Tree Decomposition: A Feasibility Study

Diplomarbeit

von Hein Röhrig

September 1998

Angefertigt nach einem Thema von Herrn Prof. Dr. Torben Hagerup am
Max-Planck-Institut für Informatik in Saarbrücken

Hiermit erkläre ich, daß ich diese Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Saarbrücken, den 23. September 1998 ....
Hein Röhrig

Abstract

Graphs of small treewidth resemble a tree in a certain (precise) sense. Many computationally hard graph problems can be solved efficiently on graphs of small treewidth using a "tree decomposition," which represents the constructive aspect of treewidth. However, computing minimum-width tree decompositions is NP-hard in general, but for fixed treewidth, there exist algorithms with polynomial running time.

In order to evaluate the practical usability of tree-decomposition algorithms for graphs of arbitrary treewidth, we have implemented several fundamental algorithms related to computing tree decompositions. We present the theory behind solving graph problems using tree decompositions and show how it can be applied in practice to compute "path decompositions." Then we give a survey of the tree-decomposition algorithms considered and discuss their practical value on the basis of benchmarks.

Test graphs were produced using a suite of graph-generating programs that we developed as part of this thesis. Our experiments indicate that the algorithms for graphs of unrestricted treewidth are not viable for input graphs with treewidth beyond the scope of present special-purpose algorithms, which exist for treewidth up to four.

Acknowledgments

I would like to thank Torben Hagerup for introducing me to the subject of this thesis, for many helpful discussions and valuable advice. I am indebted to Volker Priebe and Rudolf Janz for their comments on drafts of the work; Bernd Färber from the Rechnerbetriebsgruppe of the Max-Planck-Institut did me a great favor in allowing me to use the new Starfire computer of MPII before it was opened to the public.

Contents

1 Introduction **6**

2 Using Tree Decompositions **14**
2.1 Preliminaries 14
2.2 The Generic Tree-Automaton Technique 19
2.3 A Recipe 24
2.4 The Implementation 26

3 Finding Path Decompositions **29**
3.1 Applications of Path Decompositions 29
3.2 Interfacing to the Tree-Automaton Technique 32
3.3 Preliminary Characteristics 36
3.4 Compressing Utilization Sequences 42
3.5 The Final Characteristic at Work 47
3.6 Analyzing the Algorithm 55
3.7 The Implementation 57

4 Tree-Decomposition Algorithms **72**
4.1 Shrinking Tree Decompositions 72
4.2 The Separator Approach 80
4.3 The Algorithm by Bodlaender 84

5 Computing Tree Decompositions **91**
5.1 Generating Test Cases 91
5.2 The Algorithm by Arnborg, Corneil, and Proskurowski 94

6 Conclusions **96**
6.1 Shrinking Tree Decompositions Is Not Feasible 96
6.2 Further Directions 97
6.3 Comments on the Development Tools 99

A Notes on the Software **106**
A.1 Graph Utilities 106
A.2 Tree Decomposition and Path Decomposition 109

Bibliography **114**

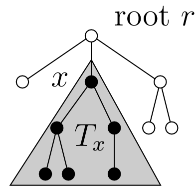
Index **118**

Chapter 1

Introduction

"How much does a given graph resemble a tree?" — this question has led to the notion of the treewidth of a graph and to the related notion of tree decompositions, which represent the constructive content of the treewidth measure of a given graph. In this work, we describe and analyze approaches to derive small tree decompositions of arbitrary graphs.

Trees are very simple graphs: many graph problems can be solved efficiently on trees, because these problems often require only a bottom-up or top-town traversal of the nodes with constant work at each node. Asking how similar an arbitrary graph is to a tree is motivated by the hope of finding efficient algorithms that exploit the "tree-like" structure. Consider for example the IndependentSet problem: given a graph $G = (V, E)$ and an integer ℓ, is there a vertex set $W \subseteq V$ of size ℓ such that no two vertices in W are adjacent? This problem is NP-complete on general graphs [Kar72], but on a tree $T = (V, E)$, we can solve it in linear time in one bottom-up pass: We choose an arbitrary $r \in V$ as root and let $T_x = (V_x, E_x)$ denote the maximal subtree of T with root $x \in V$; proceeding from the leaves up to the root, we mark each node x with a pair of integers (i, j), where i is the size of the largest independent set in T_x that includes x and j is the size of the largest independent set that does not include x. Leaves get labeled $(1, 0)$; for inner nodes, we can easily calculate (i, j) from the corresponding values of the children (see Figure 1): The largest independent set with x in T_x is the union of $\{x\}$ and largest independent sets in the subtrees rooted at the children of x so that each independent set in a subtree does not include the root of the subtree. The largest independent set without x in T_x is the union of the largest independent sets in the subtrees rooted at the children of x;



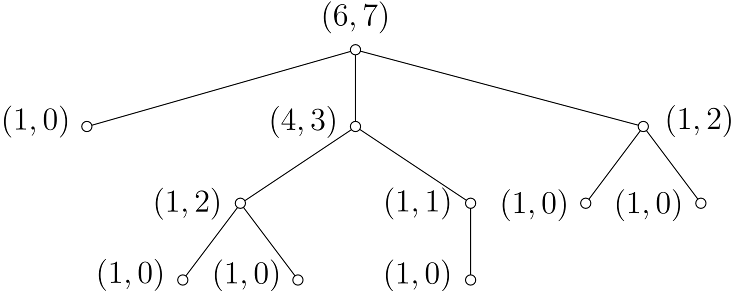
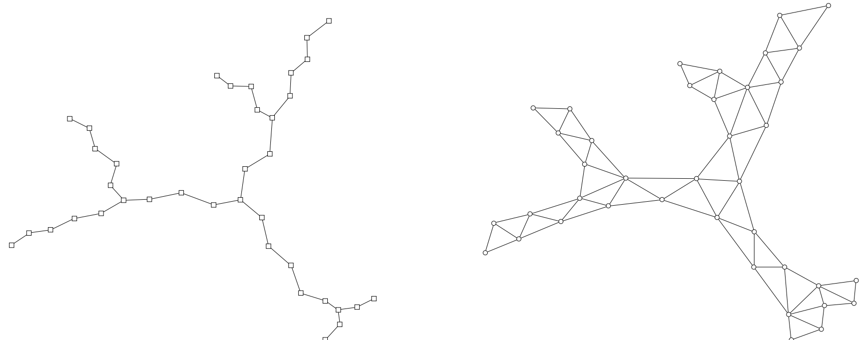


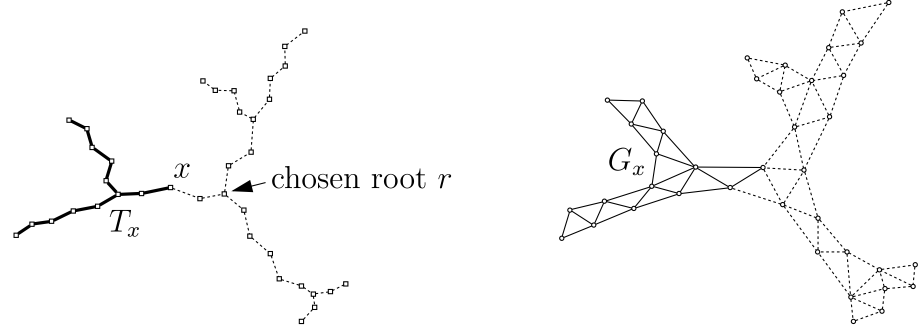
Figure 1: A tree with node labels (i, j) indicating the size of the largest independent set in the subtree with the root (i), and without it (j).

if for the root of T, i or j is greater than ℓ, then the algorithm accepts, otherwise it rejects.

