q)})$ be a typical sequence in $\tau[y_e]$, and $y_h \in \tau(y_e^{(q)})$. 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 $Y_{y_h} = (Z_e^{(q)})$, for some q. Properties G1–G5 are easily now 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 induction hypothesis, all graphs $\mathcal{G}(y_h)$ stored with characteristic C_j have Properties G1–G5. Analogously, if i is a join node with children j and k, $|T_\mathsf{j}| < |T_i|$ and $|T_\mathsf{k}| < |T_\mathsf{i}|$, so that 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 our algorithm constructs graphs $\mathcal{G}(y_h)$ for the elements of the typical sequences in C_i correctly. It is easily verified that $\mathcal{G}(y_h)$ is series parallel.

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

Consider the path-decompositions $Y_e^{(a-1)}$ and $Y_e^{(a)}$ corresponding to intervals $Z_e^{(a-1)}$ and $Z_e^{(a)}$. By induction hypothesis, $Y_e^{(a-1)} = Y_{y_1} \circ Y_{y_2} \circ \cdots \circ Y_{y_m}$, where $\tau(y_e^{(a-1)}) = (y_1, \ldots, y_m)$ and for every $1 \leq h \leq m$, $\mathcal{G}(y_h)$ has Properties G1–G5, i.e., represents Y_{y_h} properly. Analogously, $Y_e^{(a)} = Y_{y'_1} \circ Y_{y'_2} \circ \cdots \circ Y_{y'_n}$, where $\tau(y_e^{(a)}) = (y'_1, \ldots, y'_n)$ 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 $Y_e^{*(a)} = Y_{y_1^\circ} \circ \cdots \circ Y_{y_{m+n}^\circ}$. Now, every element $y_h \in \tau(y_e^{*(a)})$ 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 induction hypothesis (on $|T_\mathsf{i}|$). So assume that $j > 0$. To show the validity of $\mathcal{G}_j$, we have to prove that the coloring of $\mathcal{G}_j$ is independent of the spanning tree, once we have picked a color for σ. This is true for $\mathcal{G}_{j-1}$ by 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 $\mathcal{G}(y_{k+h}^\circ)$ because every spanning tree of $\mathcal{G}_j$ must contain edge $\{\tau_{j-1}, \sigma_j\}$. By 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 valid. 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 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 $c(\sigma_j)$, as edge $\{\tau_{j-1}, \sigma_j)\}$ has color $(c(\tau_{j-1}), c(\sigma_j))$. By 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 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 $x \in R_{\nu(x)}$. As $\mathcal{G}(y_f^\circ)$ 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 all path-decompositions $Y_p^\circ, \ldots, Y_q^\circ$ must be a valid 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 $x \in L_{\sigma_f}$, $p < f \leq q$ and $x \in R_{\tau_f}$, $p \leq f < q$. 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 $x \in L_{\nu(x)}$. By induction hypothesis, these are the only vertices in $\mathcal{G}(y_p^\circ), \ldots, \mathcal{G}(y_q^\circ)$ containing x. Our construction procedure removes x from all sink and source vertices in this set of vertices, so that there remain only $\mu(x)$ such that $x \in L_{\mu(x)}$ and $\nu(x)$ such that $x \in R_{\nu(x)}$.

The proof of Property G5 distinguishes the same two cases as the proof of Property G3. 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 $I_{\mathcal{G}(y)}(x) = [c,d']$ as defined by the two vertices $\mu(x)$ and $\nu(x)$ is correct.

If $\mathcal{T} \neq \mathcal{T}^*$, we removed a leaf a from $\mathcal{T}$ to obtain $\mathcal{T}^*$. If the neighbor b of a is a node of $\mathcal{T}^*$, the path-decompositions for edges $e \in \mathcal{T}^*$ are the same as for their counterparts in $\mathcal{T}$. Hence, by leaving the graph associated with every entry $y_h \in \tau(y_e^{(q)})$ unchanged, we maintain Properties G1–G5.

