Electronic Colloquium on Computational Complexity, Report No. 126 (2012)

Computing Bounded Path Decompositions in Logspace

Shiva Kintali* Sinziana Munteanu †‡

Abstract

We present a logspace algorithm to compute path decompositions of bounded pathwidth graphs, thus settling its complexity. Prior to our work, the best known upper bound to compute such decompositions was linear time [Bod96, BK96]. We also show that deciding if the pathwidth of a graph is at most a given constant is **L**-complete. Besides being of fundamental interest, our results represent an important step to gain a better understanding of the complexity of Graph Isomorphism of bounded pathwidth graphs.

1 Introduction

Wagner [Wag37] introduced simplicial tree decompositions. The notions of treewidth and tree decomposition were introduced (under different names) by Halin [Hal76]. Robertson and Seymour [RS86] reintroduced these concepts in their seminal work on graph minors. Roughly speaking, the treewidth of an undirected graph measures how close the graph is to being a tree. Let $G(V, E)$ denote a simple undirected graph with n vertices *labeled* arbitrarily from 1 to n. For a subset $V' \subseteq V$, $G[V']$ denotes the subgraph of G induced by the vertices of V'.

Definition 1.1. *(Tree decomposition, Treewidth)* A *tree decomposition* of a graph $G(V, E)$ is a pair $\mathcal{D} = (\{X_i \mid i \in I\}, T(I, F))$ where $\{X_i \mid i \in I\}$ is a collection of subsets of V (called bags) and $T(I, F)$ is a tree such that

- $\bigcup_{i \in I} X_i = V$. (TD-1)
- for all edges $(u, v) \in E$, there is an $i \in I$ with $u, v \in X_i$. (TD-2)
- for all $i, j, k \in I$, if j is on the path from i to k in T, then $X_i \cap X_k \subseteq X_j$. (TD-3)

The *width* of a tree decomposition $\mathcal{D}$ is $\max_{i \in I} |X_i| - 1$. The *treewidth* of a graph G, denoted by $tw(G)$, is the minimum width over all tree decompositions of G. The elements of I are called the *nodes* of the tree T.

Besides playing a crucial role in structural graph theory, treewidth also proved to be very useful in algorithmic graph theory. Several problems that are NP-hard on general graphs are solvable in polynomial time (some even in linear time) on graphs with bounded treewidth (also known

*Department of Computer Science, Princeton University, Princeton, NJ 08540. Email : *kintali@cs.princeton.edu*
†Computer Science Department, Carnegie Mellon University, Pittsburgh PA 15213. Email : *smuntean@cs.cmu.edu*
‡This work was done while the author was at Princeton University.

ISSN 1433-8092

as partial k-trees). These problems include hamiltonian cycle, graph coloring and vertex cover. Courcelle's theorem [Cou90] states that for every monadic second-order (MSO) formula ϕ and for every constant k there is a *linear time* algorithm that decides whether a given logical structure of treewidth at most k satisfies ϕ.

Recently, Elberfeld, Jakoby and Tantau [EJT10] proved a *logspace* version of Courcelle's theorem, thus showing that several MSO properties of partial k-trees can be decided in logspace. They also presented a logspace algorithm to compute tree decompositions of bounded treewidth graphs. Unfortunately their techniques are not applicable to compute path decompositions of bounded pathwidth graphs. Bounded pathwidth graphs are a natural subset of bounded treewidth graphs. Pathwidth, defined as follows, measures how close a graph is to being a path.

Definition 1.2. *(Path decomposition, Pathwidth)* A *path decomposition* of a graph $G(V, E)$ is a sequence $\mathcal{P} = (Y_1, Y_2, \ldots, Y_s)$ of subsets of V, such that

- $\bigcup_{1 \leq i \leq s} Y_i = V$ (PD-1)
- for all edges $(u, v) \in E$, there is an $i, 1 \leq i \leq s$, with $u, v \in Y_i$ (PD-2)
- for all i, j, k with $1 \leq i < j < k \leq s$: $Y_i \cap Y_k \subseteq Y_j$ (PD-3)

The *width* of a path decomposition $\mathcal{P}$ is $\max_{i \in I} |Y_i| - 1$. The *pathwidth* of a graph G, denoted by $pw(G)$, is the minimum width over all path decompositions of G.

In this paper, we present a logspace algorithm to compute path decompositions of bounded pathwidth graphs. Our techniques are completely different from those of [EJT10]. However, in our main algorithm we first use the algorithm of [EJT10] to compute a tree decomposition with an underlying tree of height $O(\log n)$. We then show how to convert this tree decomposition into a path decomposition. Since the input graph G has bounded pathwidth (say k), its treewidth is at most k. More generally, our algorithm can start with any graph of treewidth at most l and decide whether the pathwidth of G is at most k, and if so, compute a path decomposition of width at most k. We also show that deciding if the pathwidth of a graph is at most a given constant is **L**-complete. Our main theorem is as follows:

Theorem 1.3. For all constants $k, l \geq 1$, there exists a *logspace* algorithm that, when given a graph G of treewidth $\leq l$, decides whether the pathwidth of G is at most k, and if so, finds a path decomposition of G of width $\leq k$ in *logspace*.

Prior to our work, the best known upper bound to compute such decompositions was linear time [Bod96, BK96]. Bodlaender and Kloks [BK96] presented a polynomial time algorithm for computing path decompositions of *bounded treewidth* graphs. This result played a crucial part in a later paper of Bodlaender [Bod96] giving a linear time algorithms to compute tree decompositions and path decompositions of bounded pathwidth graphs. This in-turn played a crucial role in Courcelle's Theorem.

Since bounded treewidth graphs can have pathwidth as large as $\Omega(\log n)$, our techniques cannot be directly applied to achieve a logspace algorithm to compute path decompositions of *bounded treewidth* graphs. This is one of the main open problems arising from our work.

One of the main motivations behind our work is to gain better understanding of the complexity of Graph Isomorphism of bounded pathwidth graphs. Due to lack of space, we present the details of this motivation in the **appendix**. Of course, computing bounded path decompositions in logspace is of independent interest too. We believe that the techniques in our paper, combined with those of [EJT10], can be applied to settle the complexity of computing other graph width parameters.

2 Overview of our algorithm

Our algorithm is based on Bodlaender-Kloks' polynomial time algorithm for computing path decompositions of bounded treewidth graphs. We first give an overview of their algorithm.

Assume we are given a tree decomposition of the input graph G. Let the underlying tree of this decomposition be T. Every node of T has a *bag* containing at most constant number of vertices of G. We convert this tree decomposition into a *nice tree decomposition*. We root the tree T at an arbitrary node of T. Define G_p to be the *subgraph of G rooted at p*, the subgraph of G induced by the vertices in the nodes of the subtree of T rooted at p.

We perform a bottom-up traversal of T. If the current node p is a leaf, then G_p has a constant number of vertices and hence its path decompositions can be found in constant time. Otherwise, we extend the path decompositions of the subgraphs of G rooted at the children of the current node p, to a path decomposition of the subgraph of G rooted at p. This is done by a brute-force search over all the "local" separators of the graph G. The information about these separators is available in the bags of p and the children of p.

We now explain how this search can be done in polynomial time. By property PD-3, the vertices in G_p that are not in X_p do not belong to the nodes of T that remain to be explored, i.e. the only nodes they belong to are nodes of the subtree of T rooted at p. So the number of such vertices in any bag of a path decomposition of G_p offers sufficient "local" information and the labels of these vertices can be disregarded. This motivates the use of the following data structures. We define the *interval model* of a path decomposition Y to be the path decomposition of the subgraph of G induced by X_p, obtained by removing the consecutive duplicate bags of a path decomposition Y', where Y' is obtained by intersecting every bag of Y with X_p. The path decomposition Y is split into a chain of subpaths, so that all the bags in the same subpath have the same intersection with X_p. Taking the cardinality of each of these bags, we obtain the *list* of Y, a sequence of integer sequences. To reduce the size of these lists, we define the *typical list* of a list y, obtained by repeatedly eliminating all the integers, except for the first and last ones, of every nondecreasing subsequence of y. Hence, we associate to each path decomposition, a *characterisctic*, a pair containing its interval model and its typical list.

We then consider all the path decompositions of G_p and associate to p all their characteristics. To reduce the number of characteristics at each node, we define an ordering on the path decompositions and eliminate all the characteristics of path decompositions for which there exists a smaller path decomposition. We define the resulting set of characteristics to be $FS(p)$, the *full set of characteristics* at p. Another way to view the full set of characteristics is as a set X of fingerprints of path decompositions such that whenever there exists a path decomposition, there exists a (possibly different) path decomposition with a fingerprint in X.

We now present **a high-level description of our algorithm**. We first use an algorithm of [EJT10] to compute a tree decomposition of the input graph. The underlying tree (say T) of this tree decomposition has height $O(\log n)$. This plays a very crucial role in several subroutines of our algorithm. We then perform a logspace bottom-up traversal of T, using an algorithm of Lindell [Lin92]. We give a brief description of Lindell's algorithm in the **appendix**. If the current node p is a leaf, then G_p has a constant number of vertices and hence its path decompositions can be found in constant space. Otherwise, we show that the above polynomial-time algorithms (computing the full set of characteristics at the current node p from the full set of characteristics at the children of p) can be implemented in logspace. Observe that when p has two children, say q and r, both $FS(q)$ and $FS(r)$ are needed to compute $FS(p)$ and that we need to retain $FS(q)$ when performing the

traversal of the subtree of T rooted at r. Since the depth of T is $O(\log n)$, we design a technique to reduce the space required to store $FS(q)$ from logspace to constant space. We represent all the vertices in a bag of an interval model by their rank in an ordering of the vertices of X_p. In the course of the traversal, we store the relabelings of the full set of characteristics of all smaller children of the explored vertices for which we have not already computed their full set of characteristics. We recompute the original full set of characteristics at each node when needed, i.e. when computing the full set of characteristics at its parent.

We associate to each node p, a unique path decomposition, which we call the *characteristic path decomposition*. We now perform another bottom-up traversal of T and compute for each current node p, the *gap list* $g(p)$ at p, the list of the number of bags between consecutive integers in the typical list of the characteristic path decomposition. However, performing this traversal would give to an $O(\log^2 n)$ algorithm, since $g(p)$ requires logspace and the depth of T is $O(\log n)$. Note that when p has two children, say q and r, we need to retain $g(q)$ when traversing the subtree of T rooted at r. To obtain a logspace algorithm, we observe that the gap list at the root of T is the sum of lists, each computable only from the nodes on a root-to-leaf path of T. This observation plays a crucial role to achieve our intended logspace upper bound.

By property PD-3 of a path decomposition, all the bags containing a vertex form a connected subpath and hence to specify all the bags containing a vertex, it is enough to specify the first and last bag containing it. Therefore, a path decomposition is completely determined by specifying for each vertex, the first and last bag containing it. We define the *left endpoint* of a vertex at a node to be the index of the first bag containing it in the characteristic path decomposition at that node and we similarly define the *right endpoint* to be the index of the last bag containing it. We iterate the vertices of G and for each current vertex v, we compute its left and right endpoints at the root. By symmetry, it is enough to show that the left endpoint can be computed in logspace. The algorithm is similar to the algorithm for computing the gap list, except that the left endpoint of v at the root is the *minimum* of the left endpoints of v in the path decompositions associated to every root-to-leaf path of T.

We now give a more **detailed description of our algorithm**, in which we include references to our definitions and theorems. We first use an algorithm of Elberfeld, Jakoby and Tantau [EJT10] to compute a tree decomposition $\mathcal{D}'$ of G in logspace.

Theorem 2.1. (Elberfeld, Jakoby and Tantau [EJT10]) For every constant $l \geq 1$, there is a logspace algorithm that given graph G determines whether $tw(G) \leq l$, if so, outputs a width-l rooted tree decomposition $\mathcal{D}' = (\{X_i \mid i \in I'\}, T'(I', F'))$ of G. Moreover, T' is a *binary tree* of height $O(\log n)$.

In Section 4, we define a *nice tree decomposition* (see Definition 4.1) and then give a logspace algorithm to transform $\mathcal{D}'$ into a nice tree decomposition $\mathcal{D} = (\{X_i \mid i \in I\}, T(I, F))$ (Theorem 4.2). We define each node of $\mathcal{D}$ to be one of the following types: *start*, *forget*, *introduce* or *join* node (see Definition 4.3). Let *root* be the root node of $\mathcal{D}$, $v \in V(G)$ and $p \in I$. By the definition of a nice tree decomposition, every node of $\mathcal{D}$ has at most two children. If p has one child, we denote its child by q. If p has two children, we denote its children by q and r. We distinguish between q and r by the lexicographic ordering of their labels.

In Section 5, we give a logspace algorithm to compute $FS(root)$, the full set of characteristics at the root node of T. Since $FS(root) \neq \emptyset$ if and only if $pw(G) \leq k$, once we have shown that $FS(root)$ can be computed in logspace, we have solved the *decision version* of the problem. The

results in Section 5.3 - Section 7.2 contribute to the "constructive" part of the solution, in which we show that a path decomposition can be found in logspace. For each node p, we define the *full set of characteristics* at p, denoted by $FS(p)$ (see Definition 3.11). In Section 5.1, we show how to compute in logspace $FS(p)$, given the full set of characteristics at the children of p. This is done separately based on the type of the node p. We call the procedures for achieving this ComputeStartFS, ComputeForgetFS, ComputeIntroduceFS and ComputeJoinFS. These procedures are the same as the corresponding polynomial-time procedures of [BK96]. We show that they can be implemented in logspace. In Section 5.2, we give a logspace algorithm to compute $FS(root)$ (Theorem 5.15). This uses Lindell's tree traversal algorithm [Lin92], the logspace procedures of the previous section and a logspace relabeling procedure. In Section B, we give an overview of Lindell's tree traversal algorithm. The relabeling procedure reduces the space required to store a bag with constant number of vertices from logarithmic to constant (Lemma 5.13) by changing the label of each vertex to its rank in an ordering of the vertices in the bag (see Definition 5.12). In Section 5.3, we give a logspace algorithm to compute a unique characteristic at each node. For any descendant p' of p and any characteristic $\mathcal{C}_p \in FS(p)$, we define the descendant characteristic $dc(p, p', \mathcal{C}_p)$ of p at p' (see Definition 5.16) and then give a logspace procedure ComputeDescendantCharacteristic for computing it (Theorem 5.17). The unique characteristic $\mathcal{C}_p^*$ will be the descendant characteristic at p of the lexicographically smallest characteristic in $FS(root)$ (see Definition 5.18).

In Section 6, we give a logspace algorithm for computing the *gap list* at the root, an auxiliary data structure useful for computing the path decompositions. We define the characteristic path decomposition at p, denoted by $\mathcal{CP}(p)$ (see Definition 6.1) and we define the gap list at p, denoted by $g(p)$ (see Definition 6.2). In Section 6.1, we see how to compute in logspace $g(p)$, given the gap lists at the children of p. This is done separately depending on the type of the node p. We call the procedures for achieving this ComputeStartGapList, ComputeForgetGapList, ComputeIntroduceGapList and ComputeJoinGapList. In Section 6.2, we give a logspace algorithm that computes $g(root)$ (Theorem 6.8). This follows from the fact that $g(root)$ is the sum of lists, each computable only from the nodes on a path from a leaf to the root (Theorem 6.7).

In Section 7, we give a logspace algorithm for computing the *endpoints* of a vertex at the root. We define $l(p, v)$ and $r(p, v)$ to be the left and right endpoints of v at p (see Definition 7.1). In Section 7.1, we give a logspace algorithm for computing $l(p, v)$, given the left endpoints of v at the children of p. This is done separately depending on the type of the node p. We call the procedures for achieving this ComputeStartLeftEndpoint, ComputeForgetLeftEndpoint, ComputeIntroduceLeftEndpoint and ComputeJoinLeftEndpoint. In Section 7.2, we give a logspace algorithm that computes $l(root, v)$ (Theorem 7.5). This follows from the fact that $l(root, v)$ is the minimum of the left endpoints of v in the path decompositions corresponding just to the nodes on individual paths from a leaf to the root (Theorem 7.4).

To compute a path decomposition of G, start from $\mathcal{D}'$ and apply successively the logspace algorithms of Section 4, Section 5, Section 6 and Section 7. Since the class of logspace computable functions is closed under composition, we thus obtain a logspace algorithm to compute path decompositions of bounded pathwidth graphs. The algorithms of Section 4, Section 5, Section 6 and Section 7 work for any constants l and k, implying our main theorem (Theorem 1.3).