For computing information about its subtree T_x, each node x uses the information from its children. Can this dynamic-programming technique be extended to graphs that in some way look like a tree? The graph below on the right is derived from the tree on the left by replacing each node with a triangle of vertices so that the triangles of adjacent nodes share an edge (and no edge is shared more than once):



We now explain how to extend the algorithm to solve IndependentSet on "tree-like" graphs derived by the "triangle construction" above. In addition to a graph $G = (V, E)$, the input also comprises the instructions for building G, namely a tree $T = (X, F)$ and a mapping $B : x \mapsto \{u, v, w\}$ that associates tree nodes x with triangles u, v, w in the graph G. The computation proceeds bottom-up in T (with a root chosen arbitrarily), and nodes $x \in X$ get labeled with information about certain subgraphs G_x of G. Informally speaking, G_x is the subgraph corresponding to T_x in the tree:

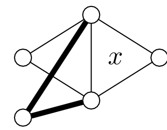


More precisely, for leaves x, G_x is just the triangle $B(x)$, and for inner tree nodes, the graph G_x results from joining the graphs G_y corresponding to children y of x to the triangle $B(x)$. For the root r, G_r is the entire graph. Information about large independent sets in G_x is stored with x, just like the pair (i, j) for T_x in the case of a tree. Note that for each child y, the subgraph G_y shares exactly two vertices with $B(x)$. Independent sets of G_x can be restricted to independent sets of G_y; the restriction of the largest independent set of G_x will be an independent set of G_y that is the largest one satisfying the set-membership status of the two boundary vertices. It is therefore sufficient to label nodes x with three integers (i_1, i_2, i_3) indicating the sizes of the largest independent set in G_x when none, the first or the second boundary vertex must be in the independent set. Finally, if and only if at the root node, the maximum of the integers is at least ℓ, we know that there is an independent set of the required size ℓ.

We have therefore just extended a dynamic-programming algorithm solving a graph problem on trees to a class of somewhat more complex graphs, maintaining the linear running time. In doing so, we made a distinction between the graph G and its “underlying” tree T; constant-time operations at nodes x of the tree produced information about partial solutions on subgraphs G_x corresponding to subtrees T_x with root x. Rather than storing complete solutions, we kept only characteristic data—the size of the largest independent set for each configuration of included boundary vertices.

The outlined approach can be applied to a considerably larger class of graphs and problems. Graphs to which our algorithm can be adapted are

- graphs that allow one triangle edge to be shared by several children:

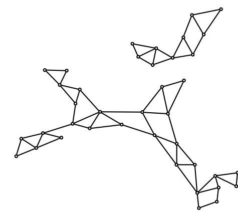


In this case, there is no longer a one-to-one correspondence between

the graph and the tree. For instance, the graph above and the one on the left below both originate from the tree on the right.

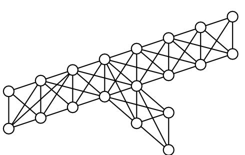
Therefore for each tree node, we need to record where in the parent triangle the two new edges of the graph are connected. The extension of the algorithm itself is straightforward; specifically, the linear running time is preserved.

- graphs resulting from "triangle graphs" by edge deletion:



If the edge between two boundary vertices is missing, they can both be part of a large independent set; hence we need to extend the triples (i_1, i_2, i_3) at tree nodes x to quadruples (i_1, i_2, i_3, i_4) where i_4 indicates the size of the largest independent set in G_x containing both boundary vertices. After edge deletion, the tree from which the graph was constructed is no longer obvious and must therefore be supplied as part of the input.

- graphs constructed by using $= K_4$ or $= K_5$ or larger complete graphs K_{k+1} instead of triangles:



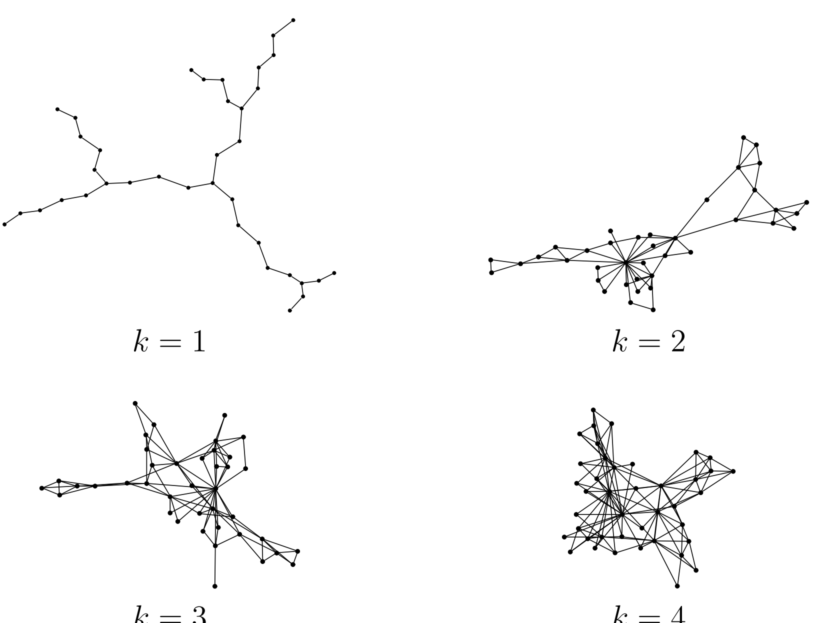
If we use K_{k+1}, there are k vertices on the boundary between parent and child, that is, parent and child overlap on k vertices. For each of the 2^k subsets S of those k vertices, we need to record the size of the largest independent set containing S. It should not come as a surprise that k contributes an exponential factor to the running time of our algorithm—any graph with $n = k + 1$ vertices is a subgraph

of $K_n = K_{k+1}$ and the IndependentSet problem is NP-complete on general graphs.

The class of graphs at which we arrive by applying all three generalizations, that is, by choosing a k, not restricting the degree of tree nodes and taking the closure with respect to edge deletion, will be defined later as "partial k-trees" [Ros74, ACP87] or "graphs of treewidth at most k" [RS83, RS86]. The treewidth k will be taken as a measure of how much a graph resembles a tree. We support this claim with the following remarks:

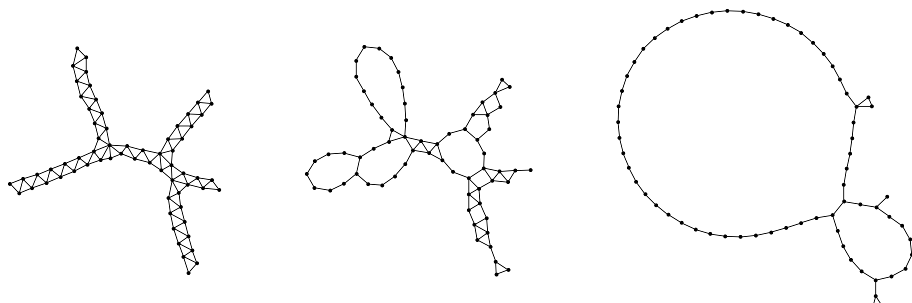
- Collections of trees, called forests, are perfectly "tree-like." They have treewidth $k = 1$.

- With growing k, we employ larger and larger complete graphs K_{k+1} in building graphs of treewidth k. These complete graphs are very much different from trees. Moreover, if we look at graphs of growing treewidth (laid out using a spring-embedder method), they intuitively look less and less like trees:

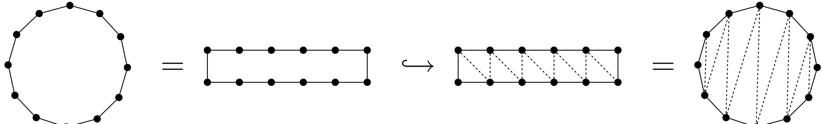


- The running time of our algorithm for IndependentSet depends exponentially on k, thus being linear for trees and graphs of constant treewidth and exponential for general graphs. For general graphs, a tight bound on k is $n - 1$ as it can be shown that K_n cannot be constructed from a tree using K_m with $m < n$ (this is a consequence of Lemma 11 in the next chapter).