If b is not a node of $\mathcal{T}^*$, we have to merge the 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 similar 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 graphs $\mathcal{G}(y_h)$ for $y_h \in \tau(y_{(\mathsf{b},\mathsf{c})}^{(q)})$ correctly, observe that by induction hypothesis, coloring σ with color $c(\sigma)$ directs edges from σ to τ and colors τ with color $c(\tau)$. This implies by the validity of $\mathcal{G}(y_h)$ that coloring τ with color $\overline{c}(\tau)$ colors σ with color $\overline{c}(\sigma)$ 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 nodes: We discuss the different possible cases. First assume that $\mathcal{T} = \mathcal{T}^*$. In this case, we have to augment a path-decomposition Y_e to a 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. Only, for all sets $Z_e^{*(i)}$ containing the introduced vertex x, we add 1 to list $\tau(y_e^{*(i)})$. This means that we have to introduce vertex x into every path-decomposition Y_h represented by an element $y_h \in \tau(y_e^{*(i)})$. 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. Properties G1–G5 are now 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 handled analogously.)

Write $\tau[y_e] = (\tau(y_e^{(1)}), \ldots, \tau(y_e^{(a-2)}), \tau(y_e^{(a)}, \ldots, \tau(y_e^{(s)}))$. Then every typical list $\tau(y_e^{*(q)})$ except $\tau(y_e^{*(a-1)})$ and $\tau(y_e^{*(a)})$ are derived from the corresponding typical sequence $\tau(y_e^{(q)})$ as in the case $|Z_e| = |Z_e^*|$. Also, we compute the information stored with every element in such a sequence 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 $Z_e^{*(a)}$. Here we computed a split (τ_1, τ_2) of sequence $\tau(y_e^{(a)})$ and set $\tau(y_e^{*(a-1)}) = \tau_1$ and $\tau(y_e^{*(a)}) = 1 + \tau_2$. If the split is of type two, Properties G1–G5 are easily verified for all graphs $\mathcal{G}(y_h)$ because again these graphs are just copies of the graphs in $\tau(y_e^{(a)})$, possibly augmented by 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 created a new one-vertex graph $\mathcal{G}(y_f)$. In the 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 graphs along all other edges as for the case $\mathcal{T} = \mathcal{T}^*$.

Join nodes: Now 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.

First, 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 i.

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 for $\mathcal{G}(y_f^\circ)$. 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 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 induction hypothesis. Independently 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 valid. 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 Properties G3 and G5, we argue as follows. The construction of graphs $\mathcal{G}_a(y_f^\circ)$ and $\mathcal{G}_b(y_f^\circ)$ corresponds to the construction of path-decompositions A_f° and B_f° corresponding to elements a_f° and b_f° in a° and b°. Increasing values ρ_{σ_a} and ρ_{σ_b} for the source vertices σ_a of $\mathcal{G}_a(y_f^\circ)$ and σ_b of $\mathcal{G}_b(y_f^\circ)$ corresponds to the duplication of the initial sets of A_f° and B_f° to construct two path-decompositions $\bar{A}_f^\circ$ and $\bar{B}_f^\circ$ of the same length. We "compute" path-decompositions Y_f° by a parallel composition of graphs $\mathcal{G}_a(y_f^\circ)$ and $\mathcal{G}_b(y_f^\circ)$.

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 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, the only vertices shared between the sets of $\mathcal{G}_a(y_f^\circ)$ and $\mathcal{G}_b(y_f^\circ)$ can be vertices in X_i. Now y_f° corresponds to a set $Z_e^{(q)}$, so that exactly the vertices in $Z_e^{(q)}$ are shared between these two graphs. Moreover, as the 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 $Z_e^{(q)}$ must be contained in every set of $\bar{A}_f^\circ$ and every set of $\bar{B}_f^\circ$. Hence, $Z_e^{(q)} \subseteq L_{\sigma_a}$, $Z_e^{(q)} \subseteq R_{\tau_a}$, $Z_e^{(q)} \subseteq L_{\sigma_b}$, and $Z_e^{(q)} \subseteq R_{\tau_a}$. Now we defined $L_\sigma \leftarrow L_{\sigma_a} \cup L_{\sigma_b}$ and $L_{\sigma_a} \leftarrow L_{\sigma_b} \leftarrow \emptyset$. Thus, every vertex in $L_{\sigma_a} \cup L_{\sigma_b}$ occurs in exactly one set, namely L_σ, and every vertex x in any other set L_α occurs only in this set because if $x \in \mathcal{G}_a(y_f^\circ)$, then no vertex $\beta \in \mathcal{G}_a(y_h^\circ)$ can exist whose set L_β contains vertex x, and no such vertex in $\mathcal{G}_b(y_f^\circ)$ can exist because $x \notin Z_e^{(q)}$. For sets R_α we argue analogously. Hence, Property G4 holds for every graph $\mathcal{G}(y_f^\circ)$. □