In Section 8, we prove that the language path-width-k $= \{G \mid pw(G) \leq k\}$ is L-complete. This complements a result of Elberfeld, Jakoby and Tantau [EJT10], showing that the language tree-width-k $= \{G \mid tw(G) \leq k\}$ is L-complete.

3 Definitions

3.1 Treewidth and Pathwidth

Definition 3.1. *(Rooted tree decomposition)* A *rooted tree decomposition* is a tree decomposition $\mathcal{D} = (\{X_i \mid i \in I\}, T(I, F))$ where T is a rooted tree.

Definition 3.2. *(Minimal path decomposition)* A path decompostion $\mathcal{P} = (Y_1, Y_2, \ldots Y_s)$ is called *minimal* if for all $i, 1 \leq i < s, Y_i \neq Y_{i+1}$.

Definition 3.3. *(Subgraph rooted at a node)* Let $\mathcal{D} = (\{X_i \mid i \in I\}, T(I, F))$ be a rooted tree decomposition of a graph $G(V, E)$. For each node i of T, let T_i be the subtree of T rooted at node i and let $V_i = \bigcup_{j \in T_i} X_j$. We call $G_i = G[V_i]$ the *subgraph of G rooted at i*.

Definition 3.4. *(Partial path decomposition)* A *partial path decomposition* rooted at node $i \in I$ is a path decomposition of G_i, the subgraph of G rooted at i.

3.2 Interval Model, Typical Lists and Characteristics

Definition 3.5. *(Interval model)* Let $\mathcal{P} = (Y_1, Y_2, \ldots Y_s)$ be a partial path decomposition rooted at node i. We define $(Z_j)_{1 \leq j \leq s} = (Y_1 \cap X_i, Y_2 \cap X_i, \ldots, Y_s \cap X_i)$, a path decomposition of the bag X_i. Let $1 = r_1 < \ldots < r_{t+1} = s + 1$ be defined by

$$\forall_{1 \leq j \leq t} \ \forall_{r_j \leq k < r_{j+1}} \ [Z_k = Z_{r_j}] \bigwedge \forall_{1 \leq j < t} [Z_{r_j} \neq Z_{r_{j+1}}]$$

The *interval model* for $\mathcal{P}$ at i is the path decomposion $\mathcal{Z} = (Z_{r_j})_{1 \leq j \leq t}$.

Definition 3.6. *(List of a partial path decomposition)* Let $\mathcal{P} = (Y_1, Y_2, \ldots, Y_s)$ be a partial path decomposition with interval model $\mathcal{Z} = (Z_{r_j})_{1 \leq j \leq t}$. Let the *list* of $\mathcal{P}$ be $[y] = (y^1, y^2, \ldots, y^t)$, where for $1 \leq j \leq t$, we have $y^j = (|Y_{r_j}|, |Y_{r_j+1}|, \ldots, |Y_{r_{j+1}-1}|)$. Note that $[y]$ is a list of integer sequences.

For any integer sequence a, let $l(a)$ be the number of integers in a and for any list $[y] = (y^1, y^2, \ldots, y^t)$, let $l[y] = l(y^1) + l(y^2) + \ldots + l(y^t)$. Notice the use of square brackets in the notation $l[y]$ for the length of the list $[y]$ and the use of round brackets in the notation $l(a)$ for the length of the sequence a.

For any two integer sequences $a = (a_1, a_2, \ldots, a_n)$ and $b = (b_1, b_2, \ldots, b_n)$ with $l(a) = l(b)$, define $a + b = (a_1 + b_2, a_2 + b_2, \ldots, a_n + b_n)$. For any two lists $[y_1] = (y_1^1, y_1^2, \ldots, y_1^t)$ and $[y_2] = (y_2^1, y_2^2, \ldots, y_2^t)$ with $l(y_1^j) = l(y_2^j)$, for any $1 \leq j \leq t$, define $[y_1] + [y_2] = (y_1^1 + y_2^1, y_1^2 + y_2^2, \ldots, y_1^n + y_2^n)$.

Definition 3.7. *(Typical sequence, Typical list)* Let $a = (a_1, a_2, \ldots, a_n)$ be an integer sequence. Define the *typical sequence* $\mathcal{T}(a)$ to be the sequence obtained by repeatedly applying the following operation, until it is no longer possible:

- if there exists $1 \leq i < n$ with $a_i = a_{i+1}$ replace $(a_1, a_2, \ldots, a_n)$ by $(a_1, a_2, \ldots, a_i, a_{i+2}, \ldots, a_n)$.
- if there exists $1 \leq i < n-1$ with $a_i < a_{i+1} < a_{i+2}$ or $a_i > a_{i+1} > a_{i+2}$, replace $(a_1, a_2, \ldots, a_n)$ by $(a_1, a_2, \ldots, a_i, a_{i+2}, \ldots, a_n)$.

Let $[y] = (y^1, y^2, \ldots y^t)$ be a list of integer sequences. Define the *typical list* $\mathcal{T}[y]$ to be the list $\mathcal{T}[y] = (\mathcal{T}(y^1), \mathcal{T}(y^2), \ldots, \mathcal{T}(y^t))$.

Notice the use of square brackets in the notation $\mathcal{T}[y]$ for the typical list of $[y]$ and the use of round brackets in the notation $\mathcal{T}(a)$ for the typical sequence of a. Moreover, if $[y]$ is a typical list, $\mathcal{T}[y] = [y]$ and hence we can write $[y]$ instead of $\mathcal{T}[y]$.

Definition 3.8. *(Characteristic)* Let $\mathcal{D} = (\{X_i \mid i \in I\}, T(I,F))$ be a tree decomposition. Let $i \in I$ and $\mathcal{P}$ be a partial path decomposition at i with interval model $\mathcal{Z}$ and typical list $[y]$. Define the *characteristic* of $\mathcal{P}$ at i to be $\mathcal{C} = (\mathcal{Z}, [y])$.

Definition 3.9. *(Extensions)* Let $a = (a_1, a_2, \ldots, a_n)$ be an integer sequence. Define $\mathcal{E}(a)$ to be the set of *extensions* of a:

$$\mathcal{E}(a) = \{\ a^* \mid \exists_{1=t_1<t_2<\ldots<t_{n+1}}\ \forall_{1\leq i\leq n}\ \forall_{t_i\leq j<t_{i+1}}\ [a^*_j = a_i]\ \}$$

Definition 3.10. *(Precedence)* Let a and b be two integer sequences. Define $a \preceq b$ if and only if there exist $e^a = (e^a_1, e^a_2, \ldots, e^a_t) \in \mathcal{E}(a), e^b = (e^b_1, e^b_2, \ldots, e^b_t) \in \mathcal{E}(b)$ such that for all $1 \leq i \leq t, e^a_i \leq e^b_i$.

Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two partial path decompositions rooted at the same node with the same interval model. Let their corresponding lists be $[y_1] = (y^1_1, y^2_1, \ldots y^t_1)$ and $[y_2] = (y^1_2, y^2_2, \ldots y^t_2)$. Define $\mathcal{P}_1 \preceq \mathcal{P}_2$ if and only if for all $1 \leq j \leq t, y^j_1 \preceq y^j_2$.

Definition 3.11. *(Full set of characteristics)* For $i \in I$, define $FS(i)$ to be a *full set of characteristics* at i if and only if for every partial path decomposition $\mathcal{P}$ at i of width at most k there exists a partial path decomposition $\mathcal{P}'$ at i such that $\mathcal{P}' \preceq \mathcal{P}$ and the characteristic of $\mathcal{P}'$ is in $FS(i)$.

4 Nice Tree Decompositions

Let $l \geq 1$ be a constant and G be a graph of treewidth at most l. Let $\mathcal{D}' = (\{X_i \mid i \in I'\}, T'(I', F'))$ be a width-l rooted tree decomposition of G obtained by using the algorithm of Theorem 2.1. In this section, we present a logspace algorithm to transform $\mathcal{D}'$ into a nice tree decomposition $\mathcal{D}$.

Recall that the elements of I are called the *nodes* of the tree T.

Definition 4.1. *(Nice tree decomposition)* A rooted tree decomposition $\mathcal{D} = (\{X_i \mid i \in I\}, T(I,F))$ is called a *nice tree decomposition* if the following conditions are satisfied:

- every node of T has at most two children (NTD-1)
- if a node i has two children j and k, then $X_i = X_j = X_k$ (NTD-2)
- if a node i has one child j, then exactly one of the following holds (NTD-3)
  - $|X_i| = |X_j| - 1$ and $X_i \subset X_j$
  - $|X_i| = |X_j| + 1$ and $X_i \supset X_j$

Theorem 4.2. There exists a logspace algorithm that converts $\mathcal{D}' = (\{X_i \mid i \in I'\}, T'(I', F'))$ into a width-l nice tree decomposition $\mathcal{D} = (\{X_i \mid i \in I\}, T(I,F))$. Moreover, the height of T is $O(\log n)$.

Proof. There are three properties of a nice tree decomposition: NTD-1, NTD-2 and NTD-3. Note that $\mathcal{D}'$ already satisfies NTD-1 (see Theorem 2.1).

We perform the following steps for each node p with two children (say q and r) in T'. We subdivide the edge (p,q) into (p,q') and (q',q) such that $X_{q'} = X_p$. Similarly, we subdivide the edge (p,r) into (p,r') and (r',r) such that $X_{r'} = X_p$. The tree decomposition thus constructed (say $\mathcal{D}'' = (\{X_i \mid i \in I''\}, T''(I'', F''))$) satisfies NTD-1 and NTD-2.

We perform the following steps for each node p with exactly one child (say q) in T''. If $X_p = X_q$, then we contract the edge (p,q). If $X_p \neq X_q$, let $X_{pq} = X_p \cap X_q$, $X_p \setminus X_q = \{v_{p_1}, v_{p_2}, \ldots, v_{p_s}\}$ and $X_q \setminus X_p = \{v_{q_1}, v_{q_2}, \ldots, v_{q_t}\}$. Let $X_{p_i} = X_{pq} \cup \{v_{p_1}, v_{p_2}, \ldots, v_{p_i}\}$ for $1 \leq i < s$. Let $X_{q_i} = X_{pq} \cup \{v_{q_1}, v_{q_2}, \ldots, v_{q_i}\}$ for $1 \leq i < t$. Replace the edge (p,q) by a path with nodes $p, p_{s-1}, \ldots, p_1, pq, q_1, q_2, \ldots, q_{t-1}, q$. The corresponding bags associated with these nodes are $X_p, X_{p_{s-1}}, \ldots, X_{p_1}, X_{pq}, X_{q_1}, X_{q_2}, \ldots, X_{q_{t-1}}, X_q$. Note that the length of this path is at most $2l$. The tree decomposition thus constructed (say $\mathcal{D} = (\{X_i \mid i \in I\}, T(I,F))$) satisfies NTD-1, NTD-2 and NTD-3. Moreover, the height of T is $O(\log n)$. □

Additionally, we perform the following steps for each leaf node of $\mathcal{D}$. Let p be a leaf node with $X_p = \{v_1, v_2, \ldots v_s\}$. For all $1 \leq i < s$, let $X_{p_i} = \{v_1, v_2, \ldots v_i\}$. Replace the node p by a path with nodes $p, p_{s-1}, \ldots, p_1$. The corresponding bags associated with these nodes are $X_p, X_{p_{s-1}}, \ldots, X_{p_1}$. Note that the length of this path is at most l and that the new leaf node X_{p_1} has one vertex.

Henceforth, we assume that all the leaf nodes of $\mathcal{D} = (\{X_i \mid i \in I\}, T(I,F))$ have one vertex in their bags.

The nodes of $\mathcal{D}$ are of the following four types :

Definition 4.3. *(Start, Join, Forget and Introduce nodes)* Let $\mathcal{D} = (\{X_i \mid i \in I\}, T(I,F))$ be a nice tree-decomposition and let $p \in I$. Node p is one of the following types:

- *Start*: a leaf node, having one vertex in its bag.
- *Forget*: a node with one child q, with $X_p \subset X_q$ and $|X_p| = |X_q| - 1$,
- *Introduce*: a node with one child q, with $X_p \supset X_q$ and $|X_p| = |X_q| + 1$,
- *Join*: a node with two children q and r, with $X_p = X_q = X_r$.

Since any nice tree decomposition is a rooted tree decomposition, one of the nodes in I is the root of T. We denote this node by *root*. Observe that *root* is still of one of the above four types.

5 Computing Characteristics

In this section, we show how to compute in logspace the full set of characteristics at a node, given the full sets of characteristics at its children. We then show how to compute the full set of characteristics at the root. Finally, we compute a unique characteristic at each node.

Since $FS(root) \neq \emptyset$ if and only if $pw(G) \leq k$, once we have shown that $FS(root)$ can be computed in logspace, we have solved the decision version of the problem. The results in Section 5.3 - Section 7.2 contribute only to the constructive part of the solution, in which we show that a path decomposition can be found in logspace.

Throughout this section, let $k \geq 1$ be a constant and $\mathcal{D} = (\{X_i \mid i \in I\}, T(I,F))$ be the nice tree decomposition of G computed in Section 4. Let $p, q, r \in I$ be nodes of T.

5.1 Computing the Characteristic from the Characteristics at Children

We now present a logspace algorithm to compute $FS(p)$, the full set of characteristics at p, given the full sets of characteristics at the children of p.

5.1.1 Start Node

Let $p \in I$ be a start node with $X_p = \{v\}$. By definition, $FS(p) = \{(\ (\{v\}),\ [(1)]\ ),\ (\ (\{v\},\{\emptyset\}),\ [(1),(0)]\ ),(\ (\{\emptyset\},\{v\}),\ [(0),(1)]\ ),(\ (\{\emptyset\},\{v\},\{\emptyset\}),\ [(0),(1),(0)]\ )\}$. We define the procedure ComputeStartFS(p) that outputs this set $FS(p)$. It is easy to see that ComputeStartFS(p) is a logspace function.

5.1.2 Forget Node

We first give some lemmas referring to the space requirements of an interval model, minimal path decomposition, typical sequence, typical list and full set of characteristics.

Lemma 5.1. *(Bodlaender and Kloks [BK96])* The number of different interval models at p is bounded by $(2k+3)^{2k+3}$. The number of bags in any interval model is at most $2k+3$. These bounds hold for the minimal path decompositions of $G[X_p]$ as well.

Corollary 5.2. Any interval model can be stored in logspace.

Lemma 5.3. *(Bodlaender and Kloks [BK96])* Let $a = (a_1, a_2, \ldots, a_n)$ be a sequence of nonnegative integers with $\max(a) = k+1$. Then $l(\mathcal{T}(a)) \leq 2k+3$. The number of different typical sequences of integers in $\{0, 1, \ldots, k+1\}$ is at most $\frac{8}{3}2^{2(k+1)}$.

Lemma 5.4. Let $(\mathcal{Z}, [y])$ with $\mathcal{Z} = (Z_j)_{1\leq j\leq t}$ be a characteristic at p. Then $l[y] \leq (2k+3)^2$. Furthermore, the number of different typical lists that are part of some characteristic at p is at most $(\frac{8}{3}2^{2(k+1)})^{2k+4}$.

Proof. For any $1 \leq j \leq t$, $\max(y^j) \leq k+1$ and hence by Lemma 5.3, $l(y^j) \leq 2k+3$. By Lemma 5.1, $t \leq 2k+3$. Therefore, $l[y] \leq (2k+3)^2$.

By Lemma 5.3, there are $\frac{8}{3}2^{2(k+1)}$ different typical sequences that are elements of a typical list that is part of a characteristic at p. Hence for every i, there are $(\frac{8}{3}2^{2(k+1)})^i$ different typical lists that are part of some characteristic at p and that have i typical sequences. Since the number of typical sequences in a typical list that is part of characteristic at p is at most $2k+3$, we get that the number of different typical lists that are the typical list of some characteristic at p is at most $\sum_{i=1}^{2k+3}(\frac{8}{3}2^{2(k+1)})^i \leq (\frac{8}{3}2^{2(k+1)})^{2k+4}$. □

Bodlaender and Kloks [BK96] gave a bound on the number of different characteristics of possible path-decompositions with pathwidth at most k. We will use the following slightly weaker bound, as it is enough for our purposes and it is a direct consequence of the results stated previously.

Corollary 5.5. The number of different characteristics at p is at most $(2k+3)^{2k+3}(\frac{8}{3}2^{2(k+1)})^{2k+4}$.

Let p be a forget node and q be its child such that $X_q \setminus X_p = \{v\}$. Let $\mathcal{C}_q = (\mathcal{Z}, [y]) \in FS(q)$ be a characteristic at q with $\mathcal{Z} = (Z_j)_{1\leq j\leq t}$. We define the procedure ComputeForgetFS($p, \mathcal{C}_q$) as follows.