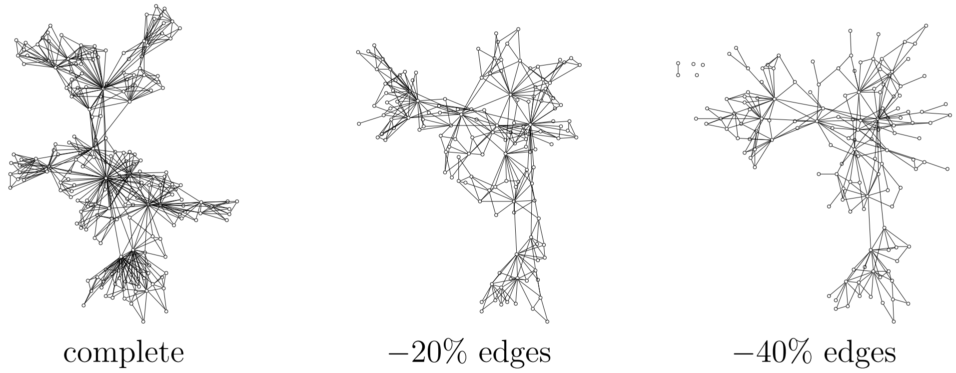
Why do we need to supply the underlying tree with the input to our algorithm? If we delete edges gradually from “triangle graphs,” the underlying tree disappears from our perception when we lay out the graph in a natural way (again using a spring-embedder layout method):



Squeezing cycles will give us potential embeddings of a partial “triangle graph” into a complete one:



For general k-trees however, the situation appears to be much more difficult. What tree structure we have at the start of the following example gets lost by removing edges ($k = 3$):



complete   −20% edges   −40% edges

Let us now take a step back and consider how the observations about resemblance of graphs to trees and its algorithmic use fit into the “big picture.” Clearly, most graphs do not resemble trees—thus talking about algorithms for “graphs of bounded treewidth” means to talk about algorithms that do not work on all graphs, but only on a subset of general graphs. Why should we put up with such a limitation? Our example problem, IndependentSet, is NP-hard and therefore is unlikely to have an efficient, i.e., polynomial-time,

algorithm. Two approaches offer remedy. First, we may decide to settle for the second-best solution and seek approximation algorithms. Second, and this is the route we took in solving IndependentSet on partial k-trees, we can restrict the set of permissible inputs so that the restricted problem can be efficiently solved. Obviously, we want to make this restricted input class as large as possible while maintaining a small running time. Preferably the restriction should be in a certain way natural, parameterized to form an ascending chain $\mathcal{R}_1 \subset \mathcal{R}_2 \subset \ldots$ of more and more general inputs (so that for every input G, there is an index i with $G \in \mathcal{R}_i$) and applicable to a general class of problems. The classes of graphs of treewidth at most k meet all these goals. Keeping k constant, we have a linear-time algorithm for the IndependentSet problem, and by choosing k appropriately large, we cover a rather large class of graphs, including for example series-parallel graphs ($k = 2$) [Bod93] and ℓ-outerplanar graphs ($k = 3\ell - 1$) [Bod93], but also the control flow graph of imperative programming languages [Tho97]. Graphs are ubiquitous combinatorial structures, and the dynamic-programming technique for solving problems using a tree decomposition extends to a large class of problems; Courcelle [Cou90, Bod93] pioneered in formulating logical systems in which each proposition about a graph can be checked efficiently on graphs of bounded treewidth.

However, there are drawbacks and issues that we have not yet addressed. We already noted that letting k grow with the graph size, i.e., $k = |V| - 1$, leads to a class that encompasses all graphs and to an exponential-time algorithm. In general, the time complexity of algorithms operating on graphs of treewidth k will depend at least exponentially on k, since for $k = n - 1$ they reduce to exhaustive search. Worse, the problem of determining treewidth is NP-hard [ACP87], and thus computing minimum-width tree decompositions of arbitrary graphs seems to be out of question. Still, computing the minimum-width tree decomposition of a graph with a known bound k on the treewidth *is* possible, even in time linear in $|V|$, but, again, exponential in k [Bod96a]. Furthermore, the problem of computing tree decompositions exhibits a property called *fixed-parameter tractability* [DF95]—there exist algorithms with running time polynomial (even linear) in n where the degree of the polynomial is independent of k, and k can only influence the "constants."

From a theoretical point of view, we might be quite satisfied with these results. After all, we cannot really expect much more from NP-hard problems. On the other hand, the practical value of these results has not yet been investigated. They definitely merit an assessment of practicality, because the problems solvable efficiently on graphs of bounded treewidth have plenty of real-world applications, and some applications provably produce only problem instances with graphs of bounded treewidth. The goal of this master's

thesis is to shed light on the question of practical usability of general tree-decomposition techniques and algorithms for computing tree decompositions. To this end, Chapter 2 focuses on the dynamic-programming technique using tree decompositions; in Chapter 3, we discuss a major application of the approach, namely how to compute "path decompositions." Chapter 4 presents the candidates for practical tree-decomposition algorithms, among them Bodlaender's linear-time algorithm. Evaluating implementations means producing and executing benchmarks; in Chapter 5, we describe methods to create test inputs and show how our implementation of the tree-decomposition algorithm by Arnborg, Corneil, and Proskurowski [ACP87] performs on them. In Chapter 6, we present our conclusions on the practicality of the various tree-decomposition algorithms and give a short critique of our development environment. Further information on the software can be found in the appendix.

Chapter 2

Using Tree Decompositions

2.1 Preliminaries

In this section, we will give fundamental definitions and properties related to tree decomposition; most of these are drawn from [BK96] and [Bod97], others follow [BH98]. Graphs will always be undirected, finite, simple, and without loops. Some basic notation is fixed in the following definition:

Definition 1 (Graphs and Trees).

(1) $G = (V, E)$ is called a *graph* if V is a finite set of *vertices* and E is a subset of the set $\{\{u, v\} : u, v \in V \text{ and } u \neq v\}$ of unordered pairs $\{u, v\}$ from V, which are called *edges* and which are sometimes written as (u, v).

(2) For $u, v \in V$, we call u and v *adjacent* if there exists an edge $\{u, v\} \in E$. We also say that u is a *neighbor* of v. The *degree* of a vertex is the number of its neighbors.

(3) A graph $G' = (V', E')$ is called a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$. G is also called a *supergraph* of G'.

(4) For $V' \subseteq V$, the *subgraph* $G[V']$ *induced by* V' is the subgraph with the vertex set V' and the edges $E' = E \cap \{\{u, v\} : u, v \in V' \text{ and } u \neq v\}$.

(5) A graph $G = (V, E)$ is *complete*, if there is an edge between every pair of vertices. A complete subgraph with k vertices is called a k*-clique*.

(6) A *path* in G is a subgraph $P = (V', E')$ of G where V' can be written as $V' = \{v_1, \ldots, v_p\}$ so that $E' = \{\{v_i, v_{i+1}\} : 1 \leq i < p\}$. We require that $V' \neq \emptyset$ and define the *length* of P as $p - 1$. P is called a *path between* u *and* v if $v_1 = u$ and $v_p = v$; note that in our definition, a path does not have a distinguished direction.

(7) A graph G is *connected*, if there exists a path between any two of its vertices; a *connected component* of G is a maximal connected subgraph of G.