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

Lemma 20 *Graph $\mathcal{G}$ has size $O(N)$.*

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 $O(N)$ vertices. Also, it is easy to see that the in-degree and out-degree of every vertex are bounded by 2. Hence, $\mathcal{G}$ has $O(N)$ edges. □

Phase 4 — Constructing the tree-decomposition explicitly In this section, we compute for every graph $\mathcal{G}(e)$ computed in the previous section, the path-decomposition Y_e represented by $\mathcal{G}(e)$, and we link all these path-decompositions to obtain the desired tree-decomposition $\mathcal{E}$. We first describe how to construct all path-decompositions Y_e.

We start by untangling $\mathcal{G}$. In particular, we pick a vertex α_e and a color c_e for every subgraph $\mathcal{G}(e)$ of $\mathcal{G}$ and then apply the approach of Section 4 to $\mathcal{G}$. The result is a DAG $\mathcal{G}'$ whose connected components $\mathcal{G}'(e)$ represent the path-decompositions Y_e. Given the DAG $\mathcal{G}'$, we topologically sort it in $O(sort(N))$ I/Os [12].

First 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 [12]. Next we have to 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, storing the at most two triples $(e,x,I(\alpha))$ and $(e,x,I(\beta))$ with e and x as their first two components consecutively. $I(x)$ is now the smallest interval containing $I(\alpha)$ and $I(\beta)$. If there is only one such interval $I(\alpha)$, then $x \in L_\alpha \cap R_\alpha$, i.e., appears only at 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.

Now we write for every vertex x represented in L, triples (e,a,x) to a list $\mathcal{S}$, where $a \in I(x)$. We sort the list $\mathcal{S}$ by the first two components of its entries, storing all entries with the same component (e,a) consecutively. $(e,1),\ldots,(e,|Y_e|)$ are the nodes of the path-decomposition Y_e. Now we scan $\mathcal{S}$ 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. Now observe that we can perform the above steps for all graphs $\mathcal{G}(e)$ at the same time, as we label every record in L and $\mathcal{S}$ concerning tree-decomposition Y_e with the ID of edge e.

It remains to describe how to link path-decompositions Y_e into 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 appropriate edges between the 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. Now we sort LT by the first components of its entries and $\mathcal{L}$ by first components. In a single scan we can translate entries (α,β) in $\mathcal{L}$ into entries $((e_1,a),\beta)$. We sort $\mathcal{L}$ by the second components and scan again to translate entries $((e_1,a),\beta)$ into entries $((e_1,a),(e_2,b))$ and add these entries to the edge set of U.

We summarize this section on the following theorem.

Theorem 4 *Given a graph $G = (V,E)$, two constants k,ℓ, and a tree-decomposition $\mathcal{D} = (\mathcal{X},T)$ of width k and size $O(N)$ for G, it takes $O(sort(N))$ I/Os and $O(N/B)$ blocks of external memory to decide whether G has tree-width at most ℓ and, if so, compute a tree-decomposition $\mathcal{E} = (\mathcal{Y},U)$ of width at most ℓ and size $O(N)$ for G.*

Proof. As graph $\mathcal{G}$ consists of a collection of series parallel graphs, $\mathcal{G}$ is planar. As the class of planar graphs is sparse and closed under edge contraction, Lemma 8 can be applied to $\mathcal{G}$. Hence, the I/O-bound of the theorem follows if we can prove that the graph $\mathcal{G}$ as well as the constructed tree-decomposition have size $O(N)$. Lemmas 18 and 20 do this. The correctness of the algorithm is established if the correctness of each of its phases can be shown. The correctness of Phase 1 is shown in [11]. Phase 2 is trivial. Lemma 19 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 is obvious. □

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 the path-decompositions corresponding to trunk-edges.

5.3.3 Constructing a Nice Tree-Decomposition

In this section we consider the problem of constructing a nice tree-decomposition $\mathcal{E} = (\mathcal{Y},U)$ of a graph $G = (V,E)$, given a tree-decomposition $\mathcal{D} = (\mathcal{X},T)$ of G. In [21], it is shown how to compute a triangulation $G' = (V,E')$ with the same tree-decomposition as G. In Section 5.3.4, we show how to compute a PEO for G' in $O(sort(N))$ I/Os. We briefly sketch the algorithm by Bodlaender and Kloks [11] for constructing the nice tree-decomposition in linear time. Afterwards we show how to realize this algorithm in $O(sort(N))$ I/Os in external memory.