Define $\mathcal{Z} \setminus \{v\} = (Z_j \setminus \{v\})_{1\leq j\leq t}$ and let $\mathcal{Z}'$ be the minimal path decomposition obtained by removing the consecutive duplicate bags of $\mathcal{Z} \setminus \{v\}$. Define $[y']$ as follows:

- if $\mathcal{Z} \setminus \{v\}$ does not contain any consecutive duplicate bags, then $[y'] = [y]$,
- if $\mathcal{Z} \setminus \{v\}$ contains exactly a pair of consecutive duplicate bags, say $Z_{j_1} \setminus \{v\} = Z_{j_1+1} \setminus \{v\}$, for some $1 \leq j_1 < t$, then $[y'] = (y^1, y^2, \ldots, y^{j_1-1}, y^{j_1}y^{j_1+1}, y^{j_1+2}, \ldots, y^t)$,
- if $\mathcal{Z} \setminus \{v\}$ contains two pairs of consecutive duplicate bags, but no three consecutive duplicate bags, say $Z_{j_1} \setminus \{v\} = Z_{j_1+1} \setminus \{v\}$ and $Z_{j_2} \setminus \{v\} = Z_{j_2+1} \setminus \{v\}$ for some $1 \leq j_1 < j_2 < t$, then $[y'] = (y^1, y^2, \ldots, y^{j_1-1}, y^{j_1}y^{j_1+1}, y^{j_1+2}, \ldots, y^{j_2-1}, y^{j_2}y^{j_2+1}, y^{j_2+2}, \ldots, y^t)$,
- if $\mathcal{Z} \setminus \{v\}$ contains three consecutive bags, say $Z_{j_1} \setminus \{v\} = Z_{j_1+1} \setminus \{v\} = Z_{j_1+2} \setminus \{v\}$ for some $1 \leq j_1 \leq t-2$, then $[y] = (y^1, y^2, \ldots, y^{j_1-1}, y^{j_1}y^{j_1+1}y^{j_1+2}, y^{j_1+3}, \ldots, y^t)$.

Let $[y''] = \mathcal{T}[y']$ and let the output of ComputeForgetFS$(p, \mathcal{C}_q)$ be $(\mathcal{Z}', [y''])$. Note that $(\mathcal{Z}', [y''])$ is a characteristic at p and $FS(p) = \{$ ComputeForgetFS$(p, \mathcal{C}_q) \mid \mathcal{C}_q \in FS(q)\}$. By Corollary 5.5, the number of characteristics in $FS(p)$ is at most a constant and hence, to show that $FS(p)$ can be computed in logspace from $FS(q)$, it is sufficient to show that ComputeForget FS$(p, \mathcal{C}_q)$ can be implemented in logspace.

Theorem 5.6. ComputeForgetFS$(p, \mathcal{C}_q)$ is a logspace computable function.

Proof. Let $\mathcal{C}_q = (\mathcal{Z}, [y])$ with $\mathcal{Z} = (Z_j)_{1 \leq j \leq t}$. ComputeForgetFS$(p, \mathcal{C}_q)$ works as follows.

Step 1 : Computing the interval model $\mathcal{Z}'$

By Corollary 5.2, $\mathcal{Z}$ can be stored in logspace and hence, we can compute $\mathcal{Z} \setminus \{v\}$ in logspace. Now we eliminate the consecutive duplicate bags of $\mathcal{Z} \setminus \{v\}$ in logspace. Iterate the bags of $\mathcal{Z} \setminus \{v\}$. Always output $Z_1 \setminus \{v\}$. Let the current bag be $Z_j \setminus \{v\}$ and the last bag output be $Z_{j'} \setminus \{v\}$. We can determine in logspace if $Z_{j'} \setminus \{v\} \neq Z_j \setminus \{v\}$ using the algorithm from Lemma 5.7. If $Z_{j'} \setminus \{v\} \neq Z_j \setminus \{v\}$, output $Z_j \setminus \{v\}$ and otherwise, store j and continue the iteration of the bags of $\mathcal{Z} \setminus \{v\}$.

Step 2 : Computing the typical list $[y'']$

The previous step computes the indices j such that $Z_j \setminus \{v\} = Z_{j-1} \setminus \{v\}$. Now we can iterate the sequences of $[y]$ and using the definition of $[y']$, compute $[y']$ in logspace. By Lemma 5.4, $l[y]$ is at most a constant. We now apply the algorithm of Lemma 5.8 to compute $[y'']$. □

Lemma 5.7. Let $Y_1, Y_2 \subseteq X_p$ with $|Y_1| \leq k+1$ and $|Y_2| \leq k+1$. There exists a logspace algorithm deciding whether $Y_1 = Y_2$.

Proof. Iterate the vertices in Y_1 and for each current vertex v, iterate the vertices of Y_2, comparing each of them with v. Since Y_1 and Y_2 have at most some constant number of vertices, each with a logarithmic label, this can be done in logspace. If v is not found in Y_2, conclude that $Y_1 \neq Y_2$ and terminate. If we reach the end of the iteration without terminating prematurely, it means that $Y_1 \subseteq Y_2$. Now repeat the above procedure for the pair (Y_2, Y_1). If this also does not terminate prematurely, we have $Y_2 \subseteq Y_1$ and therefore, we conclude that $Y_1 = Y_2$ and terminate. □

Lemma 5.8. Let $n \leq c(k)$, where $c(k)$ is a constant depending on k. Let $a = (a_1, a_2, \ldots a_n)$ be a sequence of integers such that for $1 \leq i \leq n$, a_i can be stored in logspace. There exists a logspace algorithm computing $\mathcal{T}(a)$.

Proof. Iterate the integers of a. Output a_1 and proceed to the next element. Let a_i be the current integer and $a_{i'}$ be the last integer output. If $i = n$, output a_i. Otherwise, output a_i if and only if one of the following holds:

- $a_i > a_{i'}$ and $a_i > a_{i+1}$
- $a_i < a_{i'}$ and $a_i < a_{i+1}$

From the definition of a typical sequence, we see that this procedure correctly outputs $\mathcal{T}(a)$. Since each of the integers of a can be stored in logspace, this is a logspace algorithm. □

5.1.3 Introduce Node

Let p be an introduce node and q be its child such that $X_p \setminus X_q = \{v\}$.

Definition 5.9. *(Split)* Let $a = (a_1, a_2, \ldots, a_n)$ be an integer sequence.

- For $1 \leq i < n$, define $((a_1, a_2, \ldots, a_i), (a_{i+1}, a_{i+2}, \ldots, a_n))$ to be a *split of type I* of a at i.
- For $1 \leq i \leq n$, define $((a_1, a_2, \ldots, a_i), (a_i, a_{i+1}, \ldots, a_n))$ to be a *split of type II* of a at i.
- Define (a^1, a^2) to be a *split* of a if there exists $1 \leq i \leq n$ such that (a^1, a^2) is a split of type I or II of a at i.

Let $\mathcal{C}_q = (\mathcal{Z}, [y])$ be a characteristic at q with $\mathcal{Z} = (Z_j)_{1 \leq j \leq t}$. Define the procedure ComputeIntroduceFS$(p, \mathcal{C}_q)$ as follows.

First compute the set $S(\mathcal{Z})$ of all minimal path decompositions $\mathcal{Z}' = (Z'_j)_{1 \leq j \leq t'}$ for $G[X_p]$, such that by removing the consecutive duplicate bags of $\mathcal{Z}' \setminus \{v\}$, we obtain $\mathcal{Z}$, where $\mathcal{Z}' \setminus \{v\} = (Z'_j \setminus \{v\})_{1 \leq j \leq t'}$.

For every $\mathcal{Z}' \in S(\mathcal{Z})$, do the following. Let l, r be the smallest and largest indices such that $v \in Z'_l$ and $v \in Z'_r$, resp. For any integer sequence $a = (a_1, a_2, \ldots, a_n)$, we denote $a + 1 = (a_1 + 1, a_2 + 1, \ldots, a_n + 1)$. Define the set of typical lists $S'(\mathcal{Z}')$ as follows:

- if $t' = t$, then $S'(\mathcal{Z}') = \{(y^1, y^2, \ldots, y^{l-1}, y^l + 1, y^{l+1} + 1, \ldots, y^r + 1, y^{r+1}, \ldots, y^t)\}$,
- if $t' = t + 1$ and $Z'_l \setminus \{v\} = Z'_{l-1}$, then $S'(\mathcal{Z}') = \{(y^1, y^2, \ldots, y^{l-2}, (y')^{l-1}, (y')^l + 1, y^l + 1, \ldots, y^{r-1} + 1, y^r, \ldots, y^t) \mid ((y')^{l-1}, (y')^l)$ is a split of $y^{l-1}\}$,
- if $t' = t + 1$ and $Z'_r \setminus \{v\} = Z'_{r+1}$, then $S'(\mathcal{Z}') = \{(y^1, y^2, \ldots, y^{l-1}, y^l + 1, \ldots, y^{r-1} + 1, (y')^r + 1, (y')^{r+1}, y^{r+1}, \ldots, y^t) \mid ((y')^r, (y')^{r+1})$ is a split of $y^r\}$,
- if $t' = t + 2$ and $l < r$, then $S'(\mathcal{Z}') = \{(y^1, y^2, \ldots, y^{l-2}, (y')^{l-1}, (y')^l + 1, y^l + 1, \ldots, y^{r-2} + 1, (y')^r + 1, (y')^{r+1}, y^r, \ldots, y^t) \mid ((y')^{l-1}, (y')^l)$ is a split of y^{l-1} and $((y')^r, (y')^{r+1})$ is a split of $y^{r-1}\}$,
- if $t' = t + 2$ and $l = r$, then $S'(\mathcal{Z}') = \{(y^1, y^2, \ldots, y^{l-2}, (y')^{l-1}, (y')^l + 1, (y')^{l+1}, y^l, \ldots, y^t) \mid ((y')^{l-1}, y'')$ is a split of y^{l-1} and $((y')^l, (y')^{l+1})$ is a split of $y''\}$.

For all $[y'] \in S'(\mathcal{Z}')$, output $(\mathcal{Z}', [y'])$. Note that the output of ComputeIntroduceFS$(p, \mathcal{C}_q)$ is a set of characteristics at p and $FS(p) = \{$ ComputeIntroduceFS$(p, \mathcal{C}_q) \mid \mathcal{C}_q \in FS(q)\}$. Hence, to show that $FS(p)$ can be computed in logspace from $FS(q)$, it is sufficient to show that ComputeIntroduceFS(p, C_q) can be implemented in logspace.

Theorem 5.10. ComputeIntroduceFS(p, C_q) is a logspace computable function.

Proof. Let $\mathcal{C}_q = (\mathcal{Z}, [y])$ with $\mathcal{Z} = (Z_j)_{1 \leq j \leq t}$. ComputeIntroduceFS$(p, \mathcal{C}_q)$ works as follows.

Step 1 : Computing the minimal path decompositions for $G[X_p]$

By Lemma 5.1, the number of bags in a minimal path decomposition for $G[X_p]$ is bounded by a constant (say c). We first see that we can iterate in logspace the sequences of integers $y = (y_1, y_2, \ldots, y_s)$ with $s \leq c$ and for all $1 \leq i \leq s,\ y_i \leq k+1$. Since s is at most some constant and each for $1 \leq i \leq s$, y_i is also at most some constant, we can store y in constant space. We can iterate these sequences in logspace by storing only the last computed sequence and at the next step computing the next valid sequence in lexicographical order.

For each such sequence y, we see that we can iterate in logspace the sequences of bags $Y = (Y_1, Y_2, \ldots, Y_s)$ such that for all $1 \leq i \leq s,\ Y_i \subseteq X_p$ and $|Y_i| = y_i$. Since s is at most some constant and for $1 \leq i \leq s,\ Y_i$ contains at most some constant number of vertices, each with a logarithmic label, Y can be stored in logspace. We can iterate these sequences in logspace by storing only the last computed sequence and at the next step computting the next valid sequence in lexicographical order.

For each such sequence Y, we shall now prove that we can decide in logspace whether Y is a path decomposition. We check in logspace each of the three defining properties of a path decomposition (see Definition 1.2).

- PD-1: Iterate all vertices in X_p. Since X_p has at most some constant number of vertices, each with a logarithmic label, this can be done in logspace. For each current vertex u, iterate the vertices in each of the bags in Y, comparing each of them with u. Since Y can be stored in logspace, this can be done in logspace. Decide that Y has property PD-1 iff for all vertices u, there exists $1 \leq i \leq s$ such that $u \in Y_i$.

- PD-2: Iterate in logspace all edges in X_p. For each current edge (u, w), iterate the vertices in each of the bags in Y, comparing each of them with u and w. Decide that Y has property PD-2 iff for all edges (u, w) there exists $1 \leq i \leq s$ such that $u, w \in Y_i$.

- PD-3: Iterate in logspace the vertices in X_p. For each current vertex u, iterate the vertices in each of the bags in Y, comparing each of them with u. During this procedure, store two boolean variables $found$ and $lost$. Initially, $found = false$ and $lost = false$. Let the current bag be Y_i. If $u \in Y_i$, set $found = true$ and if $found = true$ and $u \notin Y_i$, set $lost = true$. If at some point during the iteration, we have $found = true, lost = true$ and $u \in Y_i$, conclude that Y does not have have PD-3 and terminate. If the iteration of the vertices in X_p does not terminate prematurely, conclude that Y has property PD-3.

For each path decomposition Y, we can decide in logspace whether Y is minimal. Iterate in logspace consecutive pairs of bags of Y. For each current pair of bags (Y_i, Y_{i+1}) apply the logspace algorithm of Lemma 5.7 to determine whether they are equal. If they are found equal, terminate the procedure and conclude that Y is not minimal. If no pair of bags causes the termination of the procedure, conclude that Y is indeed a minimal path decomposition.

Applying successively the above procedures yields a logspace algorithm that computes all minimal path decompositions for $G[X_p]$.

Step 2 : Computing the interval models in $S(\mathcal{Z})$

For each minimal path decompositions $\mathcal{Z}'$, we want to determine in logspace the sequence of bags $\mathcal{Z}''$ obtained by removing the consecutive duplicate bags of $\mathcal{Z}' \setminus \{v\} = (Z'_j \setminus \{v\})_{1 \leq j \leq t'}$. This can be done using the algorithm from the "Computing the interval model" step of the ComputeForgetFS$(p, \mathcal{C}_q)$ procedure.

Let $\mathcal{Z}'' = (Z''_j)_{1 \leq j \leq t''}$. We now check if $\mathcal{Z}'' = \mathcal{Z}$. Because $\mathcal{Z}'$ can be stored in logspace, t'' can be determined in logspace. If $t \neq t''$, conclude that $\mathcal{Z}'' \neq \mathcal{Z}$ and hence $\mathcal{Z}'' \notin S(\mathcal{Z})$. Otherwise, for all $1 \leq j \leq t = t''$, determine in logspace if $Z''_j \neq Z_j$ using the algorithm of Lemma 5.7. Conclude that $\mathcal{Z}'' \notin S(\mathcal{Z})$ iff there exists j such that $Z''_j \neq Z_j$.

Step 3 : Computing the typical lists in $S'(\mathcal{Z}')$

Let $\mathcal{Z}' \in S(\mathcal{Z})$ and let l, r be the smallest and largest indices such that $v \in Z'_l$ and $v \in Z'_r$, resp. We now see that l and r can be computed in logspace. Iterate the bags in $\mathcal{Z}'$. Let Z'_j be the current bag. If $v \in Z'_j$, set $l = j$ and terminate the iteration. Now iterate all the pairs of consecutive bags in $\mathcal{Z}'$. Let (Z'_j, Z'_{j+1}) be the current pair of bags. If $v \in Z'_j$ and $v \notin Z'_{j+1}$, set $r = j$ and terminate the iteration.

For all $1 \leq j \leq t$, we can compute all splits of y^j in logspace. Let $y^j = (y^j_1, y^j_2, \ldots, y^j_{l(y^j)})$. For each $1 \leq j' < l(y^j)$, output $((y^j_1, y^j_2, \ldots, y^j_{j'}), (y^j_{j'+1}, y^j_{j'+2}, \ldots, y^j_{l(y^j)}))$ and for all $1 \leq j' \leq l(y^j)$, output $((y^j_1, y^j_2, \ldots, y^j_{j'}), (y^j_{j'}, y^j_{j'+1}, \ldots, y^j_{l(y^j)}))$. By Lemma 5.4, $[y]$ can be stored in logspace and thus, this procedure can be done in logspace.