(8) A *tree* $T = (X, F)$ is a connected graph with $|F| = |X| - 1$. A *subtree* is a connected subgraph of a tree.

(9) A *rooted* tree $T = (X, F, r)$ is a tree with a root $r \in X$. The *depth* of a node $x \in X$ in a rooted tree is the length of a shortest path from x to r. The neighbors of $x \in X$ with greater depth than x are called the *children* of x.

(10) The *root* of a subtree $T' = (X', F')$ of a rooted tree $T = (X, F, r)$ is the node $x \in X'$ with the smallest depth. The *subtree T_x rooted at x* of a rooted tree $T = (X, F, r)$ is the largest subtree of T with root x.

Lemma 2. In a tree, there is exactly one path between any two vertices. Moreover, between two disjoint subtrees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$, there is exactly one path $P = (V', E')$ with $V' = \{v_1, \ldots, v_p\}$ so that $V_1 \cap V' = \{v_1\}$ and $V_2 \cap V' = \{v_p\}$ (and $E' = \{\{v_i, v_{i+1}\} : 1 \leq i < p\}$). □

Unless otherwise noted, for $G = (V, E)$, we set $n = |V|$. When talking about a graph $G = (V, E)$ and an associated tree $T = (X, F)$, the word *vertex* will be reserved for elements of V whereas *node* or *tree node* will be used for elements of X.

To make use of the "tree structure" of the input graph G, the IndependentSet algorithm presented in the introduction needs to know how G derives from a tree. A "tree decomposition" represents this information in a format suitable for algorithms exploiting the bounded treewidth. Recall the strategy we used: Proceeding in the underlying tree from the leaves up to the root, we computed at each tree node x the sizes of large independent sets in the subgraph G_x corresponding to the tree T_x rooted at x. In addition to the sizes produced by the children of x, these sizes depended only on the part of G_x in which solutions for G_y of children y of x could intersect. Thus the part of G_x relevant to the computation at x is precisely the complete graph K_{k+1} (or what remains of it after edge deletion) we put for x during the construction of G; a mapping B from x to the corresponding "triangle" K_{k+1} then is what we need in the general case as well: A mapping from tree nodes x to the corresponding $k + 1$ vertices of G or, equivalently, for each tree node x a bag $B_x \subseteq V$. This leads to the following definition:

Definition 3 (Tree Decomposition). A *tree decomposition* of an undirected graph $G = (V, E)$ is a tree $T = (X, F)$ with bags $B_x \subseteq V$ for each $x \in X$ such that

(1) $\bigcup_{x \in X} B_x = V$

(2) for all graph edges $(u, v) \in E$, there is a tree node $x \in X$ such that $u \in B_x$ and $v \in B_x$

(3) for all tree nodes x, y, $z \in X$: if y is on the path from x to z in T, then $B_x \cap B_z \subseteq B_y$.

If T is a rooted tree, we call $(T, \{B_x\}_{x \in X})$ a *rooted tree decomposition*.

Some explanations of the conditions are in order. (1) is clear, since we want to cover all vertices of the graph. Each edge must be considered at some point during the computation, hence condition (2). Condition (3) enforces a locality constraint; we may look at (3) in the following way:

Proposition 4. Condition (3) in the definition of tree decompositions can be replaced by

(3′) for every vertex $v \in V$, the nodes corresponding to bags containing v form a connected component of T.

We will assume that for each bag $B_x \subseteq V$, the corresponding tree node x is known, which allows us to identify tree nodes x and their bags B_x; the matching picture of a tree decomposition then consists of a tree with bags as nodes where overlapping bags share an ancestor containing the overlap.

Proof. Consider the subgraph $T(v)$ of T induced by the bags containing $v \in V$. If $T(v)$ is not connected, there is a path between two of its components, which has a tree node y that does not contain v, i.e., $v \notin B_y$. This contradicts (3), hence $T(v)$ must be connected.

Conversely, if every vertex occurs in a connected component of T, then for y on the path from x to z, B_y contains all v for which $T(v)$ contains both x and z. Therefore $B_y \supseteq B_x \cap B_z$. □

Definition 5 (Treewidth). The *width* of a tree decomposition is

$$\max_{x \in X} |B_x| - 1.$$

The *treewidth* of a graph is the minimum width of all its tree decompositions.

As desired, trees and forests have treewidth 1—just put every pair of adjacent vertices in a bag of size 2 and make each new bag (but the first) adjacent to an older bag with which it shares a vertex or to any bag, if there is no older bag with which it shares a vertex.

The notion of tree decomposition arose in the context of analyzing a graph. If we take a constructive approach, as we did in the introduction

when we considered extensions to the triangle construction, we arrive at the concept of k-trees, which by Proposition 8 below are maximal graphs of treewidth k.

Definition 6 (k-Trees). The class of (total) *k-trees* is characterized inductively as follows:

(1) The complete graph with $k+1$ vertices, K_{k+1}, is a k-tree.

(2) If $G = (V, E)$ is a k-tree and $B \subseteq V$ is a k-clique in G, then the graph

$$G' = (V \dot{\cup} \{v\},\ E \cup \{(u, v) : u \in B\})$$

that results from adding a new vertex v adjacent to all vertices of the *basis* B of v, is a k-tree.

(3) Only the graphs defined by (1) and (2) are k-trees.

If $G = (V, E)$ is a k-tree, then any graph $G' = (V, E')$ with the same vertices and a subset of edges $E' \subseteq E$ is a *partial k-tree.*

Adding a new vertex yields a new $(k+1)$-clique; if we put each of these into a bag, these bags fit together like a tree. "Thinning out" k-trees yields all possible trees of treewidth k:

Proposition 7. A graph $G = (V, E)$ is a partial k-tree if and only if it has a tree decomposition of width at most k.

Proof. A k-tree has a tree decomposition of width k: The first $(k+1)$-clique makes up a bag; whenever we add a vertex v, we put the resulting $(k+1)$-clique into a new bag and connect this bag to any existing bag that contains the k-clique B to which v was made adjacent. This yields a tree decomposition, since the bags form a tree, all vertices and edges are covered, and vertices occur in connected components of the tree. Moreover, this tree decomposition has bags of size $k+1$ and thus width k. A partial k-tree G is the result of deleting a number of edges from some k-tree G_0, therefore a tree decomposition for G_0 is also a tree decomposition for G.

Given a graph $G = (V, E)$ and a tree decomposition of width at most k, we rewrite the tree decomposition so that we can use it to construct a supergraph $G_0 = (V, E \cup E_0)$ that is a k-tree. Our goal is to arrive at a tree decomposition in which each bag has size $k+1$ and where adjacent bags differ in exactly two vertices. Given such a tree decomposition, we turn the bags into $(k+1)$-cliques by inserting new edges and thus get a k-tree: we can construct the augmented graph starting from any bag and performing a depth-first traversal of the tree, adding for each non-visited neighbor its new

vertex to the graph and making the new vertex adjacent to a k-clique in the current bag.

Any tree decomposition can be adjusted to the required form, maintaining its width: Start by contracting adjacent bags that are equal or where one is contained within the other. Choose arbitrarily a root of the tree decomposition, and complement bags with less than $k+1$ vertices with vertices from their parent bags. Now adjacent bags differ in at least two vertices and their size is $k+1$. Finally, insert new bags between bags that differ in more that two vertices. □

In the following sense, a k-tree is a maximal graph of treewidth k:

Proposition 8. If $G = (V, E)$ is a k-tree, then the graph $G' = (V, E \dot{\cup} \{(u, v)\})$, which is obtained from G by adding a new edge (u, v), has treewidth $k+1$.