The algorithm by Bodlaender and Kloks starts by creating a node in the tree-decomposition containing the last $k+1$ vertices of the PEO. After this, the vertices of G' are inserted one by one. Let v be the next vertex to be inserted. As v is simplicial for the subgraph of G' induced by all vertices following and including v in the PEO, the neighbors of v form a clique C of size k in this graph and must be stored at some node in the current tree-decomposition. Moreover, there exists such a node i that is either a leaf or has only one child.

If i is a leaf, we add one or two vertices below i. If $X_\mathsf{i} = C$, we add a child j of i with $X_\mathsf{j} = C \cup \{v\}$. Otherwise, add two nodes j and k with j being a child of i and k being a child of j. Let $X_\mathsf{j} = C$ and $X_\mathsf{k} = C \cup \{v\}$.

If i has one child j, insert a node k between i and j with $X_\mathsf{k} = X_\mathsf{i}$ and add another child l of i with $X_\mathsf{l} = X_\mathsf{i}$. Node l is a leaf, and we can insert v at node l as described in the previous paragraph.

Our external memory algorithm has to deal with two difficulties. First, we have to show how to find the node i 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 observation tells us how to find the node i where to insert the next vertex v. Let w be the last vertex in v's neighborhood that has been inserted before v. As the neighbors of v form a clique, they must all be neighbors of w. Hence, the insertion of v can be performed at the leaf produced by the insertion of w. Note that there can be more than one vertex v that has w as the last vertex in its neighborhood. This creates some difficulties: When node w was inserted, node i was a leaf. The insertion of the first such vertex v creates a child of i. The insertion of the next node v' makes i a join node, and the next node v'' should be inserted at one of the children of i instead of i. We describe our solution to this problem next.

Given the PEO, it takes sorting and scanning to find for every vertex v the last vertex w in v's neighborhood inserted before v. Call this vertex $c(v)$. We link all vertices v with the same vertex $c(v)$ into a list $I(c(v))$, sorted by decreasing position in the PEO. If such a list contains some of the last $k+1$ vertices in the PEO, we delete all such vertices from the head of the list. Every vertex w stores the name of the head of its corresponding list $I(w)$ of vertices v with $c(v) = w$.

Now we use time-forward processing to construct the nice tree-decomposition. We start by creating a single node i containing the last $k+1$ vertices in the PEO. For every vertex w stored at i, we pass the ID of i together with the list of these $k+1$ vertices to the head of the list $I(w)$. Now we process vertices one by one in reverse PEO. By Lemma 21, every vertex v receives exactly one node ID i together with the $k+1$ vertices

in X_i from either its node $w = c(v)$ or from its predecessor in $I(w)$. By the same lemma, node i has at most one child, so that we can start inserting v at this node.

When inserting vertex v at node i, we create a new node j with parent i and set $X_\mathsf{j} = X_\mathsf{i}$. Let C be the neighborhood of v. We check whether $X_\mathsf{j} = C$. If so, we create a new node k with parent j and set $X_\mathsf{k} = C \cup \{v\}$. Otherwise, $X_\mathsf{j} = C \cup \{z\}$. We create a node j' with parent j and $X_{\mathsf{j}'} = C$ and another node k with parent j' and $X_\mathsf{k} = C \cup \{v\}$. Now, if v has a successor in the list $I(w)$, where $w = c(v)$, we send X_j as well as the node ID of node j to this successor. If the list $I(v)$ is non-empty, we send the node ID of node k together with the set X_k to the head of this list.

Lemma 21 *Every vertex v receives exactly one node ID i in the above procedure. At the time when v is being inserted, node i has at most one child.*

Proof. Let us first prove that every vertex receives exactly one node ID. If v is the head of the list $I(c(v))$, then v receives a node ID from $c(v)$ but cannot receive any node ID from any other node as there has not been any other node v' with $c(v) = c(v')$ yet. If v is not the head of the list, it receives a node ID from its predecessor in $I(c(v))$; it cannot receive any other node ID, as no node v' with $c(v') \neq c(v)$ sends any node ID to v, no node v' with $v' \neq c(v)$ sends any node ID to v, and node $c(v)$ itself sends a node ID only to the head of the list $I(c(v))$ and not to v.