Assume $t' = t + 1$ and $Z'_l \setminus \{v\} = Z'_{l-1}$. The other cases are treated similarly. Iterate all possible splits of y^{l-1}. For each current split $((y')^{l-1}, (y')^l)$, iterate the sequences of $[y]$ and output $(y^1, y^2, \ldots, y^{l-2}, (y')^{l-1}, (y')^l + 1, y^l + 1, \ldots, y^{r-1} + 1, y^r, \ldots, y^t)$. Since $[y]$ can be stored in constant space and all the splits can be computed in logspace, this is a logspace procedure. □

5.1.4 Join Node

Let p be a join node and q, r be its children such that $X_p = X_q = X_r$.

Let $\mathcal{C}_q = (\mathcal{Z}_q, [y_q]) \in FS(q)$ and $\mathcal{C}_r = (\mathcal{Z}_r, [y_r]) \in FS(r)$ be characteristics at q and r, respectively. We define the procedure ComputeJoinFS$(p, \mathcal{C}_q, \mathcal{C}_r)$ as follows.

If $\mathcal{C}_q$ and $\mathcal{C}_r$ do not have the same interval model, the procedure terminate without any output. Otherwise, let $\mathcal{Z} = \mathcal{Z}_q = \mathcal{Z}_r$ with $\mathcal{Z} = (Z_j)_{1 \leq j \leq t}$. Let the procedure output all characteristics of the form $(\mathcal{Z}, [y'_p])$, where $[y'_p] = \mathcal{T}[y_p]$ and for all $1 \leq j \leq t, y^j_p = e^j_q + e^j_r - |Z_j|$, for some $e^j_q \in \mathcal{E}(y^j_q), e^j_r \in \mathcal{E}(y^j_r)$, with $l(e^j_q) = l(e^j_r) \leq l(y^j_q) + l(y^j_r)$ and $\max(y^j_p) \leq k + 1$.

The output of ComputeJoinFS$(p, \mathcal{C}_q, \mathcal{C}_r)$ is a set of characteristics at p and moreover, $FS(p) = \{$ ComputeJoinFS$(p, \mathcal{C}_q, \mathcal{C}_r) \mid \mathcal{C}_q \in FS(q), \mathcal{C}_r \in FS(r)\}$. Hence, to show that $FS(p)$ can be computed in logspace from $FS(q)$ and $FS(r)$, it is sufficient to show that ComputeJoinFS$(p, \mathcal{C}_q, \mathcal{C}_r)$ can be implemented in logspace.

Theorem 5.11. ComputeJoinFS$(p, \mathcal{C}_q, \mathcal{C}_r)$ is a logspace computable function.

Proof. Let $\mathcal{C}_q = (\mathcal{Z}_q, [y_q])$ and $\mathcal{C}_r = (\mathcal{Z}_r, [y_r])$. ComputeJoinFS$(p, \mathcal{C}_q, \mathcal{C}_r)$ works as follows.

Using the algorithm of Lemma 5.7, we can determine in logspace whether $\mathcal{Z}_q = \mathcal{Z}_r$. If $\mathcal{Z}_q \neq \mathcal{Z}_r$, do not output anything and terminate. Otherwise, let $\mathcal{Z} = \mathcal{Z}_q = \mathcal{Z}_r$, with $\mathcal{Z} = (Z_j)_{1 \leq j \leq t}$.

For $1 \leq j \leq t$, we can iterate in logspace all extensions $e^j_q \in \mathcal{E}(y^j_q)$ with $l(e^j_q) \leq l(y^j_q) + l(y^j_r)$. By Lemma 5.4, $l(y^j_q)$ and $l(y^j_r)$ are at most some constants and hence $l(e^j_q)$ is also at most some constant.

Iterate all nondecreasing sequences of integers $(i_1, i_2, \ldots i_l)$ with $l \leq l(y_q^j) + l(y_r^j), i_1 = 1, i_l = l(y_q^j)$ and $i_2 \leq i_1+1, i_3 \leq i_2+1, \ldots, i_l \leq i_{l-1}+1$. Since l is at most some constant and each of $i_1, i_2, \ldots, i_l$ is at most some constant, the sequence $(i_1, i_2, \ldots i_l)$ can be stored in logspace. The iteration over these sequences can be done in logspace, as we can store the last sequence output and then compute the next valid one in lexicographical ordering. For each current sequence $(i_1, i_2, \ldots, i_l)$, output $((y_q^j)_{i_1}, (y_q^j)_{i_2}, \ldots, (y_q^j)_{i_l})$.

We now see that we can compute the set of typical lists $S^j = \{(y'_p)^j \mid (y'_p)^j = \mathcal{T}(y_p^j),\ y_p^j = e_q^j + e_r^j - |Z_j| \text{ for some } e_q^j \in \mathcal{E}(y_q^j), e_r^j \in \mathcal{E}(y_r^j), \text{ with } l(e_q^j) = l(e_r^j) \leq l(y_q^j) + l(y_r^j) \text{ and } \max(y_p^j) \leq k+1\}$. For each extension e_q^j computed by the previous algorithm, we can similarly iterate all extensions $e_r^j \in \mathcal{E}(y_r^j)$ with $l(e_r^j) = l(e_q^j)$. For each current pair of extensions (e_q^j, e_r^j), compute $y_p^j = e_q^j + e_r^j - |Z_j|$. Since Z_j has a constant number of vertices, we can compute $|Z_j|$ in logspace. Moreover, $l(e_q^j) = l(e_r^j)$ is at most some constant and each of the integers of e_q^j and e_r^j are at most some constants and hence we can compute y_p^j in logspace. Using the algorithm of Lemma 5.8, we can compute $(y'_p)^j$ in logspace. Add $(y'_p)^j$ to S^j iff $\max(y_p^j) \leq k+1$.

Hence, we can compute $S^1, S^2, \ldots S^t$ in logspace. Now iterate all typical lists $((y'_p)^1, (y'_p)^2, \ldots, (y'_p)^t)$ such that for all $1 \leq j \leq t, (y'_p)^j \in S^j$. By Lemma 5.4, each typical sequence can be stored in logspace and by Lemma 5.1, t is at most some constant. Hence, we can store each element of the iteration in logspace. At the next step, we can compute the next valid sequence in lexicographical ordering and hence, this iteration can be done in logspace. This produces the desired output. □

5.2 Computing the Full Set of Characteristics at the Root

Lindell in [Lin92] gave a logspace algorithm for tree traversal. We give a brief description of this algorithm in Appendix B. In this section, we extend his algorithm to obtain a logspace algorithm for computing $FS(root)$, where *root* is the root node of T.

Let $p \in I$. We will see that $FS(p)$ can be stored in logspace. We now introduce a relabeling function, which transforms $FS(p)$ so that it can be stored in only constant space. This is crucial to proving the logarithmic space bound of our entire algorithm.

Definition 5.12. *(Relabeling)* Let $V = \{v_1, v_2, \ldots, v_n\}$ and $X_p = \{v_{p_1}, v_{p_2}, \ldots, v_{p_s}\}$ with $1 \leq p_1 < p_2 < \ldots p_s \leq n$ and let $\mathcal{C} = (\mathcal{Z}, [y]) \in FS(p)$ with $\mathcal{Z} = (Z_j)_{1 \leq j \leq t}$.

We define the *relabeling* function $\mathcal{R}$ as follows:

- for $1 \leq i \leq s$, the relabeling of v_{p_i} at p is $\mathcal{R}(p, v_{p_i}) = i$,
- the relabeling of $\mathcal{Z}$ at p is $\mathcal{R}(p, \mathcal{Z}) = \mathcal{Z}'$, where $\mathcal{Z}' = (Z'_j)_{1 \leq j \leq t}$ and for all $1 \leq j \leq t,\ Z'_j = \{\mathcal{R}(p, v) \mid v \in Z_j\}$,
- the relabeling of $\mathcal{C}$ at p is $\mathcal{R}(p, \mathcal{C}) = (\mathcal{R}(p, \mathcal{Z}), [y])$,
- the relabeling of $FS(p)$ at p is $\mathcal{R}(p, FS(p)) = \{\mathcal{R}(p, \mathcal{C}_p) \mid \mathcal{C}_p \in FS(p)\}$.
- the relabeling of *null* is $\mathcal{R}(p, null) = null$.

Define the *inverse relabeling* $\mathcal{R}^{-1}$ to be inverse function of the relabeling function.

Lemma 5.13. $FS(p)$ can be stored in $(2k+3)^{2k+3}(\frac{8}{3}2^{2(k+1)})^{2k+4}[(2k+3)(k+1)\log n + (\log(k+1))^{(2k+3)^2}]$ space and $\mathcal{R}(p, FS(p))$ can be stored in $(2k+3)^{2k+3}(\frac{8}{3}2^{2(k+1)})^{2k+4}[(2k+3)(k+1)\log k + (\log(k+1))^{(2k+3)^2}]$ space.

Proof. By Corollary 5.5, the number of different characteristics is at most $(2k+3)^{2k+3}(\frac{8}{3}2^{2(k+1)})^{2k+4}$.

Let $\mathcal{C} = (\mathcal{Z}, [y]) \in FS(p)$ with $\mathcal{Z} = (Z_j)_{1 \leq j \leq t}$. By Lemma 5.1, $t \leq 2k+3$ and since $|Z_j| \leq k+1$ and each vertex has logarithmic labels, we can store $\mathcal{Z}$ in $(2k+3)(k+1)\log n$ space. Each vertex in each of the bags of $\mathcal{R}(p, \mathcal{Z})$ has label of size $\log k$ and hence $\mathcal{R}(p, \mathcal{Z})$ can be stored in $(2k+3)(k+1)\log k$ space. By Lemma 5.4, $l[y] \leq (2k+3)^2$ and since each of the integers in $[y]$ is at most $k+1$, $[y]$ can be stored in $(\log(k+1))^{(2k+3)^2}$ space. Thus, $\mathcal{C}$ can be stored in $(2k+3)(k+1)\log n + (\log(k+1))^{(2k+3)^2}$ space and $\mathcal{R}(p, \mathcal{C})$ can be stored in $(2k+3)(k+1)\log k + (\log(k+1))^{(2k+3)^2}$ space. □

Lemma 5.14. $\mathcal{R}(p, FS(p))$ and $\mathcal{R}^{-1}(p, \mathcal{R}(p, FS(p))$ are logspace computable functions.

Proof. Since the class of logspace computable functions is closed under composition, to show that $\mathcal{R}(p, FS(p))$ is a logspace function, it is enough to show that for any $v \in X_p$, $\mathcal{R}(p, v)$ is a logspace function. Iterate the vertices of X_p, keeping a counter recording the number of vertices seen so far with labels smaller than or equal to the label of v. Let $\mathcal{R}(p, v)$ be the value of the counter at the end of the iteration.

To show that $\mathcal{R}^{-1}(p, \mathcal{R}(p, FS(p)))$ is a logspace function, it is enough to show that for any $1 \leq i \leq |X_p|$, $\mathcal{R}^{-1}(p, i)$ is a logspace function. Iterate the vertices of X_p and output the current node v iff $\mathcal{R}(p, v) = i$. □

Theorem 5.15. The full set of characteristics $FS(root)$ can be computed in logspace.

Proof. Perform a logspace depth-first search tree traversal of T using the algorithm of [Lin92], described in Section B. We choose a non-recursive implementation of DFS, with the nodes to be explored added to a stack. For the join nodes, we consistently explore the child with the smaller label first.

During the iteration, we store the following relabelings. We store $\mathcal{R}(q, FS(q))$, where q is the last node popped off the DFS stack. We also store a stack *stackRelabelingFS* of all $\mathcal{R}(p', FS(p'))$, where the parent p'' of p' has two children and we have computed $\mathcal{R}(p', FS(p'))$, but not $\mathcal{R}(p'', FS(p''))$. To maintain this stack, we push $\mathcal{R}(p, FS(p))$ on *stackRelabelingFS*, each time a node p is popped off the DFS stack and p has a sibling with a larger label and we pop an element off *stackRelabelingFS* each time a join node p is popped off the DFS stack.

We now see that with the above information we can compute $\mathcal{R}(p, FS(p))$ each time a node p is popped off the DFS stack. Perform the stack-based DFS traversal and let p be the node currently popped off the DFS stack. Then:

- if p is a start node, compute $FS(p) = \textsc{ComputeStartFS}(p)$ and then find $\mathcal{R}(p, FS(p))$.
- if p is a forget node, let q be its child. Then q is the last node popped and we have already computed and stored $\mathcal{R}(q, FS(q))$, so we can compute $FS(q) = \mathcal{R}^{-1}(q, \mathcal{R}(q, FS(q)))$. Then compute $FS(p) = \textsc{ComputeForgetFS}(p, FS(q))$ and finally $\mathcal{R}(p, FS(p))$.
- if p is an introduce node, proceed as in the case when p is a forget node.
- if p is a join node, let q and r be its children, such that q is lexicographically smaller than r. Then the last node popped is r and hence we have already computed and stored $\mathcal{R}(r, FS(r))$. Furthermore, observe that q is on top of *stackRelabelingFS*, so we know $\mathcal{R}(q, FS(q))$. We can now compute $FS(q) = \mathcal{R}^{-1}(q, \mathcal{R}(q, FS(q)))$ and $FS(r) = \mathcal{R}^{-1}(r, \mathcal{R}(r, FS(r)))$ and then compute $FS(p) = \textsc{ComputeJoinFS}(p, FS(q), FS(r))$ and finally $\mathcal{R}(p, FS(p))$.

We now see that the algorithm can be implemented in logspace. By Lemma 5.13, $\mathcal{R}(p, FS(p))$ can be stored in constant space for any node p and since the depth of T is $O(\log n)$, *stackRelabeling* FS can be stored in logspace. By Section 5.1, the procedures ComputeStartFS, ComputeForgetFS, ComputeIntroduceFS and ComputeJoinFS are logspace functions and by Lemma 5.14, the functions $\mathcal{R}$ and $\mathcal{R}^{-1}$ are also logspace functions. The result follows since the class of logspace computable functions is closed under composition. □

5.3 Computing a Unique Characteristic at Each Node

In this section, we see how to compute in logspace a unique characteristic at each node such that there exists a path decomposition $(Y_1, Y_2, \ldots, Y_s)$ of G such that for every node t of T, the unique characteristic at t is the characteristic of the partial path decomposition $(Y_1 \cap G_t, Y_2 \cap G_t, \ldots, Y_s \cap G_t)$. Recall that G_t denotes the subgraph of G rooted at t.

Let $p \in I, \mathcal{C}_p \in FS(p)$ and q be a child of p. Let $p' \in I$ be a descendant of p in T. We define a unique characteristic at p, at q and then at p'.

Definition 5.16. *(Descendant characteristic)* Define the *descendant characteristic* $dc(p, p, \mathcal{C}_p)$ of $\mathcal{C}_p$ at p for p to be $\mathcal{C}_p$.

Define the *descendant characteristic* $dc(p, q, \mathcal{C}_p)$ of $\mathcal{C}_p$ at p for q as follows:

- if p is a forget node, then $dc(p, q, C_p)$ is the lexicographically smallest characteristic in $\{\mathcal{C}_q \mid \mathcal{C}_q \in FS(q), \mathcal{C}_p = \text{ComputeForgetFS}(p, \mathcal{C}_q)\}$,
- if p is an introduce node, then $dc(p, q, C_p)$ is the lexicographically smallest characteristic in $\{\mathcal{C}_q \mid \mathcal{C}_q \in FS(q), \mathcal{C}_p \in \text{ComputeIntroduceFS}(p, \mathcal{C}_q)\}$,
- if p is a join node, let r be its other child. Then,
  - if q is lexicographically smaller than r, let $(dc(p, q, \mathcal{C}_p), dc(p, r, \mathcal{C}_p))$ be the lexicographically smallest pair of characteristics in $\{(\mathcal{C}_q, \mathcal{C}_r) \mid \mathcal{C}_q \in FS(q), \mathcal{C}_r \in FS(r), \mathcal{C}_p \in \text{ComputeJoinFS}(p, \mathcal{C}_q, \mathcal{C}_r)\}$.
  - otherwise, let $(dc(p, r, \mathcal{C}_p), dc(p, q, \mathcal{C}_p))$ be the lexicographically smallest pair of characteristics in $\{(\mathcal{C}_r, \mathcal{C}_q) \mid \mathcal{C}_q \in FS(q), \mathcal{C}_r \in FS(r), \mathcal{C}_p \in \text{ComputeJoinFS}(p, \mathcal{C}_r, \mathcal{C}_q)\}$.