Proof. A k-tree with $n = |V|$ vertices has $\binom{k+1}{2} + (n-(k+1))k = nk - \binom{k+1}{2}$ edges. If G' had treewidth k, it would be a subgraph of some k-tree. This is impossible, because G' has $|E \dot{\cup} \{(u, v)\}| = nk - \binom{k+1}{2} + 1$ edges. However, a width-k tree decomposition of G can be turned into a tree decomposition of G' of width $k+1$ by adding vertex u to all bags. □

Further fundamental properties of tree decompositions are presented in the following lemmas. An immediate consequence of the construction in the proof of Proposition 7 is Lemma 9:

Lemma 9. Every graph $G = (V, E)$ has a tree decomposition of size $\Theta(n)$, and any larger tree decomposition can be reduced to linear size in time proportional to the size of the given tree decomposition. □

However, finding minimum-width tree decompositions of general graphs is NP-hard. The TREEWIDTH problem takes as input a graph G and an integer k and decides whether G has treewidth at most k. Arnborg, Corneil, and Proskurowski [ACP87] proved

Theorem 10. TREEWIDTH is NP-complete. □

Lemma 11 and 12 below give conditions under which certain vertices are guaranteed to share a bag; Lemma 11 in particular is an important tool for reasoning about tree decompositions.

Lemma 11. Let K be a clique of G. In any tree decomposition of G, there is a bag that contains all vertices of K.

Proof. Fix a tree decomposition $(T = (X, F), \{B_x\})$ of G. For any vertex v of G, the subgraph $T(v)$ of tree T induced by the bags containing v is (by Proposition 4) a subtree of T. We need to prove that the intersection of all $T(v)$, for $v \in K$, is non-empty. Choosing an arbitrary node of T as root turns every subtree $T(v)$ into a rooted subtree and we can talk of the depth of a node in T. Let $v_0 \in K$ be a vertex whose subtree $T(v_0)$ has the root with the greatest depth among the roots of subtrees $T(v)$, $v \in K$. Because v_0 is adjacent to all $u \in K$, $T(v_0)$ and $T(u)$ overlap, and because $T(v_0)$ has the deepest root, they must overlap at this very root. This holds for all $u \in K$, therefore the bag corresponding to the root of $T(v_0)$ contains all vertices from K. □

Lemma 12. Let $G' = (U \dot\cup W, E')$ be a complete bipartite subgraph of $G = (V, E)$, i.e., $E' = \{(u, w) : u \in U, w \in W\} \subseteq E$. Then in any tree decomposition of G, at least one of U and W will be contained in one bag.

Proof. We use the notation from the previous proof. For $u \in U$ and $w \in W$, $T(u)$ and $T(w)$ overlap because of the edge (u, w). Assume there are $u_1, u_2 \in U$ with $T(u_1)$ and $T(u_2)$ disjoint. Let P be the path in T that connects $T(u_1)$ and $T(u_2)$. For any $w \in W$, $T(w)$ must have non-empty intersection both with $T(u_1)$ and $T(u_2)$, hence it must be a supergraph of P. Consequently, all $T(w)$ overlap on P, so there is a bag containing all $w \in W$. □

2.2 The Generic Tree-Automaton Technique

We will now present and analyze a "generic" algorithm for determining a graph property (such as the existence of a large independent set) using a tree decomposition. The presented form of the framework is due to Bodlaender and Kloks [BK96, Bod97]; the IndependentSet problem of Chapter 1 will serve again as an example.

Our setup is as follows (see Figure 2): As input, we are given a graph $G = (V, E)$ and a tree decomposition $(T = (X, F), \{B_x\}_{x \in X})$. We choose any tree node r as root of T, so that $(T, \{B_x\})$ becomes a rooted tree decomposition and T_x can be defined as the maximal subtree of T rooted at x. Every node x has a bag B_x, which we identify with the subgraph $G[B_x]$ of G induced by the vertices of B_x. Similarly, the subtree T_x gives rise to a vertex set, the union of all bags in T_x, which again is identified with the corresponding subgraph:

Definition 13 (Subgraph at a Tree Node). For a graph $G = (V, E)$, let $(T = (X, F, r), \{B_x\}_{x \in X})$ be a rooted tree decomposition with root $r \in X$. The

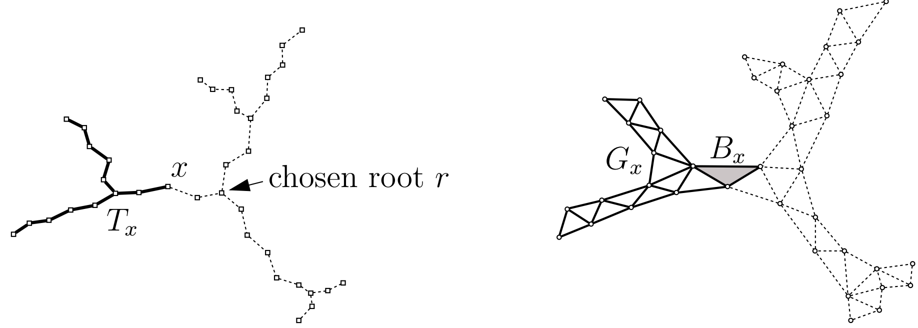


Figure 2: Examples of x, B_x, T_x, and G_x

subgraph G_x *at tree node* $x \in X$ is the subgraph of G induced by the vertices in the bags of $T_x = (X_x, F_x)$, i.e.,

$$G_x = G\left[\bigcup\{B_y : y \in X_x\}\right].$$

Moving in T from the leaves up to the root, the corresponding subgraphs G_x get larger and larger, up to $G_r = G$. At node x with children $y_1, \ldots, y_d$, the graph G_x is the union of B_x and $G_{y_1}, \ldots, G_{y_d}$. The idea is to combine solutions on each of the G_{y_i} to solutions on G_x, taking into account the structure of B_x. For many graph problems it is possible to define the notion of a *partial solution* on a subgraph G_x as the restriction of a solution on G to G_x so that

- the partial solutions on $G_r = G$ include the actual solutions to the problem, and
- partial solutions on each subgraph G_{y_i} at the children y_i of x can be combined to partial solutions of the subgraph G_x of their parent x.

For IndependentSet, we saw already that partial solutions are large independent sets; for HamiltonianCircuit—the problem of finding a path with adjacent endpoints that visits all vertices of G—a partial solution on G_x is a set of disjoint paths in G_x that cover all vertices of G_x and have their endpoints in B_x, or a complete Hamiltonian circuit [Bod97]. The combination of partial solutions is facilitated by the property of tree decompositions that only vertices in B_x can occur in more than one subgraph G_{y_i}:

Lemma 14. Let $G = (V, E)$ be a graph with a rooted tree decomposition $(T = (X, F, r), \{B_x\}_{x \in X})$. Let x be a tree node with children $y_1, \ldots, y_d$. If the vertex $v \in V$ appears in both G_x and $G \setminus G_x$, or if v occurs in at least two of the subgraphs $G_{y_1}, \ldots G_{y_d}$, then v must be in the bag B_x of node x.

Conversely, if some vertex $u \in V$ is contained in bags B_y and B_z, then it also belongs to the bag of the lowest common ancestor of y and z. □

So when combining a partial solution on each G_{y_i} to partial solutions on G_x, "interference" between the partial solutions on the G_{y_i} can only occur via the vertices in B_x. Of course, some partial solutions on subgraphs G_{y_i} may be incompatible with each other and cannot be combined to a partial solution on G_x. Yet if there is a partial solution on G_x, we require that it can be constructed from partial solutions on the G_{y_i}. Since we proceed bottom-up in the tree, we cannot know which partial solution at G_{y_i} can contribute to a partial solution on G_x, therefore at each node, we must be able to compute *all* partial solutions.