To prove that at the time when v is being inserted, node i has at most one child, we distinguish two cases. When a vertex w is being inserted into the tree-decomposition, the new node containing w and being sent to the first node in $I(w)$ is a leaf. Hence, the claim is true for the head of $I(w)$. When inserting a vertex $v \in I(w)$, we create a new child j of the node i where the previous node $v' \in I(w)$ was inserted. This may increase the degree of i to 2. However, node j has degree one, and the next node in $I(w)$ is being inserted at j. □

Note that whenever a node becomes a join node, our algorithm does not immediately create the node between the join node and its original child. This can now easily be fixed, as every join node i has two children j and k; for one of the two children, say j, we have $X_\mathsf{j} \neq X_\mathsf{i}$, while for the other one $X_\mathsf{k} = X_\mathsf{i}$. Hence, we know that we have to introduce another node j' with $X_{\mathsf{j}'} = X_\mathsf{i}$ between i and j. Also, we always duplicate a node i when inserting a new vertex at node i. This may produce chains of vertices with only one child each such that the assigned set X_i is the same for all these vertices. These chains can now easily be contracted using standard external memory graph techniques.

Lemma 22 *Given a graph $G = (V,E)$ with tree-decomposition $\mathcal{D} = (\mathcal{X},T)$ of width k and size $O(N)$, a nice tree-decomposition $\mathcal{E} = (\mathcal{Y},U)$ of width at most k and size $O(N)$ for G can be computed in $O(sort(N))$ I/Os using $O(N/B)$ blocks of external memory.*

Proof. Given a tree-decomposition $D = (\mathcal{X},T)$ of G, a triangulation $G' = (V,E')$ of G induced by D can be computed in $O(sort(N))$ I/Os: for every node $\mathsf{i} \in T$ and every pair of vertices $\{v,w\} \in X_\mathsf{i}$, add edge $\{v,w\}$ to E'. In [21] it is proved that G' is a triangulation of treewidth k. By Lemma 23, it takes $O(sort(N))$ I/Os to compute a PEO of G'. Given the PEO, we apply the algorithm described in this section to compute the nice tree-decomposition. The preprocessing and postprocessing steps require sorting and scanning, hence $O(sort(N))$ I/Os. The time-forward processing steps take $O(sort(N))$ I/Os, as every vertex receives only a constant amount of data. The correctness of the algorithm follows from Lemma 21. □

5.3.4 Finding a Perfect Elimination Ordering

In this section, we consider the following problem: Given an undirected graph $G = (V,E)$ with tree-decomposition $\mathcal{D} = (\mathcal{X},T)$ of width k, assume that this graph has been augmented to a triangulation

$G' = (V, E')$ by adding all possible edges between vertices that are stored at the same node of T. 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 to 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 23 *Given a graph $G = (V, E)$, a triangulation $G' = (V, E')$, and a tree-decomposition as defined above, it takes $O(sort(N))$ I/Os to compute a PEO of G'.*

Proof. Clearly, we obtain an ordering of the vertices in $O(sort(N))$ I/Os. We have to prove that the obtained 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'. We show that edge $\{v, w\}$ must also be in G', so that the computed ordering is indeed a PEO. Let i, j, and k be the first nodes in the preorder traversal 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 were not an ancestor of i, the set X_{a}, where a is the LCA of i and k would form a separator separating u from w. For the same reason j must be an ancestor of i. However, no ancestor of i contains u. Hence, as there are edges $\{u, v\}$ and $\{u, w\}$, $\{v, w\} \subseteq X_{\mathsf{i}}$, and edge $\{v, w\}$ is in G'. □

6 Applications

In this section, we give two applications of our tree-decomposition algorithm. In Section 6.1, we show how to solve the single source shortest path problem for graphs of bounded treewidth in $O(sort(N))$ I/Os. In Section 6.2, we describe an approach for solving NP-hard problems on graphs of bounded treewidth in $O(sort(N))$ I/Os and apply it to the following problems: maximum independent set, k-coloring, minimum vertex cover, minimum dominating set, Hamiltonian cycle, and maximum matching.