Let $p = p_0, p_1, \ldots p_{t-1}, p_t = p'$ be the nodes in order on the path in T between p and p'. Let $\mathcal{C}_{p_0} = \mathcal{C}_p$ and for $0 \leq i < t$, let $\mathcal{C}_{p_{i+1}} = dc(p_i, p_{i+1}, \mathcal{C}_{p_i})$. Define the *descendant characteristic* $dc(p, p', \mathcal{C}_p)$ of $\mathcal{C}_p$ at p for p' to be $\mathcal{C}_{p'} = \mathcal{C}_{p_t}$.

Theorem 5.17. There exists a logspace algorithm ComputeDescendantCharacteristic$(p, p', \mathcal{C}_p)$ that outputs the descendant characteristic $dc(p, p', \mathcal{C}_p)$.

Proof. We first show that the statement holds when p' is q, a child of p. Theorem 5.15 applied to the subtree of T rooted at q gives $FS(q)$ in logspace.

- If p is a forget node, iterate the characteristics in $FS(q)$, storing a characteristic $\mathcal{C}_{min}$, initialized with *null*, where *null* is defined to be lexicographically larger than any characteristic. If the current characteristic is $\mathcal{C}_q$ such that $\mathcal{C}_p = \text{ComputeForgetFS}(p, \mathcal{C}_q)$ and $\mathcal{C}_q$ is lexicographically smaller than $\mathcal{C}_{min}$, let $\mathcal{C}_{min}$ be $\mathcal{C}_q$. Return the value of $\mathcal{C}_{min}$ at the end of the iteration.

- If p is an introduce node, proceed similarly to the case when p is a forget node.
- If p is a join node, let r be the other child of p. Suppose wlog q is lexicographically smaller than r. Compute $FS(r)$ in logspace. Iterate the characteristics in $FS(q)$ and for each of them, iterate the characteristics in $FS(r)$, storing a characteristic $\mathcal{C}_{min}$, initialized with *null*. If the current characteristics are $\mathcal{C}_q \in FS(q), \mathcal{C}_r \in FS(r)$ such that $\mathcal{C}_p \in \textsc{ComputeJoinFS}(p, \mathcal{C}_q, \mathcal{C}_r)$ and $\mathcal{C}_q$ is lexicographically smaller than $\mathcal{C}_{min}$, let $\mathcal{C}_{min}$ be $\mathcal{C}_q$. Return the value of $\mathcal{C}_{min}$ at the end of the iteration.

Observe that $dc(p, q, \mathcal{C}_p)$ is a logspace function, because the functions ComputeForgetFS, ComputeIntroduceFS, ComputeJoinFS are logspace functions by Theorem 5.6, Theorem 5.10 and Theorem 5.11 and $FS(q)$ and $\mathcal{C}_{min}$ can be stored in logspace by Lemma 5.13.

For any ancestor $p^* \in I$ of p', define $child(p^*, p')$ to be the child of p^* that is an ancestor of p'. We now see that $child(p^*, p')$ can be implemented in logspace. Iterate the nodes on the path from p' to p^*, storing the current node p^{**} and its child p^{***} that is an ancestor of p'. Initialize $p^{**} = p'$ and $p^{***} = null$. At each step, update p^{***} by p^{**} and p^{**} by $parent(p^{**})$. Stop the iteration when $p^{**} = p^*$ and output p^{***}.

We now show that $dc(p, p', \mathcal{C}_p)$ is a logspace function. Iterate the nodes on the path from the p to p', storing the current node p^* and a characteristic $\mathcal{C}^*$ at p^*. Initialize $p^* = p$ and $\mathcal{C}^* = \mathcal{C}_p$. Replace $\mathcal{C}^*$ by $dc(p^*, child(p^*, p'), \mathcal{C}^*)$ and then p^* by $child(p^*, p')$. Stop when $p^* = p'$ and output the current value of $\mathcal{C}^*$. □

Definition 5.18. *(Unique characteristic)* Let $\mathcal{C}_{root} \in FS(root)$ be the lexicographically smallest characteristic in $FS(root)$. Define the *unique characteristic* $\mathcal{C}^*_p$ at p to be the descendant characteristic $dc(root, p, \mathcal{C}_{root})$.

It follows from Theorem 5.17 that the unique characteristic $\mathcal{C}^*_p$ at p can be computed in logspace.

6 Computing the Gap List

In this section, we give a logspace algorithm for computing the gap list, an auxiliary data structure useful for computing the path decompositions. We first show how to compute the gap list at a node given the gap list at its children and then show how to compute it at the root.

Let $p \in I$ and $\mathcal{C}^*_p = (\mathcal{Z}, [y])$ be the unique characteristic at p, with $\mathcal{Z} = (Z_j)_{1 \leq j \leq t}$ and $[y] = (y^1, y^2, \ldots, y^t)$.

Definition 6.1. *(Characteristic path decomposition)* Let the *characteristic path decomposition* $\mathcal{CP}(p) = (Y_1, Y_2, \ldots Y_s)$ be the width-k path decomposition at p output by the algorithm of [BK96] given $\mathcal{D}$ and choosing $\mathcal{C}^*_{root}$ as the characteristic at the root.

We now define the gap list at p and see that it can be stored in logspace. Each of the integers in the typical list of $\mathcal{C}^*_p$ is the cardinality of some bag in $\mathcal{CP}(p)$. We define the gap list as the list of the number of bags in $\mathcal{CP}(p)$ between any two bags corresponding to consecutive elements in the typical list.

Definition 6.2. *(Gap list)* Let $1 \leq j \leq t$ and $y^j = (|Y_{i_1^j}|, |Y_{i_2^j}|, \ldots, |Y_{i_{l(y^j)}^j}|)$. Define

$$g(p)^j = (i_2^j - i_1^j - 1, i_3^j - i_2^j - 1, \ldots, i_{l(y^j)}^j - i_{l(y^j)-1}^j - 1, i_1^{j+1} - i_{l(y^j)}^j - 1),$$

where if $j = t$, we define $i_1^{j+1} = s + 1$. Define $g(p) = (g(p)^1, g(p)^2, \ldots, g(p)^t)$ to be the *gap list* at p.

Notice that $g(p)$ can be stored in logspace. For any $1 \leq j \leq t$ and $1 \leq i \leq l(y^j)$, we have $g(p)_i^j \leq s$ and since s can be stored in logspace, $g(p)_i^j$ can also be stored in logspace. Moreover, $l[g(p)] = l[y]$ and by Lemma 5.4, $l[y]$ is at most a constant (depending on k). The desired result now follows since $g(p)$ contains at most a constant number of integers, each of which can be stored in logspace.

6.1 Computing the Gap List from the Gap Lists at Children

In this section, we see how to compute in logspace $g(p)$, given the gap lists at the children of p.

6.1.1 Start Node

Let p be a start node and $g(p)$ be its gap list. Observe that:

- if $\mathcal{C}_p^* = (\ (\{v\}),\ [(1)]\ )$, then $g(p) = [(0)]$,
- if $\mathcal{C}_p^* = (\ (\{v\}, \{\emptyset\}),\ [(1), (0)]\ )$, then $g(p) = [(0), (0)]$,
- if $\mathcal{C}_p^* = (\ (\{\emptyset\}, \{v\}),\ [(0), (1)]\ )$, then $g(p) = [(0), (0)]$,
- if $\mathcal{C}_p^* = (\ (\{\emptyset\}, \{v\}, \{\emptyset\}),\ [(0), (1), (0)]\ )$, then $g(p) = [(0), (0), (0)]$.

Define the procedure ComputeStartGapList(p) to compute the gap list $g(p)$. It is easy to see that ComputeStartGapList(p) is a logspace computable function.

6.1.2 Forget Node

Let p be a forget node and q be its child. Let $\mathcal{C}_p^* = (\mathcal{Z}', [y'']), \mathcal{C}_q^* = (\mathcal{Z}, [y])$ be the unique characteristics at p, q, resp, with $\mathcal{Z} = (Z_j)_{1 \leq j \leq t}$ and $[y] = (y^1, y^2, \ldots, y^t)$. Recall that $[y''] = \mathcal{T}[y']$.

Definition 6.3. *(Typical gap sequence)* Let $a = (a_1, a_2, \ldots, a_n)$ and $b = (b_1, b_2, \ldots, b_n)$ be two integer sequences. Let a', b' be two integer sequences, initialized with $a' = a$ and $b' = b$. Repeat the following procedure until no update is made:

- if there exists $1 \leq i < n$ with $a'_i = a'_{i+1}$, replace a' by $(a'_1, a'_2, \ldots, a'_i, a'_{i+2}, \ldots, a'_n)$ and b' by $(b'_1, b'_2, \ldots, b'_{i-1}, b'_i + b'_{i+1} + 1, b'_{i+2}, \ldots b'_n)$,
- if there exists $1 \leq i < n - 1$ with $a'_i < a'_{i+1} < a'_{i+2}$ or $a'_i > a'_{i+1} > a'_{i+2}$, replace a' by $(a'_1, a'_2, \ldots, a'_i, a'_{i+2}, \ldots, a'_n)$ and b' by $(b'_1, b'_2, \ldots, b'_{i-1}, b'_i + b'_{i+1} + 1, b'_{i+2}, \ldots b'_n)$.

Let (a'', b'') be the value of (a', b') at the end of the iteration.

Define the *typical gap list* $\mathcal{TG}(a, b)$ of b for a to be b''.

Define the output of the procedure ComputeForgetGapList$(g(q))$ to be $g(p)$, where $g(p)$ is defined as follows:

- if $\mathcal{Z} \setminus \{v\}$ does not contain any consecutive duplicate bags, let $g(p) = g(q)$,
- if $\mathcal{Z} \setminus \{v\}$ contains exactly a pair of consecutive duplicate bags, say $Z_{j_1} \setminus \{v\} = Z_{j_1+1} \setminus \{v\}$, for some $1 \leq j_1 < t$, let $g(p) = (g(q)^1, g(q)^2, \ldots, g(q)^{j_1-1}, \mathcal{TG}(y^{j_1} y^{j_1+1}, g(q)^{j_1} g(q)^{j_1+1}), g(q)^{j_1+2}, \ldots, g(q)^t)$,
- if $\mathcal{Z} \setminus \{v\}$ contains two pairs of consecutive duplicate bags, but there are no three consecutive duplicate bags, say $Z_{j_1} \setminus \{v\} = Z_{j_1+1} \setminus \{v\}$ and $Z_{j_2} \setminus \{v\} = Z_{j_2+1} \setminus \{v\}$ for some $1 \leq j_1 < j_2 < t$, let $g(p) = (g(q)^1, g(q)^2, \ldots, g(q)^{j_1-1}, \mathcal{TG}(y^{j_1} y^{j_1+1}, g(q)^{j_1} g(q)^{j_1+1}), g(q)^{j_1+2}, \ldots, g(q)^{j_2-1}, \mathcal{TG}(y^{j_2} y^{j_2+1}, g(q)^{j_2} g(q)^{j_2+1}), g(q)^{j_2+2}, \ldots, g(q)^t)$
- if $\mathcal{Z} \setminus \{v\}$ contains three consecutive bags, say $Z_{j_1} \setminus \{v\} = Z_{j_1+1} \setminus \{v\} = Z_{j_1+2} \setminus \{v\}$ for some $1 \leq j_1 \leq t$, let $g(p) = (g(q)^1, g(q)^2, \ldots, g(q)^{j_1-1}, \mathcal{TG}(y^{j_1} y^{j_1+1} y^{j_1+2}, g(q)^{j_1} g(q)^{j_1+1} g(q)^{j_1+2}), g(q)^{j_1+3}, \ldots, g(q)^t)$.

We now show that the output of ComputeForgetGapList$(g(q))$ is indeed the gap list $g(p)$. By construction, $\mathcal{CP}(p) = \mathcal{CP}(q)$. Hence, if $[y'] = [y]$, $\mathcal{Z} \setminus \{v\}$ contains no duplicate bags and hence $g(p) = g(q)$. Otherwise, for each integer $(y')^j_i$ of $[y']$ that is eliminated, the number of bags between the bags corresponding to the two newly adjacent integers is the sum of the number of bags that were initially between each of them and the bag of cardinality $(y')^j_i$ plus one, to account for the bag of cardinality $(y')^j_i$. Hence, the procedure has the desired output.

It remains to show that ComputeForgetGapList$(g(q))$ is a logspace computable function. This follows since $g(p)$ and $g(q)$ can be stored in logspace and $\mathcal{C}^*_p$ and $\mathcal{C}^*_q$ can be computed in logspace (Theorem 5.17) and stored in logspace (Lemma 5.13). Moreover, similarly to the proof of Lemma 5.7, the typical gap list can be computed in logspace.

6.1.3 Introduce Node

Let p be an introduce node and q be its child, with $X_p \setminus X_q = \{v\}$. Let $\mathcal{C}^*_p = (\mathcal{Z}', [y'])$, $\mathcal{C}^*_q = (\mathcal{Z}, [y])$ be the unique characteristics at p, q, resp, with $\mathcal{Z} = (Z_j)_{1 \leq j \leq t}, [y] = (y^1, y^2, \ldots, y^t)$ and $\mathcal{Z}' = (Z'_j)_{1 \leq j \leq t}, [y'] = (\ (y')^1, (y')^2, \ldots, (y')^t\ )$.

Definition 6.4. *(Gap split)* Let $a = (a_1, a_2, \ldots, a_n)$ and $b = (b_1, b_2, \ldots b_n)$ be two integer sequences. For any split (a^1, a^2) of a, define (b^1, b^2) as follows:

- if $(a^1, a^2) = ((a_1, a_2, \ldots, a_i), (a_{i+1}, a_{i+1}, \ldots, a_n))$, let $((b_1, b_2, \ldots, b_{i-1}, b_i), (b_{i+1}, b_{i+2}, \ldots, b_n))$
- if $(a^1, a^2) = ((a_1, a_2, \ldots, a_i), (a_i, a_{i+1}, \ldots, a_n))$, let $((b_1, b_2, \ldots, b_{i-1}, 0), (b_i, b_{i+1}, \ldots, b_n))$

Define the *gap split* of b for (a^1, a^2) to be (b^1, b^2).

Define the output of the procedure ComputeIntroduceGapList$(g(q))$ to be $g(p)$, where $g(p)$ is defined as follows:

- if $t' = t$, let $g(p) = g(q)$,
- if $t' = t + 1$ and $Z'_l \setminus \{v\} = Z'_{l-1}$ for some l, let $g(p) = (g(q)^1, g(q)^2, \ldots, g(q)^{l-2}, g(p)^{l-1}, g(p)^l, g(q)^l, \ldots, g(q)^t)$, where $(g(p)^{l-1}, g(p)^l)$ is the gap split of $g(q)^{l-1}$ for $((y')^{l-1}, (y')^l)$.
- if $t' = t+1$ and $Z'_r \setminus \{v\} = Z'_{r+1}$ for some r, let $g(p) = (g(q)^1, g(q)^2, \ldots, g(q)^{r-1}, g(p)^r, g(p)^{r+1}, g(q)^{r+1}, \ldots, g(q)^t)$, where $(g(p)^r, g(p)^{r+1})$ is the gap split of $g(q)^r$ for $((y')^r, (y')^{r+1})$.

- if $t' = t+2$ and $Z'_l \setminus \{v\} = Z'_{l-1}$ and $Z'_{r+1} \setminus \{v\} = Z'_r$ for some $l < r$, let $g(p) = (g(q)^1, g(q)^2, \ldots, g(q)^{l-2}, g(p)^{l-1}, g(p)^l, g(q)^l, \ldots, g(q)^{r-2}, g(p)^r, g(p)^{r+1}, g(q)^r, \ldots, g(q)^t)$, where $(g(p)^{l-1}, g(p)^l)$ is the gap split of $g(q)^{l-1}$ for $((y')^{l-1}, (y')^l)$ and $(g(p)^r, g(p)^{r+1})$ is the gap split of $g(q)^{r-1}$ for $((y')^r, (y')^{r+1})$.
- if $t' = t+2$ bags and $Z'_{l-1} = Z'_l \setminus \{v\} = Z'_{l+1}$, let $g(p) = (g(q)^1, g(q)^2, \ldots, g(q)^{l-2}, g(p)^{l-1}, g(p)^l, g(p)^{l+1}, g(q)^l, \ldots, g(q)^t)$, where $(g(p)^{l-1}, g'')$ is the gap split of $g(q)^{l-1}$ for $((y')^{l-1}, y'')$ and $(g(p)^l, g(p)^{l+1})$ is the gap split of g'' for $((y')^l, (y')^{l+1})$.