What information about partial solutions is passed upwards in the tree? Even to solve decision problems, the combination step needs information with constructive content, such as large independent sets as possible parts of the largest independent set. The tree will usually have $\Theta(n)$ nodes (Lemma 9), hence to get a linear time bound, the algorithm may only perform constant work at each node. Passing entire partial solutions—e.g., independent sets in the subgraph—to parents is not an option, because there may be exponentially many and producing all possible combinations would take exponential time. Therefore, information passed along the edges of the tree must be restricted to *characteristics* of partial solutions so that the number of characteristics at any tree node is bounded by a polynomial. How can we arrange that? Returning to our example, we noted that independent sets of subgraphs G_{y_i} may interfere only on vertices from B_x (Lemma 14), therefore characteristics for independent sets I in G_x are chosen as pairs (s, I') with s the size of the independent set and $I' = I \cap B_x$ the restriction of I to B_x. For a tree decomposition of width k, there are at each node at most $2^{|B_x|} \leq 2^{k+1}$ different pairs (among pairs with the same set I', we discard all but one with the greatest value of s). Since we consider k to be a fixed parameter, we have at most $2^{k+1} = \mathrm{O}(1)$ different characteristics. Generally, when we have found an $\mathrm{O}(\log n)$-size characteristic for a problem, we know that there is only a polynomial number of characteristics and hence a polynomial time algorithm combining characteristics from the leaves up to the root.

A characteristic should convey relevant information about a solution to the problem restricted to a subgraph. As such, a characteristic at some node x indicates that there is a solution to the problem in G_x; characteristics at the root node r thus stand for solutions to the problem on the entire graph $G_r = G$. To decide whether a characteristic exists at the root, we must consider all combinations of characteristics of the children of the root, which in turn result from the characteristics of their respective children. So we

proceed bottom-up in the tree computing *all* characteristics at each node—the *full set of characteristics* of the node—lest we miss some characteristic that represents a necessary part of every solution on the full graph.

Finding the right class of characteristics for a given problem is one half of the problem of applying the general technique. The other half is to define how characteristics are combined during the computation on the tree T. To this end, the rooted tree decomposition is simplified so that there are only four types of tree nodes:

Start Nodes are leaves of the tree and their bags contain only a single vertex.

Introduce Nodes have exactly one child. Their bag contains all the vertices of the child's bag, plus a single newly "introduced" vertex.

Forget Nodes have exactly one child. Their bag contains all the vertices of the child's bag except for exactly one "forgotten" vertex.

Join Nodes have exactly two children, whose bags must contain exactly the same vertices. The bag of a Join node contains the same vertices as the bags of the children.

In Section 2.4, it will be shown how any tree composition can be transformed into this form without increasing its width and that the size of the resulting tree decomposition remains linear in n. Defining the combination of characteristics now means to give four constant-time algorithms, one for each type of node, which on input B_x and *all* characteristics at the children of x produce *all* characteristics at node x. For IndependentSet,

- the algorithm for Start nodes with vertex v returns the two characteristics $(1, \{v\})$ and $(0, \emptyset)$;
- for Introduce nodes x with new vertex v, we take all characteristics (s_y, I'_y) of the single child y and pass them on, including a new characteristic $(s_y + 1, I'_y \cup \{v\})$ if $I'_y \cup \{v\}$ is an independent set within B_x (and hence within G_x).
- if x is a Forget node, we modify all the characteristics (s_y, I') of the child y to $(s_y, I' \setminus \{v\})$;
- at Join nodes, we consider all combinations of two characteristics (s_y, I'_y) and (s_z, I'_z) of the children y and z, respectively, and check whether $I'_y = I'_z$. In that case, a characteristic $(s_y + s_z - |I'_y|, I'_y)$ is produced.

At tree nodes x, combination procedures should only produce characteristics C_x for which a partial solution S_x on the subgraph G_x exists—we call this the *correctness* of combination procedures. Moreover, combination algorithms must also have the *completeness* property: At each tree node x, the

characteristic C_x of every partial solution S_x on the subgraph G_x must be found. This is equivalent to requiring that at every tree node the full set of characteristics is computed. Correctness and completeness are usually proved by *induction on the tree*: The Start-node combination algorithm must yield all characteristics of solutions in the single-vertex graph, and for the other node types, it must be shown that the combination algorithms produce full sets of characteristics from the full set of their children. For the INDEPENDENTSET combination procedures above, the correctness and completeness proofs are straightforward.

In the terms of finite-state automata theory, the generic approach can be interpreted as the construction of a *tree automaton* $A = (Q, \Sigma, Q_I, \delta)$ (our notation follows [Sei90]). The set of states Q is the set of all possible characteristics; the ranked alphabet $\Sigma = \Sigma_0 \dot\cup \Sigma_1 \dot\cup \Sigma_2$ is the disjoint union of tuples describing the possible nodes of a tree decomposition of width k,

$$
\begin{aligned}
\Sigma_0 &= \{(\text{Start}, v) &&: v \text{ a vertex}\} \\
\Sigma_1 &= \{(\text{Introduce}, G, v) &&: G \text{ graph with at most } k+1 \text{ vertices, with } v\} \,\dot\cup \\
&\quad\ \{(\text{Forget}, G, v) &&: G \text{ graph with at most } k \text{ vertices, without } v\} \\
\Sigma_2 &= \{(\text{Join}, G) &&: G \text{ graph with at most } k+1 \text{ vertices}\}
\end{aligned}
$$

so that tree decompositions of width k (and the graph that they describe) can be expressed as words of the tree language T_Σ, which is inductively defined as containing all symbols from Σ_0, all words $a(t)$ with $a \in \Sigma_1$ and t already in T_Σ, and all words $a(t_1, t_2)$ where $a \in \Sigma_2$ and t_1, $t_2 \in T_\Sigma$. The transition relation $\delta \subseteq \bigcup_{d=0}^{2} Q \times \Sigma_d \times Q^d$ contains for Start nodes x with vertex v all tuples $(C_x, (\text{Start}, v))$ where C_x is any characteristic at this node. For Introduce nodes x with child y and introduced vertex v, δ has all transitions of the form $(C_x, (\text{Introduce}, B_x, v), C_y)$ forcharacteristics C_x that can be obtained by inserting v into the child's characteristic C_y. Similarly, for Forget nodes x with child y and forgotten vertex v, δ includes all $(C_x, (\text{Forget}, B_x, v), C_y)$ where the Forget node combination procedure builds C_x from C_y; Join nodes x with children y and z lead to transitions $(C_x, (\text{Join}, B_x), C_y, C_z)$ if C_y and C_z can be merged into C_x, taking into account the structure of the subgraph B_x.

Using the transition relation δ, we can define a tree-automaton computation on a word $w \in T_\Sigma$ as a labeling of the nodes of the tree $w = a(t_1, \ldots, t_d)$ ($a \in \Sigma_d$, t_1, …, $t_d \in T_\Sigma$) with legal transitions from δ. We call a such a labeling a q-computation if the root gets labeled with a transition leading to state q; the language accepted by the tree automaton is the set of trees $w \in T_\Sigma$ that have a q-computation for a q in the set of initial states Q_I. Hence, by setting Q_I to the set of Q of all possible characteristics, our specific tree au-

tomaton A accepts all tree decompositions for which a characteristic at the root node can be found. The fact that the transitions are described by the relation δ and not by a function introduces nondeterminism into the computation of tree automata. The "recipe" for constructing algorithms in the next section can be seen as an instance of the well-known subset construction, where states get replaced by sets of states and the transition between sets of states can be described by a function.