6.1 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 $k+1$ 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 a kind of dynamic programming in T. In a first phase of the algorithm we process the nodes of T bottom-up, computing for every pair of vertices $\{v, w\} \subseteq X_{\mathsf{i}}$ the distance $d_{\mathsf{i}}(v, w)$ between v and w 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 distance $d(v, w)$ between v and w in G. In a third phase, we use 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 we compute $(k+1) \times (k+1)$-matrices M'_{i} for all nodes $\mathsf{i} \in I$ and store them in preorder. Let the vertices in X_{i} be $v_1, \ldots, v_{k+1}$. Then position (j, k) of matrix M'_{i} stores $d_{\mathsf{i}}(v_j, v_k)$. In the second phase we scan the list of matrices M'_{i} and transform each of them into a matrix M_{i} storing distance $d(v_j, v_k)$ at position (j, k). As each matrix M'_{i} or M_{i} has constant size and the vertex and edge sets of each vertex have constant size as well, we can implement both phases in $O(scan(N))$ I/Os if we can show the following: (1) In the first phase we can compute M'_{i} from the matrices stored at the children of i in T; (2) in the second phase we can compute M_{j} from M'_{j} and M_{i}, where i is the parent of j in T.

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 obtaim 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 $\langle v = u_0, u_1, \ldots, u_s = w\rangle$ in G_i. If x is not contained in this path, then the shortest path between v and w in X_j is the same, and we can 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 $\langle u_0, \ldots, u_{l-1}\rangle$ and $\langle u_{l+1}, \ldots, u_s\rangle$ exist in G_j and must be the shortest paths between u_0 and u_{l-1} and u_{l+1} and 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 non-infinity 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 $\hat{G}_\mathsf{i}$. We can now compute M'_i by running Dijkstra's algorithm [14] on $\hat{G}_\mathsf{i}$ in internal memory, as $|\hat{G}_\mathsf{i}| = O(1)$.

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 Dijkstra's algorithm on $\hat{G}_\mathsf{i}$. We have to prove that this gives the right result.

Consider a shortest path π in G_i between two vertices $v, w \in X_\mathsf{i}$. We can cut π into maximal subpaths $\pi_1, \ldots, \pi_q$ such that none of these paths has an interior vertex in X_i. Such a path π_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 $G_\mathsf{j} - X_\mathsf{i}$; the other contains all vertices in $G_\mathsf{k} - X_\mathsf{i}$. Hence, if u and z are the endpoints of π_k with $\{u, z\} \subseteq X_\mathsf{i}$, then π_k must be the shortest path between u and z in either G_l or G_m, so that we have assigned the length of π_k as the weight of edge (u, z) in $\hat{G}_\mathsf{i}$. As we do this for all subpaths π_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 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 the inductive assumption, 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 the shortest path π from x to v. Let w be the first vertex in X_i suceeding x on path π and consider the subpaths π_1 from x to w and π_2 from w to v. Both paths have to be shortest paths as well. Hence, the length of π_2 is stored in M_i. We claim that π_1 stays within G_j, which implies that the length of π_1 is just the distance between x and w stored in M'_j. Again, this claim is trivial to prove as the vertices in X_i form a separator of the vertices in G_j from the rest of G, so that in order to reach a vertex not in G_j, the path starting at x must cross some vertex in X_i; but w is the first such vertex, so that the path from x to w cannot contain any vertex not in G_j.

Hence, we build a graph $\tilde{G}_\mathsf{j}$ with vertex set X_j. An edge (v, w) in G_j has the weight given in M_i if $x \notin \{v, w\}$. Otherwise, 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 using Dijkstra's algorithm.

Phase 3 In this phase, 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 $v \in X_j \setminus X_i$. Then any path from s to v must contain at least one vertex in $X_i \cap X_j$. Hence, $d(s,v) = \min\{d(s,w) + d(w,v) : w \in X_i \cap X_j\}$. Distances $d(s,w)$ are already provided by the vector Δ_i, as $X_i \cap X_j \subseteq X_i$. Distances $d(w,v)$ are provided by the matrix M_j. Hence, this computation can be carried out for all trees T_l using time-forward processing again.