We now show that the output of ComputeIntroduceGapList$(g(q))$ is indeed the gap list $g(p)$. By construction, $\mathcal{CP}(p)$ is obtained by choosing a sequence of consecutive bags in $\mathcal{CP}(q)$ and adding $\{v\} = X_p \setminus X_q$ to all the bags in this sequence. Before doing this, the first bag of this sequence can be duplicated, in which case v is not added to the first copy, but is added to the second copy. Similarly, the last bag can be duplicated, in which case v is added to the first copy, but not to the second copy. In this new sequence of bags, there are no bags between the the two copies of the same bag and the number of bags between any other two bags is the same as before the transformation. Hence, the procedure has the desired output.

It remains to show that ComputeIntroduceGapList$(g(q))$ is a logspace computable function. This follows since $g(p)$ and $g(q)$ can be stored in logspace and $\mathcal{C}^*_p, \mathcal{C}^*_q$ can be computed in logspace (Theorem 5.17) and stored in logspace (Lemma 5.13). Moreover, the gap split can be computed in logspace.

6.1.4 Join Node

Let p be a join node and q, r be its children. Let $\mathcal{C}^*_p = (\mathcal{Z}, [y'_p]), \mathcal{C}^*_q = (\mathcal{Z}, [y_q]), \mathcal{C}^*_r = (\mathcal{Z}, [y_r])$ be the unique characteristics at p, q, r, resp, with $\mathcal{Z} = (Z_j)_{1 \le j \le t}$ and $[y_p] = (y^1_p, y^2_p, \ldots y^t_p), [y_q] = (y^1_q, y^2_q, \ldots y^t_q), [y_r] = (y^1_r, y^2_r, \ldots, y^t_r)$. Recall that $[y'_p] = \mathcal{T}[y_p]$ and for $1 \le j \le t, y^j_p = e^j_q + e^j_r - |Z_j|$, for some $e^j_q \in \mathcal{E}(y^j_q), e^j_r \in \mathcal{E}(y^j_r)$.

Definition 6.5. *(Gap extension)* Let $a = (a_1, a_2, \ldots, a_n)$ and $b = (b_1, b_2, \ldots, b_n)$ be two integer sequences. Let $a^* \in \mathcal{E}(a)$ be an extension of a. There exist $1 = t_1 < t_2 < \ldots < t_{n+1}$ such that for $1 \le i \le n$ and $t_i \le j < t_{i+1}$, $a^*_j = a_i$. Define b^* such that for $1 \le i \le n$ and $t_i \le j < t_{i+1} - 1$, $b^*_j = 0$ and $b^*_{t_{i+1}-1} = b_i$. Define the *gap extension* of b for a^* to be b^*.

Define the output of ComputeJoinGapList$(g(q))$ to be $ge_q = (ge^1_q, ge^2_q, \ldots, ge^t_q)$, where for $1 \le j \le t$, ge^j_q is the gap extension of $g(q)^j$ for e^j_q. Define the output of ComputeJoinGapList$(g(r))$ similarly. Define $ge_p = ge_q + ge_r$. Now define the output of the procedure ComputeJoinGapList $(g(q), g(r))$ to be $\mathcal{TG}(y_p, ge_p) = (\mathcal{TG}(y^1_p, ge^1_p), \mathcal{TG}(y^2_p, ge^2_p), \ldots, \mathcal{TG}(y^t_p, ge^t_p))$, where for $1 \le j \le t$, $\mathcal{TG}(y^j_p, ge^j_p)$ is the typical gap list of ge^j_p for y^j_p.

We now show that the output of the procedure ComputeJoinGapList$(g(q), g(r))$ is the gap list $g(p)$. By construction, $\mathcal{CP}(p)$ is the union of some extension Y_q of $\mathcal{CP}(q)$ and some extension Y_r of $\mathcal{CP}(r)$. For each integer that is duplicated in $[y_q]$, its corresponding bag in $\mathcal{CP}(q)$ is duplicated. Call Y'_q the resulting extension of $\mathcal{CP}(q)$ and similarly define Y'_r. There are no bags between duplicates of the same bag and the number of bags between the bags corresponding to any other consecutive integers in e_q is the same as the number of bags between their corresponding bags in y_q. So the output of ComputeJoinGapList$(g(q))$ is the gap list of Y'_q for the list e_q.

For any two consecutive integers $(e_q^j)_i$ and $(e_q^j)_{i+1}$ in e_q, there are $(ge_q^j)_i$ bags between their corresponding bags in Y'_q. Moreover, all these bags have cardinality $\max((e_q^j)_i, (e_q^j)_{i+1})$. Similarly, for any two consecutive integers $(e_r^j)_i$ and $(e_r^j)_{i+1}$ of e_r, there are $(ge_r^j)_i$ bags of cardinality $\max((e_r^j)_i, (e_r^j)_{i+1})$ between their corresponding bags in Y'_r. The bag in $\mathcal{CP}(p)$ of cardinality $(e_q^j)_i + (e_r^j)_i - |Z_j|$ is the union of the bag of cardinality $(e_q^j)_i$ in Y'_q and the bag of cardinality $(e_r^j)_i$ in Y'_r. Similarly for the bag of cardinality $(e_q^j)_{i+1} + (e_r^j)_{i+1} - |Z_j|$. Hence the bags in $\mathcal{CP}(p)$ between the bags corresponding to $(e_q^j)_i + (e_r^j)_i - |Z_j|$ and $(e_q^j)_{i+1} + (e_r^j)_{i+1} - |Z_j|$ are the union of an extension of the subpath in Y'_q between the bags corresponding to $(e_q^j)_i$ and $(e_q^j)_{i+1}$ and an extension of the subpath in Y'_r between the bags corresponding to $(e_r^j)_i$ and $(e_r^j)_{i+1}$. If $(e_q^j)_i > (e_q^j)_{i+1}$ and $(e_r^j)_i < (e_r^j)_{i+1}$, we cannot simply take the union of the bags in the subpaths in Y'_q and Y'_r, as the bag of their union has cardinality $(e_q^j)_i + (e_r^j)_{i+1} - |Z_j|$ and we do not know whether this is $\leq k+1$. However, we know that $(e_q^j)_i + (e_r^j)_i - |Z_j| \leq k+1$ and $(e_q^j)_{i+1} + (e_r^j)_{i+1} - |Z_j| \leq k+1$. So we duplicate $(ge_q^j)_i$ times the bag in Y'_r corresponding to $(e_r^j)_i$ and we duplicate $(ge_r^j)_i$ times the bag in Y'_q corresponding to $(e_q^j)_{i+1}$ and then set the subpath in $\mathcal{CP}(p)$ between the bags corresponding to $(e_q^j)_i + (e_r^j)_i - |Z_j|$ and $(e_q^j)_{i+1} + (e_r^j)_{i+1} - |Z_j|$ to be the union of these resulting subpaths. For consistency, we assume that for any $(e_q^j)_i$ and $(e_q^j)_{i+1}$, the subpath in Y_q between the bags corresponding to $(e_q^j)_i$ and $(e_q^j)_{i+1}$ is obtained by duplicating $(ge_r^j)_i$ times the bag corresponding to $\min((e_q^j)_i, (e_q^j)_{i+1})$ in the subpath in Y'_q between the same bags. Moreover, the subpath in Y_q between the bags corresponding to $(e_q^j)_{l(e_q^j)}$ and $(e_q^{j+1})_1$ is obtained by duplicating $(ge_r^j)_{l(e_r^j)}$ times the bag corresponding to $(e_q^j)_{l(e_q^j)}$ in the subpath in Y'_q between the same bags. Similarly for the subpats in Y_r. Therefore, ge_p is indeed the gap list for to the list y_p.

For each integer of y'_p that is eliminated, the number of bags between the bags corresponding to the two newly consecutive integers is the sum of the number of bags that were initially betweeen each of them and the bag corresponding to the integer eliminated, plus one, to acount for the bag corresponding to the integer eleminated. Therefore, $\mathcal{TG}(y_p, ge_p)$ is indeed $g(p)$.

It remains to show that ComputeJoinGapList$(g(q), g(r))$ is a logspace computable function. This follows since $g(p), g(q), g(r)$ can be stored in logspace, $\mathcal{C}_p^*, \mathcal{C}_q^*, \mathcal{C}_r^*$ can be computed in logspace (Theorem 5.17) and stored in logspace (Lemma 5.13) and the extensions e_q and e_r can be computed in logspace. Moreover, the gap extensions and the typical gap extensions can be computed and stored in logspace.

6.2 Computing the Gap List at the Root

In this section, we see how the logspace algorithms of the previous section can be used to obtain a logspace algorithm for computing $g(root)$. This follows from the fact that $g(root)$ is the sum of lists, each computable only from the nodes on a path from a leaf to the root.

Definition 6.6. *(Additive typical gap sequence)* Let $a = (a_1, a_2, \ldots, a_n)$ and $b = (b_1, b_2, \ldots, b_n)$ be two integer sequences. Let a', b' be two integer sequences, initialized with $a' = a$ and $b' = b$. Repeat the following procedure until no update is made:

- if there exists $1 \leq i < n$ with $a'_i = a'_{i+1}$, replace a' by $(a'_1, a'_2, \ldots, a'_i, a'_{i+2}, \ldots, a'_n)$ and b' by $(b'_1, b'_2, \ldots, b'_{i-1}, b'_i + b'_{i+1}, b'_{i+2}, \ldots b'_n)$,
- otherwise, if there exists $1 \leq i < n-1$ with $a'_i < a'_{i+1} < a'_{i+2}$ or $a'_i > a'_{i+1} > a'_{i+2}$, replace a' by $(a'_1, a'_2, \ldots, a'_i, a'_{i+2}, \ldots, a'_n)$ and b' by $(b'_1, b'_2, \ldots, b'_{i-1}, b'_i + b'_{i+1}, b'_{i+2}, \ldots b'_n)$.

Let (a'', b'') be the value of (a', b') at the end of the iteration.

Define the *additive typical gap list* $\mathcal{TG}'(a, b)$ of b for a to be b''.

For any forget node $p \in I$ with child q, define the procedure ComputeForgetGapList'$(g(q))$, whose output is computed in the same way as the output of the procedure ComputeForgetGapList$(g(q))$, except that all occurrences of $\mathcal{TG}$ are replaced by $\mathcal{TG}'$. Similarly, for any join node $p \in I$ with children q and r, define the procedure ComputeJoinGapList'$(g(q), g(r))$ whose output is computed in the same way as the output the procedure ComputeJoinGapList$(g(q), g(r))$, except that all occurrences of $\mathcal{TG}$ are replaced by $\mathcal{TG}'$.

Let $leaf$ be a leaf node and let $leaf = p_0, p_1, \ldots, p_d = root$ be the nodes in order on the path in T from $leaf$ to $root$. For $0 \leq i < d$, define the procedure ComputeGapList$(leaf, g(p_i))$ to be one of the following:

- ComputeForgetGapList$(g(p_i))$, if p_{i+1} is a forget node and $leaf$ is the lexicographically smallest leaf node in the subtree of T rooted at p_{i+1}.
- ComputeForgetGapList'$(g(p_i))$, if p_{i+1} is a forget node and $leaf$ is not the lexicographically smallest leaf node in the subtree of T rooted at p_{i+1}.
- ComputeIntroduceGapList$(g(p_i))$, if p_{i+1} is an introduce node.
- ComputeJoinGapList$(g(p_i))$, if p_{i+1} is a join node and $leaf$ is the lexicographically smallest leaf node in the subtree of T rooted at p_{i+1}.
- ComputeJoinGapList'$(g(p_i))$, if p_{i+1} is a join node and $leaf$ is not the lexicographically smallest leaf node in the subtree of T rooted at p_{i+1}.

Define ComputeGapList$^t(leaf)$ recursively, where ComputeGapList$^1(leaf)$ is Compute GapList$(leaf, g(leaf))$ and ComputeGapList$^t(leaf)$ is ComputeGapList$(leaf$,ComputeGap List$^{t-1}(leaf))$.

Define $\mathcal{L}(p)$ to be the set of leaves in the subtree of T rooted at p and define $dist(p, leaf)$ to be the number of edges in T on the path from the leaf node $leaf$ to a node p.

Theorem 6.7. The gap list $g(root)$ is equal to $\sum_{leaf \in \mathcal{L}(root)}$ComputeGapList$^{dist(root,leaf)}(leaf)$.

Proof. We prove by induction on d, that for all nodes p at depth d, we have $g(p) = \sum_{\in \mathcal{L}(p)}$Com puteGapList$^{dist(p,leaf)}(leaf)$. This clearly holds for all leaves p. Suppose it holds for all nodes at depth greater than d and prove that it holds for all nodes at depth d.

If p is a forget node, let q be its child and $leaf'$ be the lexicographically smallest leaf node. Using the induction hypothesis, the definition of ComputeForgetGapList and the fact that ComputeForgetGapList' is an additive function, we obtain:

$$
\begin{aligned}
g(p) &= \textsc{ComputeForgetGapList}(g(q)) \\
&= \textsc{ComputeForgetGapList}\Big(\sum_{leaf \in \mathcal{L}(q)} \textsc{ComputeGapList}^{dist(q,leaf)}(leaf)\Big) \\
&= \textsc{ComputeForgetGapList}(\textsc{ComputeGapList}^{dist(q,leaf')}(leaf')) \\
&+ \sum_{leaf \in \mathcal{L}(q) \setminus \{leaf'\}} \textsc{ComputeForgetGapList'}(\textsc{ComputeGapList}^{dist(q,leaf)}(leaf)) \\
&= \sum_{leaf \in \mathcal{L}(p)} \textsc{ComputeGapList}^{dist(p,leaf)}(leaf)
\end{aligned}
$$

If p is an introduce node, let q be its child. Using the induction hypothesis and the fact that by definition, ComputeIntroduceGapList is an additive function, we obtain:

$$\begin{aligned} g(p) &= \textsc{ComputeIntroduceGapList}(g(q)) \\ &= \textsc{ComputeIntroduceGapList}(\sum_{leaf \in \mathcal{L}(q)} \textsc{ComputeGapList}^{dist(q,leaf)}(leaf)) \\ &= \sum_{leaf \in \mathcal{L}(q)} \textsc{ComputeIntroduceGapList}(\textsc{ComputeGapList}^{dist(q,leaf)}(leaf)) \\ &= \sum_{leaf \in \mathcal{L}(p)} \textsc{ComputeGapList}^{dist(p,leaf)}(leaf) \end{aligned}$$

If p is a join node, let q, r be its children and $leaf'$ be the lexicographically smallest leaf node. Using the induction hypothesis, the definition of ComputeJoinGapList and the fact that by definition, ComputeJoinGapList' is an additive function, we obtain:

$$\begin{aligned} g(p) &= \textsc{ComputeJoinGapList}(g(q), g(r)) \\ &= \textsc{ComputeJoinGapList}(\sum_{leaf \in \mathcal{L}(q)} \textsc{ComputeGapList}^{dist(q,leaf)}(leaf) \\ &\quad , \sum_{leaf \in \mathcal{L}(r)} \textsc{ComputeGapList}^{dist(r,leaf)}(leaf)) \\ &= \textsc{ComputeJoinGapList}(\textsc{ComputeGapList}^{dist(p,leaf')-1}(leaf')) \\ &\quad + \textsc{ComputeJoinGapList'}(\sum_{leaf \in \mathcal{L}(p) \setminus \{leaf'\}} \textsc{ComputeGapList}^{dist(p,leaf)-1}(leaf)) \\ &= \textsc{ComputeJoinGapList}(\textsc{ComputeGapList}^{dist(p,leaf')-1}(leaf')) \\ &\quad + \sum_{leaf \in \mathcal{L}(p) \setminus \{leaf'\}} \textsc{ComputeJoinGapList'}(\textsc{ComputeGapList}^{dist(p,leaf)-1}(leaf)) \\ &= \textsc{ComputeGapList}^{dist(p,leaf')}(leaf') + \sum_{leaf \in \mathcal{L}(p) \setminus \{leaf'\}} \textsc{ComputeGapList}^{dist(p,leaf)}(leaf) \\ &= \sum_{leaf \in \mathcal{L}(p)} \textsc{ComputeGapList}^{dist(p,leaf)}(leaf) \end{aligned}$$

□

Theorem 6.8. The gap list $g(root)$ can be computed in logspace.