If there is any characteristic of a solution at the root of the tree decomposition, we know that there is at least one solution to the problem on the entire graph. At this point, we have solved the decision problem, but because we have passed only characteristics of partial solutions instead of the partial solutions themselves, an additional effort is needed to actually construct a solution: During the first phase of the algorithm, we store with each characteristic the characteristics that were combined to produce it. We select an arbitrary characteristic at the root node, and from the root to the leaves, we select at each node the characteristic that led to the chosen characteristic at the root. Similar to the original computation of all characteristics, we combine characteristics from the leaves up to the root, but this time, we discard all non-selected characteristics and retain for each characteristic a complete partial solution. At the root, we thus get one solution to the problem on the whole graph. The running time remains linear, if we can combine the partial solutions of the children in constant time at each tree node. This is often possible taking advantage of hints acquired in the combination of the corresponding characteristics and using "implicit" representations of partial solutions that can be merged and extended in constant time, and converted to full solutions in time linear in the size of the full solution.

2.3 A Recipe

The previous section described the intuition behind the notions of the characteristic of a partial solution and of the full set of characteristics. We now construct a "recipe" for fitting problems into the tree-automaton framework for efficiently solving graph problems on graphs of bounded treewidth. The ingredients to solving decision problems in linear time are

(1) the definition of characteristics of partial solutions,

(2) the proof that there is only a constant number of characteristics,

(3) four constant-time algorithms, one for each of Start, Introduce, Forget, and Join nodes, that take as input

- a tree node x of the algorithm's type,

- the bag B_x and introduced or forgotten vertices,
- for each child y_i of x, one characteristic C_{y_i},

and return a set of characteristics at x,

(4) proofs that for every characteristic C_x produced by a combination procedure from $C_{y_1}, \ldots, C_{y_d}$, a partial solution S_x with characteristic C_x exists at x whose restrictions S_{y_i} to G_{y_i} have characteristic C_{y_i} (this is the correctness property) and that for every partial solution S_x, the combination procedure for x finds the characteristic C_x of S_x, provided that the combination procedure is called for all combinations of characteristics from the full sets at children y_i (this is the completeness property).

Theorem 15. If the prerequisites (1)–(4) are met for some decision problem P, there is a linear-time decision algorithm for P.

Proof. Moving from the leaves up to the root, we compute characteristics by invoking the algorithms of ingredient (3) for each combination of children's characteristics and taking the union of the resulting sets of characteristics. By induction on the tree and by applying (4), these sets are full sets of characteristics. If the full set $\mathcal{C}_r$ at the root is non-empty, we accept, otherwise we reject. The running time is $O(1)$ at each node, because the algorithms at each node are invoked only a constant number of times. □

Proceeding from decision problems to computing solutions, we need to supply an additional ingredient:

(5) four polynomial time algorithms, one for Start, Introduce, Forget, and Join nodes, that take as input

- a tree node x of the algorithm's type,
- the bag B_x and introduced or forgotten vertices,
- a characteristic C_x from the full set at x,
- for each child y_i of x, a pair (C_{y_i}, S_{y_i}) of the characteristic C_{y_i} at y_i that led to C_x and a partial solution S_{y_i} at y_i with characteristic C_{y_i},

and produce as output a partial solution S_x at x that has characteristic C_x and whose restriction to G_{y_i} is S_{y_i}.

Theorem 16. If the conditions (1)–(5) are met for some problem P, there is a polynomial-time algorithm computing a solution to P. If solving the decision problem using Theorem 15 takes time $O(n)$, and the algorithms of ingredient (5) have constant running time, the solution can be computed in time $O(n)$.

Proof. Compute a characteristic of the root node using the algorithm outlined in the proof of Theorem 15, but store with each new characteristic pointers to the corresponding characteristics of the children. Then apply the algorithms of ingredient (5) in a bottom-up pass on the tree. □

2.4 The Implementation

We successfully implemented the abstract tree-automaton technique as a generic C++ "template class" [SE90]. For a concrete problem, the ingredients of the recipe from Section 2.3 are substituted into this template: Generic (compile-time) parameters supply the class of characteristics (ingredient (1)), the four combination algorithms (ingredient (3)) and optionally four algorithms to construct solutions from characteristics (ingredient (5)). By this means, we obtained algorithms for deciding IndependentSet, Coloring, and Pathwidth as well as for solving the corresponding construction problems. The Pathwidth problem and computing path decompositions will be treated in detail in the next chapter.

An input instance consists of a graph $G = (V, E)$, a tree decomposition $(T = (X, F), \{B_x\}_{x \in F})$ of G and parameters specific to the problem such as the number of colors for Coloring; we assume that the tree decomposition has size $\mathrm{O}(n)$ and let k denote its width. Processing the input starts with the conversion of the supplied tree decomposition into a rooted tree decomposition with Start, Introduce, Forget, and Join nodes. A root of T is chosen arbitrarily. Then a recursive algorithm converts subtrees of the input tree decomposition into the desired form, creating for leaves a Start node and a chain of Introduce nodes, generating a chain of Join nodes for each node with at least two children, and replacing nodes with a single child by chains of Forget and Introduce nodes. A rough estimate of the number of resulting nodes can be obtained as follows: There are as many Start nodes as leaves; each node of the original tree decomposition causes at most $k + 1$ Introduce nodes to be created; every vertex of the graph is forgotten exactly once; and there are at most twice as many Join nodes as there were nodes of degree greater than two. Therefore the converted tree decomposition still has size $\mathrm{O}(n)$; Kloks [Klo94] shows that with a more involved algorithm, the number of Start, Introduce, Forget, and Join nodes can be limited to at most $4n$.

In the further discussion, $x \in X$ denotes a tree node with children $y_1, \ldots, y_d$, i.e., $d = 0$ for Start nodes, $d = 1$ for Introduce and Forget nodes, and $d = 2$ for Join nodes. The straightforward way of finding a characteristic at the root of T would be to compute in a bottom-up pass on T the full sets of characteristics at every node; at node x, we would repeatedly

invoke the combination procedure appropriate for x with all combinations $(C_{y_1}, \ldots, C_{y_d})$ of one characteristic from the full set of each child y_i of x. This approach is unsatisfactory because we are only interested in a single characteristic at the root and not in all of them. Moreover, we would like to compute at each node only as many characteristics as necessary to find the characteristic at the root. For this reason, the computation of characteristics is pipelined: every node x remembers the state of the computation of the full set of characteristics $\mathcal{C}_x$ at x and when the parent node asks for the next characteristic from $\mathcal{C}_x$, it resumes the computation of the full set until a new C_x is found, then sends C_x to the parent, and suspends the execution until the parent issues the next request (see Section 6.3 for a discussion of pipelining in C++). Join nodes need to combine all pairs of characteristics from their two children, so, in general, they ask more than once for the same characteristic. Therefore, we store at each node the characteristics already computed in a "cache" to avoid computing them again (otherwise, we would violate the linear time bound). At some point, the problem-specific part of the algorithm may signalize that at node x no further characteristics can be found—then the cache at x must contain the full set of characteristics $\mathcal{C}_x$. Since at x, all requests for characteristics can now be satisfied from the cache, the caches at nodes in the subtree T_x can and will be discarded.