Finally, if we are interested not only in the distances but also in constructing the shortest path tree (SPT) for s, we can construct this SPT as follows: We process T top down. For every two vertices v and w stored at a node i such that there is an edge $(v,w) \in E$ and such that $d(s,v) + d(v,w) = d(s,w)$, we add the edge (v,w) to the SPT. If there is more than one such edge for vertex w, we pick one arbitrarily. Finally, in order to pick only one parent per vertex w in the SPT, every node i pushes a Boolean array to its children storing which nodes have already been assigned a parent in the SPT. We can summarize this section in the following lemma.

Theorem 5 *Given a nice tree-decomposition $\mathcal{D} = (\mathcal{X}, T)$ of width at most k and size $O(N)$ for a directed graph $G = (V,E)$, it takes $O(scan(N))$ I/Os to compute a shortest path tree for any source vertex s in G, provided that the nodes of T are stored in preorder.*

Together with our algorithm for computing a nice tree-decomposition and for computing a preorder numbering a tree, we obtain the following corollary.

Corollary 2 *Given a directed graph $G = (V,E)$ with treewidth at most k, it takes $O(sort(N))$ I/Os to compute a shortest path tree for any source vertex s in G.*

6.2 Solving NP-hard Problems on Graphs of Bounded Treewidth

Arnborg, Lagergren, and Seese [6] present polynomial time solutions for NP-hard problems on bounded treewidth graphs which are describable in monadic second order logic. Arnborg and Proskurowski [4] 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 [4], 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 [4, 7] together with the techniques in Section 3 give the following results.

Theorem 6 *Given a graph $G = (V,E)$ with treewidth bounded by some constant k, the following optimization problems can be solved in $O(sort(N))$ I/Os or in $O(scan(N))$ I/Os if 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, dominating set, and Hamiltonian cycle.*

Proof. Simulate the algorithms from [4] using Lemma 7. If the tree-decomposition is not part of the input, a nice tree-decomposition can be computed in $O(sort(N))$ I/Os by Theorem 2 and Lemma 22. Using results from [12], it takes $O(sort(N))$ I/Os to arrange the nice tree-decomposition in preorder. □

Theorem 7 *Given a graph $G = (V,E)$ with treewidth bounded by some constant k, the following decision problems can be solved in $O(sort(N))$ I/Os or in $O(scan(N))$ I/Os if 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). □

We illustrate the approach for Theorem 6 using the vertex cover problem as an example. The other solutions are similar. Details are provided in [4]. Also, the solutions in [4] 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.

Our 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 we compute a *candidate set* $\mathcal{C}_i$ for every node i. This set $\mathcal{C}_i$ contains all pairs (S, ω), where $S \subseteq X_i$ and there is a vertex cover C of G_i containing all vertices in S, but no vertex in $X_i \setminus S$. 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 $(S, \omega) \in \mathcal{C}_r$, 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 $\mathcal{C}_i$ is easy to compute for a start node because $Cov(S) = \emptyset$ if S is not a vertex cover of $G[X_i]$, and $Cov(S) = \{S\}$ if S is a vertex cover of $G[X_i]$. In the latter case, we put $(S, \omega(S))$ into $\mathcal{C}_i$.

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 $C \cap X_i = S$ if and only if $C \cap X_j = S$ or $C \cap X_j = S \cup \{x\}$. Hence, we put a pair (S, ω) into $\mathcal{C}_i$ if at least one of (S, ω_1) and $(S \cup \{x\}, \omega_2)$ is in $\mathcal{C}_j$. 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 $N_{G_i}(x) \subseteq C$. As $N_{G_i}(x) \subseteq X_i$, $N_{G_i}(x) \subseteq S$ in the latter case. Hence, we construct $\mathcal{C}_i = \mathcal{C}' \cup \mathcal{C}''$ from two sets $\mathcal{C}'$ and $\mathcal{C}''$: $\mathcal{C}' = \{(S \cup \{x\}, \omega + \omega(x)) : (S, \omega) \in \mathcal{C}_j\}$ and $\mathcal{C}'' = \{(S, \omega) : (S, \omega) \in \mathcal{C}_j \wedge N_{G_i}(x) \subseteq S\}$.

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 $G_j = (V_j, E_j)$ and $G_k = (V_k, E_k)$. Let $C_1 = C \cap V_j$, $C_2 = C \cap V_k$, and $C \cap X_i = S$. Then $C_1 \cap X_j = S$ and $C_2 \cap X_k = S$. Hence, there is a vertex cover C with $C \cap X_i = S$ if and only if $(S, \omega_1) \in \mathcal{C}_j$ and $(S, \omega_2) \in \mathcal{C}_k$, 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 $\mathcal{C}_i$.