Proof. Iterate the nodes of T, while storing a variable sum. Initialize $sum = 0$. If the current node p is a leaf node, add to the current value of sum the output of $\textsc{ComputeGapList}^{dist(root,p)}(p)$. By Theorem 6.7, the value of sum at the end of iteration will be $g(root)$. Since the composition of logspace functions is a logspace function and ComputeGapList is a logspace procedure, $g(root)$ can be computed in logspace. □

7 Computing Endpoints

Note that each path decomposition is completely determined by specifying for each vertex the indices of the bags it belongs to. By property PD-3, these bags are consecutive, so each path decomposition is also completely determined by specifying for each vertex v the indices of the first and last bags containing v. We shall call the indices of the first and last bag containing v, the *left* and *right endpoints* of v.

In this section, we see how to compute in logspace the left endpoint of a given vertex v at a node p, given the left endpoint of v at the children of p. We then give a logsapce algorithm for computing the left endpoint of v at the root of T. By symmetry, we can also determine in logspace the right endpoint of v.

Let $v \in V(G), p \in I$ and $\mathcal{CP}(p) = (Y_1, Y_2, \ldots Y_s)$ be the characteristic path decomposition at p (see Definition 6.1). Recall that G_p is the subgraph of G rooted at p (see Definition 3.3).

Definition 7.1. *(Endpoints)* Define the *left endpoint* $l(p, v)$ and the *right endpoint* $r(p, v)$ of v at p as follows:

- if $v \notin G_p$, then $l(p, v) = r(p, v) = null$,
- if $v \in G_p$, then $l(p, v)$ is the smallest index i such that $v \in Y_i$,
- if $v \in G_p$, then $r(p, v)$ is the largest index i such that $v \in Y_i$.

Since $l(p, v) \leq s$ and $r(p, v) \leq s$ and $\mathcal{CP}(p)$ is a minimal partial path decomposition, $l(p, v)$ and $r(p, v)$ can be stored in logspace.

7.1 Computing the Endpoints from the Endpoints at Children

In this section, we see how to compute in logspace $l(p, v)$, given the left endpoints of v in the characteristic path decompositions at the children of p.

7.1.1 Start Node

Let p be a start node and let $\mathcal{C}_p^*$ be the unique characteristic at p. If $X_p \neq \{v\}$, let $l(p, v) = null$. Otherwise, from the algorithm computing $FS(p)$, we observe that:

- if $\mathcal{C}_p^* = (\ (\{v\}),\ [(1)]\ )$, then $l(p, v) = 1$,
- if $\mathcal{C}_p^* = (\ (\{v\}, \{\emptyset\}),\ [(1), (0)]\ )$, then $l(p, v) = 1$
- if $\mathcal{C}_p^* = (\ (\{\emptyset\}, \{v\}),\ [(0), (1)]\ )$, then $l(p, v) = 2$
- if $\mathcal{C}_p^* = (\ (\{\emptyset\}, \{v\}, \{\emptyset\}),\ [(0), (1), (0)]\ )\}$, then $l(p, v) = 2$

Define the procedure ComputeStartLeftEndpoint(p) to compute $l(p, v)$. Since at least one of the above cases holds, ComputeStartLeftEndpoint(p) is a logspace computable function.

7.1.2 Forget Node

Let p be a forget node and q be its child. By the proof of the algorithm computing the characteristic at p given the characteristic at q, we see that $\mathcal{CP}(q) = \mathcal{CP}(p)$. Therefore, $l(p, v) = l(q, v)$. Define the procedure ComputeForgetLeftEndpoint$(l(q, v))$ to compute $l(p, v)$ given $l(q, v)$. From the above, it is easy to see that ComputeForgetLeftEndpoint$(l(q, v))$ is a logspace computable function.

7.1.3 Introduce Node

Let p be an introduce node and q be its child. Let $\mathcal{C}_p^* = (\mathcal{Z}', [y']), \mathcal{C}_q^* = (\mathcal{Z}, [y])$ be the unique characteristics at p, q, resp., with $\mathcal{Z} = (Z_j)_{1 \leq j \leq t}$. Let $X_p \setminus X_q = \{v'\}$.

We first define an auxiliary data structure, the *typical indices list*. Each integer in the typical list of $\mathcal{C}_p^*$ is the cardinality of some bag in $\mathcal{CP}(p)$. The typical indices list is the list of the indices of these bags.

Definition 7.2. *(Typical indices list)* For $1 \leq j \leq t$ and $1 \leq i \leq l(g(p)^j)$, define

$$ix(p)_i^j = \sum_{j'=1}^{j-1} \sum_{i'=1}^{l(g(p)^{j'})} (g(p)_{i'}^{j'} + 1) + \sum_{i'=1}^{i-1} (g(p)_{i'}^{j} + 1) + 1$$

Define $ix(p) = (ix(p)^1, ix(p)^2, \ldots, ix(p)^t)$ to be the *typical indices list* at p.

Lemma 7.3. The typical indices list $ix(p)$ can be computed and stored in logspace.

Proof. Since each of the integers in the typical indices list is at most s and s can be stored in logspace, each of the elements of $ix(p)$ can be stored in logspace. Moreover, $l[ix(p)] = l[g(p)] = l[y']$ and hence by Lemma 5.4, $l[ix(p)]$ is at most a constant depending on k. Therefore, $ix(p)$ can be stored in logspace.

Moreover, by applying Theorem 6.8 to the graph G_p, we can compute $g(p)$ in logspace. We can iterate the elements of $g(p)$ to compute each element of $ix(p)$. □

Define the procedure ComputeIntroduceLeftEndpoint$(l(q, v))$ to output $l(p, v)$, where $l(p, v)$ is defined as follows:

- if $v \notin V(G_p)$, then $l(p, v) = null$,
- if $v \in V(G_p)$, but $v \notin V(G_q)$, then $v = v'$ and v belongs to at least one of the bags of the interval model. Let l be minimal index such that $v \in Z_l$. Let $l(p, v) = ix(p)_1^l$,
- if $v \in V(G_q)$, then
  - if $l[y'] = l[y]$, then $l(p, v) = l(q, v)$,
  - if $l[y'] = l[y] + 1$ and $Z_l' \setminus \{v'\} = Z_{l-1}'$ for some l. Then $((y')^{l-1}, (y')^l)$ is a split of type II of y^{l-1} at some $1 \leq l' \leq l(y^{l-1})$.
    - if $l(q, v) \leq ix(q)_{l'}^{l-1}$, then $l(p, v) = l(q, v)$,
    - if $l(q, v) > ix(q)_{l'}^{l-1}$, then $l(p, v) = l(q, v) + 1$.

- if $l[y'] = l[y] + 1$ and $Z'_r \setminus \{v'\} = Z'_{r+1}$ for some r. Then $((y')^r, (y')^{r+1})$ is a split of type II of y^r at some $1 \leq r' \leq l(y^r)$.
  - if $l(q,v) \leq ix(q)^r_{r'}$, then $l(p,v) = l(q,v)$,
  - if $l(q,v) > ix(q)^r_{r'}$, then $l(p,v) = l(q,v) + 1$.
- if $l[y'] = l[y] + 2$ and $Z'_{l-1} = Z_l \setminus \{v'\}$ and $Z'_{r+1} \setminus \{v'\} = Z'_r$ for some $l < r$, there exist l, r such that $((y')^{l-1}, (y')^l)$ is a split of type II of y^{l-1} at some $1 \leq l' \leq l(y^l)$ and $((y')^r, (y')^{r+1})$ is a split of type II of y^{r-1} at some $1 \leq r' \leq l(y^r)$.
  - if $l(q,v) \leq ix(q)^{l-1}_{l'}$, then $l(p,v) = l(q,v)$,
  - if $ix(q)^{l-1}_{l'} < l(q,v) \leq ix(q)^{r-1}_{r'}$, then $l(p,v) = l(q,v) + 1$,
  - if $l(q,v) > ix(q)^{r-1}_{r'}$, then $l(p,v) = l(q,v) + 2$.
- if $l[y'] = l[y] + 2$ and $Z'_{l-1} = Z_l \setminus \{v'\} = Z'_{l+1}$ for some l. Then $((y')^{l-1}, y'')$ is a split of type II of y^{l-1} at some $1 \leq l' \leq l(y^{l-1})$ and $((y')^l, (y')^{l+1})$ is a split of type II of y'' at some $1 \leq l'' \leq l(y'')$.
  - if $l(q,v) \leq ix(q)^{l-1}_{l'}$, then $l(p,v) = l(q,v)$,
  - if $ix(q)^{l-1}_{l'} < l(q,v) \leq ix(q)^{l-1}_{l''}$, then $l(p,v) = l(q,v) + 1$,
  - if $l(q,v) > ix(q)^{l-1}_{l''}$, then $l(p,v) = l(q,v) + 2$.

We now show that ComputeIntroduceLeftEndpoint$(l(q,v))$ correctly outputs the left endpoint $l(p,v)$. One of the following cases holds:

- if $v \notin V(G_p)$, then the output of the procedure is clearly correct.
- if $v \in V(G_p)$, but $v \notin V(G_q)$, then $v = v'$. Let l be the minimal such that $v \in Z_l$. Then v does not belong to any of the bags corresponding to the integers of any of the seqeunces $(y')^1, (y')^2, \ldots, (y')^{l-1}$, but it does belong to all the bags corresponding to the integers of the sequence $(y')^l$. Hence the first bag it belongs to is the one corresponding to $(y')^l_1$.
- if $v \in G_q$, then by construction, $\mathcal{CP}(p)$ is obtained by choosing a sequence of consecutive bags in $\mathcal{CP}(q)$ and adding the introduced node to all these bags. Before doing this, the first and last bags of the chosen sequence can be duplicated. Whenever a bag is duplicated, all the bags following the duplicate are shifted to the right by one. So each duplication of a bag appearing before the the first bag containing v adds one to the initial left endpoint of v.

It remains to show that ComputeIntroduceLeftEndpoint$(l(q,v))$ is a logspace computable function. The procedure computes $ix(q)$ and then compares $l(q,v)$ with an integer of $ix(q)$. Since $l(q,v)$ can be stored in logspace, $\mathcal{C}^*_p, \mathcal{C}^*_q$ can be computed and stored in logspace and $ix(q)$ can be computed and stored in logspace, this is a logspace procedure.

7.1.4 Join Node

Let p be a join node with children q and r. Let $\mathcal{C}^*_p = (\mathcal{Z}, [y'_p]), \mathcal{C}^*_q = (\mathcal{Z}, [y_q]), \mathcal{C}^*_r = (\mathcal{Z}, [y_r])$ be the unique characteristcs at p, q, r, with $\mathcal{Z} = (Z_j)_{1 \leq j \leq t}$ and $[y_p] = (y^1_p, y^2_p, \ldots y^t_p), [y_q] = (y^1_q, y^2_q, \ldots y^t_q), [y_r] = (y^1_r, y^2_r, \ldots, y^t_r)$. Recall that $[y'_p] = \mathcal{T}[y_p]$ and for $1 \leq j \leq t, y^j_p = e^j_q + e^j_r - |Z_j|$, for some $e^j_q \in \mathcal{E}(y^j_q), e^j_r \in \mathcal{E}(y^j_r)$. For $1 \leq j \leq t$, let ge^j_q and ge^j_r be the gap extension of $g(q)^j$ for e^j_q and the gap extension of $g(r)^j$ for e^j_r.

For any $1 \leq j \leq t$, there exist $1 = t_1^j < t_2^j < \ldots < t^j_{l(y_q^j)+1}$ such that for $1 \leq i \leq l(y_q^j)$, $t_i^j \leq s < t_{i+1}^j$, $(e_q^j)_s = (y_q^j)_i$. Let j_q be minimal such that $ix(q)_1^{j_q+1} > l(q,v)$ and define $ix(q)^{j_q}_{l(ix(q)^{j_q})+1} = ix(q)_1^{j_q+1}$. Let i_q be minimal such that $ix(q)^{j_q}_{i_q+1} > l(q,v)$. Define $le'_q = l(q,v) + \sum_{j=1}^{j_q-1}(t^j_{l(y_q^j)+1} - 1 - l(y_q^j)) + t^{j_q}_{i_q} - i_q + \sum_{j=1}^{j_q-1}\sum_{i=1}^{l(eb_q^j)}(eb_r^j)_i + \sum_{i=1}^{i_q-1}(eb_r^{j_q})_i$. If $(y_q^{j_q})_{i_q} < (y_q^{j_q})_{i_q+1}$ or $i_q = l(y_q^{j_q})$, define $le_q = le'_q + (eb_r^{j_q})_{i_q}$ and otherwise, define $le_q = le'_q$.

Define the procedure ComputeJoinLeftEndpoint($l(q,v)$) to output *null* if $v \notin V(G_p)$ and le_q, otherwise. Similarly, define the procedure ComputeJoinLeftEndpoint($l(r,v)$). Define the procedure ComputeJoinLeftEndpoint($l(q,v), l(r,v)$) to output min(ComputeJoinLeftEndpoint($l(q,v)$), ComputeJoinLeftEndpoint($l(r,v)$)).

We now show that ComputeJoinLeftEndpoint($l(q,v), l(r,v)$) correctly outputs $l(p,v)$. By construction, $\mathcal{CP}(p)$ is the union of some extension Y_q of $\mathcal{CP}(q)$ and some extension Y_r of $\mathcal{CP}(r)$. Hence, $l(p,v)$ will be the minimum of the indices of the first bag containing v in Y_q and Y_r. So the output of ComputeJoinLeftEndpoint($l(q,v), l(r,v)$) is correct, provided that the outputs of ComputeJoinLeftEndpoint($l(q,v)$) and ComputeJoinLeftEndpoint($l(r,v)$) are correct.

For each integer that is duplicated in $[y_q]$, its corresponding bag in $\mathcal{CP}(q)$ is duplicated. Call Y'_q the resulting path decomposition. Define Y'_r similarly. So each bag duplication in $\mathcal{CP}(q)$ that occurs before the first bag containing v adds one to the left endpoint of v in the new path decomposition. This accounts for the second and third terms in the definition of le'_q.

As in the definition of the gap list at a join node, we see that the number of bags in Y_q between any two bags corresponding to consecutive integers in e_q is the sum the number of bags in Y'_q between their corresponding bags in Y'_q and the number of bags in Y'_r between the bags in Y'_r corresponding to the integers at the same indices in e_r. We see that the sum of integers in ge_q is included in $l(q,v)$ and hence we only need to add to the left endpoint the sum of integers in ge_r. This accounts for the last two terms in the definition of le'_q.

Now if $(y_q^{j_q})_{i_q} < (y_q^{j_q})_{i_q+1}$ or $i_q = l(y_q^{j_q})$, the bag corresponding to $(y_q^{j_q})_{i_q}$ is first duplicated $(eb_r^{j_q})_{i_q}$ times in Y_q before adding the subpath in Y'_q between the bags corresponding to $(y_q^{j_q})_{i_q}$ and $(y_q^{j_q})_{i_q+1}$. Hence, the output of ComputeJoinLeftEndpoint($l(q,v)$) is correct.

It remains to show that ComputeJoinLeftEndpoint($l(q,v), l(r,v)$) is a logspace computable function. This follows since $l(p,v), l(q,v), l(r,v)$ can be stored in logspace, $\mathcal{C}_p^*, \mathcal{C}_q^*, \mathcal{C}_r*$ can be computed in logspace (Theorem 5.17) and stored in logspace (Lemma 5.13) and the extensions e_q and e_r can be computed and stored in logspace and $g(q)$ and $g(r)$ and their extensions can be computed and stored in logspace.

7.2 Computing the Endpoints at the Root

Let $v \in V(G)$. In this section, we see how the logspace functions of the previous sections can be used to compute $l(root, v)$ in logspace. This follows from the fact that $l(root, v)$ is the minimum of the left endpoints of v in the path decompositions corresponding just to the nodes on individual paths from a leaf to the root. The algorithm of this section is similar to the algorithm for computing $g(root)$.

For any node $p \in I$ with child q, define the procedure ComputeLeftEndpoint($l(q,v)$) to be ComputeForgetLeftEndpoint($l(q,v)$), ComputeIntroduceLeftEndpoint($l(q,v)$) or ComputeJoinLeftEndpoint($l(q,v)$) depending on the type of the node p. Denote by Compute

LeftEndpoint$^t(l(leaf, v))$ the output obtained by composing the procedure ComputeLeftEndpoint t times on the input $l(leaf, v)$. Let $\mathcal{L}(p)$ be the set of leaves in the subtree of T rooted at p and let $dist(p, leaf)$ the number of edges in T on the path from the leaf node $leaf$ to p.