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 $(S, \omega) \in \mathcal{C}_r$ and reporting the weight $\omega_{\min} = \min\{\omega : (S, \omega) \in \mathcal{C}_r\}$. We construct a vertex cover C with weight $\omega(C) = \omega_{\min}$ as follows:

At the root r of T, we mark the element $(S_{\min}, \omega_{\min}) \in \mathcal{C}_r$ 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 $(S, \omega') \in \mathcal{C}_j$ 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 $(S \setminus \{x\}, \omega - \omega(x)) \in \mathcal{C}_j$ as selected. Otherwise, we mark the pair $(S, \omega) \in \mathcal{C}_j$ 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 $(S,\omega_1)\notin\mathcal{C}_j$, 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 $\mathcal{C}_i$ and we guarantee that $C\cap X_i = S$. Hence, C is a vertex cover for all graphs $G[X_i]$, where i is a leaf of T, and an inductive argument shows that C is a vertex cover for $G_i=(V_i,E_i)$. A similar inductive argument shows that if (S,ω) is the selected pair in $\mathcal{C}_i$, then $\omega(C\cap V_i)=\omega$, so that $\omega(C)=\omega_{\min}$.

7 Future Work

In order to obtain improved solutions for the above applications, it is important to find improved algorithms to construct tree-decompositions for bounded treewidth graphs. One approach we are pursuing is to exploit the additional structure exhibited by special types of bounded treewidth graphs. One such class of treewidth $2k$ is the class of k-outerplanar graphs, where a k-outerplanar embedding is likely to give us all information needed to construct tree-decompositions for these graphs.

Also, we are interested in examining the reduction techniques introduced in [5] to see whether these techniques can be used to design improved external memory algorithms for computing tree-decompositions for graphs of treewidth at most 4.

References

[1] James Abello, Adam L. Buchsbaum, and Jeffery Westbrook. A functional approach to external graph algorithms. In *Proceedings of the 6th European Symposium on Algorithms*, pages 332–343, 1998.

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

[3] Lars Arge, Gerth Stølting Brodal, and Laura Toma. On external memory MST, SSSP, and multi-way planar separators. In *Proceedings of SWAT'2000*, 2000.

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

[5] Stefan Arnborg, Bruno Courcelle, Andrzej Proskurowski, and Detlef Seese. An algebraic theory of graph reduction. *Journal of the ACM*, 40(5):1134–1164, November 1993.

[6] Stefan Arnborg, Jens Lagergren, and Detlef 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.

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

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

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

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

[12] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, January 1995.

[13] Franke Dehne, Wolfgang Dittrich, and David 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.

[14] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerical Mathematics*, 1:269–271, 1959.

[15] Babette de Fluiter and Hans L. Bodlaender. Parallel algorithms for treewidth two. In *Proceedings of the 23rd International Workshop on Graph-Theoretic Concepts in Computer Science (WG'97)*, volume 1335 of *Lecture Notes in Computer Science*, pages 157–170. Springer-Verlag, 1997.

[16] Greg N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM Journal on Computing*, 16(6):1004–1022, December 1987.

[17] Greg N. Frederickson. Planar graph decomposition and all pairs shortest paths. *Journal of the ACM*, 38(1):162–204, January 1991.

[18] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34:596–615, 1987.

[19] Philip Klein, Satish Rao, Monika Rauch, and Sairam Subramanian. Faster shortest-path algorithms for planar graphs. In *Proceedings of the 26th ACM Symposium on Theory of Computing*, pages 27–37, May 1994.

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

[21] Ton Kloks. *Treewidth: Computations and Approximations*, volume 842 of *Lecture Notes in Computer Science*. Springer Verlag, June 1994.

[22] Vijay Kumar and Eric 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 Computing*, October 1996.

[23] J. Lagergren. Efficient parallel algorithms for tree-decomposition and related problems. In *Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 173–182, 1990.

[24] 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.

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

[26] Kameshwar Munagala and Abhiram Ranade. I/O-complexity of graph algorithms. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, January 1999.

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

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

[29] 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.

[30] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, 1994.

[31] Robert E. Tarjan and Mihalis 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.

[32] Jeffrey Scott Vitter. *External memory algorithms and data structures*. American Mathematical Society, 1999.

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

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