Theorem 7.4. The left endpoint $l(root, v)$ is $\min_{leaf \in \mathcal{L}(root)}$ComputeLeftEndpoint$^{dist(root,leaf)}(l(leaf, v))$.

Proof. We prove by induction on d, that for all nodes p at depth d, we have $l(p, v) = \min_{leaf \in \mathcal{L}(p)}$ ComputeLeftEndpoint$^{dist(p,leaf)}(l(leaf, v))$. This clearly holds for all leaves p. We suppose it holds for all nodes at depth greater than d and prove that it holds for all nodes at depth d.

It is easy to see from the definitions of ComputeForgetLeftEndpoint$(l(q, v))$, ComputeIntroduceLeftEndpoint$(l(q, v))$, ComputeJoinLeftEndpoint$(l(q, v))$ and ComputeJoinLeftEndpoint$(l(r, v))$ that they are increasing functions. Observe that for any increasing function f and set S, we have $f(\min_{s \in S} s) = \min_{s \in S} f(s)$.

Now suppose p is a forget node at depth d; the other cases are treated similarly. Let q be its child. By the induction hypothesis and since ComputeForgetLeftEndpoint is a nondecreasing function, we obtain:

$$\begin{aligned}
l(p, v) &= \textsc{ComputeForgetLeftEndpoint}(l(q, v)) \\
&= \textsc{ComputeForgetLeftEndpoint} \\
&\quad \Big(\min_{leaf \in \mathcal{L}(q)} \textsc{ComputeLeftEndpoint}^{dist(q,leaf)}(l(leaf, v))\Big) \\
&= \min_{leaf \in \mathcal{L}(q)} \textsc{ComputeForgetLeftEndpoint} \\
&\quad \Big(\textsc{ComputeLeftEndpoint}^{dist(q,leaf)}(l(leaf, v))\Big) \\
&= \min_{leaf \in \mathcal{L}(p)} \textsc{ComputeLeftEndpoint}^{dist(p,leaf)}(l(leaf, v))
\end{aligned}$$

□

Theorem 7.5. The left endpoint $l(root, v)$ can be computed in logspace.

Proof. Iterate the nodes of T. For each leaf node $leaf$, we compute ComputeLeftEndpoint$^{dist(root,leaf)}(l(leaf, v))$. By Theorem 7.4, the minimum value taken over all leaf nodes is the left endpoint $l(root, v)$. Since logspace functions are closed under composition, ComputeLeftEndpoint is a logpspace procedure and hence $l(root, v)$ can be computed in logspace. □

8 L-completeness

In [EJT10], it is shown that for any constant $k \geq 1$, the language tree-width-$k = \{G \mid tw(G) \leq k\}$ is **L**-complete. We now show the corresponding result for pathwidth i.e., the language path-width-$k = \{G \mid pw(G) \leq k\}$ is **L**-complete.

As mentioned earlier, $G \in$ path-width-k if and only if the full set of characteristics at the root node of the tree of the nice tree decomposition of G is non-empty. This can be decided in logspace by using Theorem 5.15. Hence path-width-$k \in \mathbf{L}$. We now show that path-width-1 and path-width-k are **L**-hard by showing reductions from *Single Cycle Permutation problem*, a known **L**-complete problem [CM87]. It is easy to see that all our reductions can be performed in $\mathbf{NC^1}$.

Definition 8.1. *(Single Cycle Permutation problem)* The *Single Cycle Permutation problem* (SCP) is to decide if a given permutation, presented pointwise, consists of a single cycle.

Theorem 8.2. The language PATH-WIDTH-1 $= \{G \mid pw(G) \leq 1\}$ is **L**-complete.

Proof. We give a reduction from SCP to PATH-WIDTH-1. Given a permutation $\pi = (\pi(1), \pi(2), \ldots, \pi(n))$, we construct an undirected graph G with $V(G) = \{v_1, v_2, \ldots, v_n\}$ and $E(G) = \{(v_i, v_{\pi(i)}) \mid i \neq \pi(i), 1 \leq i \leq n\}$. Let $e \in E(G)$ be an arbitrary edge of G. Let G' be a graph with $V(G') = V(G)$ and $E(G') = E(G) \setminus \{e\}$.

- If $\pi \in$ SCP. Then G is a cycle, G' is a path and thus $pw(G') = 1$.
- If $\pi \notin$ SCP, G has at least two cycles and G' has at least one cycle. Hence $pw(G') \geq 2$.

Hence $\pi \in$ SCP iff $G' \in$ PATH-WIDTH-1 implying the **L**-hardness of PATH-WIDTH-1. □

Theorem 8.3. Let $k \geq 1$ be a constant. The language PATH-WIDTH-k $= \{G \mid pw(G) \leq k\}$ is L-complete.

Proof. We construct G and G' as done in Theorem 8.2. We use a construction from [EJT10] to construct G'' with $V(G'') = \{v_i^p \mid 1 \leq i \leq n, 1 \leq p \leq k\}$ and $E(G'') = \{(v_i^p, v_i^q) \mid 1 \leq i \leq n, 1 \leq p < q \leq k\} \bigcup \{(v_i^q, v_j^p) \mid (v_i, v_j) \in E(G'), 1 \leq i < j \leq n, 1 \leq p \leq q \leq k\}$.

If $\pi \in$ SCP, G' is a path. We may assume that $v_1, v_2, \ldots v_n$ are the vertices of the path, in that order. We will now define a width-k path decomposition for G''.

- For $1 \leq i \leq n$, let $X_i = \{v_i^1, v_i^2, \ldots v_i^k\}$.
- For $1 \leq i < n, 1 \leq p \leq k$, let $X_{i,i+1,p} = \{v_i^p, v_i^{p+1}, \ldots v_i^k, v_{i+1}^1, v_{i+1}^2, \ldots v_{i+1}^p\}$.
- Let $I = \{i \mid 1 \leq i \leq n\} \bigcup \{(i, i+1, p) \mid 1 \leq i < n, 1 \leq p \leq k\}$.
- Let $F = \{(\ (i, i+1, p), (i, i+1, p+1)\ ) \mid 1 \leq i < n, 1 \leq p < k\} \bigcup \{(\ i, (i, i+1, 1)\ ), (\ i+1, (i, i+1, k)\ ) \mid 1 \leq i < n\}$.
- Let $\mathcal{D} = (\{X_i \mid i \in I\}, T(I, F))$.

Note that T is a path. We now verify the three properties of a path decompostion (see Definition 1.2).

- PD-1: For any $v_i^p \in V(G'')$, we have $v_i^p \in X_i$.
- PD-2: For any $(v_i^p, v_i^q) \in E(G'')$, we have $v_i^p, v_i^q \in X_i$ and for any $(v_i^q, v_{i+1}^p) \in E(G'')$ with $1 \leq i < n$ and $1 \leq p \leq q \leq n$, we have $v_i^q, v_{i+1}^p \in X_{i,i+1,p}$.
- PD-3: For any $v_i^p \in V(G'')$ with $1 < i < n$, we see that v_i^p only belongs to the bags $X_{i-1,i,p}$, $X_{i-1,i,p+1}, \ldots X_{i-1,i,k}, X_i, X_{i,i+1,1}, X_{i,i+1,2}, \ldots, X_{i,i+1,p}$, which induce a subpath in T. Similarly for any $v_1^p \in V(G'')$, all the bags containing v_1^p induce a subpath in T and for any $v_n^p \in V(G'')$, all the bags containing v_n^p induce a subpath in T.

Note that the width of $\mathcal{D}$ is k implying $pw(G'') = k$.

If $\pi \notin$ SCP, then G' contains a cycle and G'' has a $(k+2)$-clique as a minor (see [EJT10] for details). Hence $tw(G'') > k$ implying $pw(G'') > k$.

Therefore, $\pi \in$ SCP iff $G'' \in$ PATH-WIDTH-k, implying the **L**-hardness of PATH-WIDTH-k. □

9 Conclusion and Open Problems

We presented a logspace algorithm to compute path decompositions of bounded pathwidth graphs. What is the complexity of Graph Isomorphism of bounded pathwidth graphs? Can we improve the **LogCFL** upper bound implied by the algorithm of Das, Toran and Wagner [DTW10]? Is there a logspace algorithm?

Bodlaender and Kloks [BK96] presented a polynomial time algorithm to compute path decompositions of *bounded treewidth* graphs. Since bounded treewidth graphs can have pathwidth as large as $\Omega(\log n)$, our techniques cannot be directly applied to achieve a logspace algorithm. Is there a logspace algorithm to compute path decompositions of *bounded treewidth* graphs?

References

[ADK08] Vikraman Arvind, Bireswar Das, and Johannes Köbler. A logspace algorithm for partial 2-tree canonization. In *CSR*, pages 40–51, 2008.

[BHZ87] Ravi B. Boppana, Johan Håstad, and Stathis Zachos. Does co-NP have short interactive proofs? *Inf. Process. Lett.*, 25(2):127–132, 1987.

[BK96] Hans L. Bodlaender and Ton Kloks. Efficient and constructive algorithms for the pathwidth and treewidth of graphs. *J. Algorithms*, 21(2):358–402, 1996.

[Bod90] Hans L. Bodlaender. Polynomial algorithms for graph isomorphism and chromatic index on partial k-trees. *Journal of Algorithms*, 11:631–644, 1990.

[Bod96] Hans L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth, 1996.

[CM87] Stephen A. Cook and P. McKenzie. Problems complete for deterministic logarithmic space. *Journal of Algorithms*, 8:385–394, 1987.

[Cou90] Brouno Courcelle. Graph rewriting: an algebraic and logic approach. Handbook of theoretical computer science (vol. B). pages 193–242. MIT Press, 1990.

[DLN08] Samir Datta, Nutan Limaye, and Prajakta Nimbhorkar. 3-connected planar graph isomorphism is in log-space. In *FSTTCS*, pages 155–162, 2008.

[DLN+09] Samir Datta, Nutan Limaye, Prajakta Nimbhorkar, Thomas Thierauf, and Fabian Wagner. Planar graph isomorphism is in log-space. In *IEEE Conference on Computational Complexity*, pages 203–214, 2009.

[DNTW09] Samir Datta, Prajakta Nimbhorkar, Thomas Thierauf, and Fabian Wagner. Graph isomorphism for $K_{3,3}$-free and K_5-free graphs is in log-space. In *FSTTCS*, pages 145–156, 2009.

[DTW10] Bireswar Das, Jacobo Torán, and Fabian Wagner. Restricted space algorithms for isomorphism on bounded treewidth graphs. In *STACS*, pages 227–238, 2010.

[EJT10] Michael Elberfeld, Andreas Jakoby, and Till Tantau. Logspace versions of the theorems of Bodlaender and Courcelle. In *FOCS*, pages 143–152, 2010.

[GV06] Martin Grohe and Oleg Verbitsky. Testing graph isomorphism in parallel by playing a game. *In Annual International Colloquium on Automata, Languages and Programming (ICALP)*, 2006.

[Hal76] R. Halin. S-functions for graphs. *J. Geometry*, 8:171–186, 1976.

[Klo94] Ton Kloks. Treewidth: Computation and approximation. *LNCS, Springer, Heidelberg*, 842, 1994.

[Lin92] Steven Lindell. A logspace algorithm for tree canonization. In *STOC*, pages 400–404, 1992.

[RS86] Neil Robertson and Paul D. Seymour. Graph minors. II. Algorithmic aspects of tree-width. *J. Algorithms*, 7(3):309–322, 1986.

[Sch88] Uwe Schöning. Graph isomorphism is in the low hierarchy. *J. Comput. Syst. Sci.*, 37(3):312–323, 1988.

[Wag37] K. Wagner. Über eine Eigenschaft der ebenen Komplexe. *Math. Ann*, 114:570–590, 1937.

[Wan94] Egon Wanke. Bounded tree-width and LOGCFL. *J. Algorithms*, 16(3):470–491, 1994.

Appendix

A Graph Isomorphism and Bounded TreeWidth graphs

The Graph Isomorphism problem is to decide whether two given graphs are isomorphic. The complexity of Graph Isomorphism (**GI**) is a long-standing open problem. **GI** is not known to be solvable in polynomial time and is unlikely to be **NP**-complete [BHZ87], [Sch88]. Polynomial time algorithms are known for **GI** of several special classes of graphs. One such interesting class is graphs excluding a fixed minor. Any family of graphs that is closed under taking minors falls into this class. Important examples of such families include bounded genus graphs, bounded pathwidth graphs and bounded treewidth graphs (a.k.a partial k-trees).

As mentioned earlier, Courcelle's theorem [Cou90] states that for every monadic second-order (MSO) formula ϕ and for every constant k there is a *linear time* algorithm that decides whether a given logical structure of treewidth at most k satisfies ϕ. Unfortunately, it is not known how to formulate **GI** in MSO logic.

Study of **GI** of bounded treewidth graphs has a long history. Bodlaender [Bod90] proved that **GI** of bounded treewidth graphs can be decided in polynomial time. Using a clever implementation of the Weisfeiler-Lehman algorithm, Grohe and Verbitsky [GV06] presented a $\mathbf{TC^1}$ upper bound. Lindell [Lin92] showed that trees can be canonized in logspace.

Arvind, Das and Köbler [ADK08] proved the logspace-completeness of **GI** of partial 2-trees. They decompose the given partial 2-tree into its "tree" of biconnected components and canonize this tree by using a biconnected component canonization as a subroutine. Given any graph it is easy to construct its tree of biconnected components in logspace. Canonization of biconnected components is done by using a structural property of biconnected partial 2-trees [Klo94]. Datta, Limaye and

Nimbhorkar [DLN08] showed that 3-connected planar **GI** is in logspace. Datta et al [DLN+09] showed that the decomposition of biconnected planar graphs into triconnected components can be done in logspace and planar **GI** reduces to 3-connected planar **GI**, hence proving that planar **GI** is in logspace. Datta et al [DNTW09] further extended these results to the classes of graphs excluding $K_{3,3}$ or K_5 as a minor. Their algorithm first decomposes such graphs into their triconnected components, which are known to have special structural properties. Since partial 3-trees are K_5-free, their algorithm implies a logspace algorithm for **GI** of partial 3-trees. All the above mentioned algorithms are based on efficiently computing special decompositions of the input graph.

Using an algorithm of Wanke [Wan94] that computes bounded tree decompositions in **LogCFL**, Das, Toran, and Wagner [DTW10] presented a **LogCFL** algorithm for **GI** of partial k-trees (with $k \geq 4$). Currently this is the best known upper bound.

One of the bottlenecks in the algorithm of Das, Toran, and Wagner is computing bounded width tree decompositions in logspace. Recently Elberfeld, Jakoby and Tantau [EJT10] removed this bottleneck. However, it is still not clear how to design a logspace algorithm for **GI** of partial k-trees. This motivates the study of **GI** on special cases of partial k-trees. Bounded pathwidth graphs are a natural subset of bounded treewidth graphs. The best known upper bound for **GI** of bounded pathwidth graphs is **LogCFL**, implied by the algorithm of Das, Toran, and Wagner [DTW10]. Computing bounded path decompositions in logspace is a natural first step towards improving this upper bound. This is one of the main motivations behind our work.

B Logspace Tree Traversal

In this section, we give a brief description of Lindell's logspace tree traversal algorithm [Lin92].

Let T be a tree. For any node p of T, define the following auxiliary functions:

- $root$: returns the root of T,
- $parent(p)$: returns the parent of p or $null$, if p is the root,
- $firstChild(p)$: returns the lexicographically smallest child of p or $null$, if p does not have any children,
- $nextSibling(p)$: returns the lexicographically smallest child of $parent(p)$ greater than p or $null$, if p is the root or the lexicographically largest child of $parent(p)$.

Lindell proved that these functions can be implemented in logspace.

The algorithm now performs a DFS traversal using the above procedures and storing at each step only the current node p and the last procedure performed LastProcedure. Initially, $p = root$ and LastProcedure $= root$. Update p as follows.

- If LastProcedure is $root$, $firstChild$ or $nextSibling$,
  - if $firstChild(p) \neq null$, let p be $firstChild(p)$,
  - otherwise, if $nextSibling(p) \neq null$, let p be $nextSibling(p)$,
  - otherwise, let p be $parent(p)$.
- Otherwise, LastProcedure is $parent$. Then,

– if $nextSibling(p) \neq null$, let p be $nextSibling(p)$,
– otherwise, if $parent(p) \neq null$, let p be $parent(p)$,
– otherwise, p is the root and hence we terminate.

Update LastProcedure accordingly. Since p and LastProcedure can be stored in logspace and the auxiliary procedures can be done in logspace, this is a logspace procedure.

ECCC ISSN 1433-8092
http://eccc.hpi-web.de