Once a characteristic at the root has been found, the algorithm enters a second stage, in which a solution is computed bottom-up by functions constructing a partial solution at any node x. The functions of ingredient (5) take as parameters a characteristic C_x at node x and for each of its children $y_1, \ldots, y_d$ pairs (C_{y_i}, S_{y_i}) where the S_{y_i} are partial solutions at y_i with characteristic C_{y_i} and where the C_{y_i} can be combined to C_x. Before we can call these solution-computing procedures, we have to determine at each node x a characteristic C_x so that characteristics of siblings can be combined to the characteristic of their parent. During the first stage of the algorithm, we discard the characteristics cached at x as soon as the full set of characteristics at the parent of x has been found. Therefore we have to recompute discarded characteristics by enumerating the characteristics of the children $y_1, \ldots, y_d$ of x until a combination of $(C_{y_1}, \ldots, C_{y_d})$ is found that gives rise to C_x. In recomputing, we use the caches again; for situations where memory is scarce, we optionally flush caches in the second stage as in the first stage. However, flushing caches in the second stage means sacrificing the linear running time—it may happen that the characteristics at the leaf nodes have to be recomputed $\mathrm{O}(h) = \mathrm{O}(n)$ times, where h is the height of T (see also Figure 24 in Chapter 3).

Additional features of our implementation of the tree-automaton technique are the elimination of redundant characteristics based on a problem-

specific partial order on the characteristics (assuming that "greater" characteristics are subsumed by "smaller" characteristics, as in the INDEPENDENT-SET problem, where characteristics (s_x, I'_x) were discarded in favor of characteristics $(\tilde{s}_x, I'_x)$ with $\tilde{s}_x > s_x$) and the gathering of statistics such as the number of characteristics computed and the time spent in the stages of the algorithm. Translating applications of the abstract tree-automaton technique into separate software modules for the tree automaton and problem-specific parts had the advantages that

- complex algorithms could be decomposed into small functions with clearly defined and simple requirements,
- all problems-specific implementations equally benefited from features and enhancements of the tree-automaton module,
- for new problems, the tree automaton did not need to be programmed from scratch, and
- independent testing was possible.

The performance of our implementation, including the effect of optimizations, will be discussed at the end of the next chapter in conjunction with the computation of path decompositions; information on installing our tree-decomposition software with the tree-automaton template is given in Section A.2.

Chapter 3

Finding Path Decompositions

This chapter is devoted to a particular application of the framework developed in the previous chapter: given a graph G, a bounded-width tree decomposition of G, and an integer ℓ, we compute a *path decomposition* of G of width ℓ, if one exists. Path decompositions are special cases of tree decompositions where the underlying tree is actually a path; the minimum-width path decomposition defines the *pathwidth* of a graph. Every path decomposition is also a tree decomposition, but, in general, a minimum-width tree decomposition will not be a path decomposition. This implies that the treewidth of a graph is bounded from below by its pathwidth; just like TREEWIDTH, the PATHWIDTH problem of deciding whether a graph has a path decomposition of at most a given width is NP-complete [ACP87].

3.1 Applications of Path Decompositions

We are interested in computing path decompositions for two reasons: First, computing path decompositions turns out to be an important problem in VLSI design [Möh90] and, second, the algorithm for computing path decompositions can be extended to compute tree decompositions. The latter may seem paradoxical because the algorithms already get a tree decomposition as part of the input, but an investigation of the enhanced algorithm in Chapter 4 will show that this algorithm can convert tree decompositions of any width into a tree decomposition of width ℓ, provided G has treewidth at most ℓ. Both the path-decomposition and the tree-decomposition variant of the algorithm were developed by Bodlaender and Kloks [BK96].

Gate arrays are a design style for integrated circuits where the silicon wafers have been pre-processed to a certain fabrication step, and "personalization" to a concrete application usually amounts to adding a final single

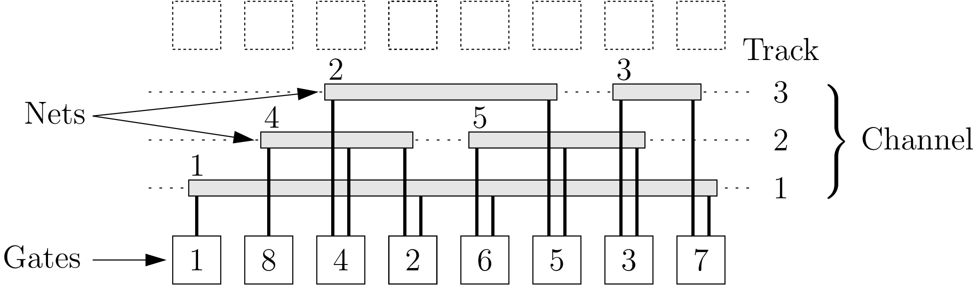


Figure 3: A gate matrix layout (based on [Bod93]).

layer of metal [WE85, Möh90]. The generic wafer is made up of rows (arrays) of gates separated by routing channels, which obey strict directional control over routing: As depicted in Figure 3, each channel consists of two layers, one for horizontal routing, the other for vertical routing. Through vertical wires, every gate can connect to any of the horizontal *tracks*; the number of tracks determines the distance between rows of gates, therefore one would like to minimize their number in order to fit more rows on the chip. *Nets* are hyperedges connecting several gates; given a number of gates assigned to a particular row and nets, our task is to arrange the gates in that row (find a permutation of the gates) and assign nets to tracks such that nets on the same track do not overlap and the number of tracks is minimized—we disregard the possibility of a net changing tracks. For n nets and m gates, the input can be encoded in an $n \times m$ boolean matrix $M = (m_{ij})_{1 \leq i \leq n, 1 \leq j \leq m}$ such that $m_{ij} = 1$ if and only if net i is connected to gate j; M is called the gate matrix, hence the problem name GateMatrixLayout. See the left-hand side of Figure 4 for an example.

Solving the GateMatrixLayout problem is equivalent to computing a minimum-width path decomposition (this result is due to Fellows and Langston [FL89]): We construct a graph G by creating a vertex v_i for each net i and by linking v_i and $v_{i'}$ by an edge whenever there is a gate j connected to both net i and net i' (i.e., $m_{ij} = 1 = m_{i'j}$, see Figure 4, right-hand side). It follows that the vertices of nets connected to the same gate form a clique. A path decomposition of G translates into a gate-matrix layout: go through the bags of the path decomposition from the left to the right and at each bag,

- if v_i occurs for the first time, assign net i to the lowest currently unused track so that it starts at the current position and ends at the last gate to which it is connected,

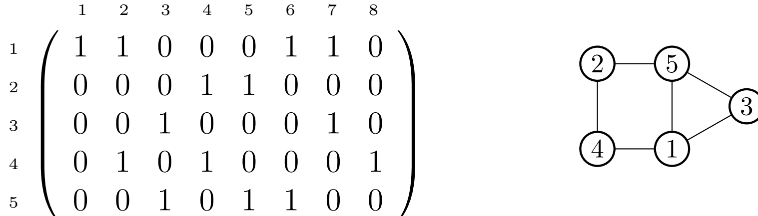


Figure 4: The input gate matrix for the example of Figure 3 and the corresponding graph. Every column of the matrix represents a gate, every row a net. $m_{ij} = 1$ means that gate j must be connected to net i; e.g., net 1 links gates 1, 2, 6 and 7, and gate 2 is connected to nets 1 and 4. The path decomposition formed by a path of the bags $\{1\}, \{1,4\}, \{1,4,2\}, \{1,5,2\}, \{1,5,3\}, \{1,3\}$ corresponds to the solution of Figure 3.

- if gate j is not yet placed and the current bag contains all nets to which j is connected, append j to the list of placed gates.

By Lemma 11, all nets of j occur in some bag, so all gates get placed; the number of tracks used equals the size of the largest bag. On the other hand, an arbitrary G gives rise to an instance of GateMatrixLayout: For every vertex v, we create a net i_v, and add for every edge (u,v) a gate $j_{(u,v)}$ connected to the nets i_u and i_v. From a layout of this instance, we produce a path decomposition by creating for each gate j a bag that contains the vertices v of nets i_v above gate j; the width of the path decomposition is strictly smaller than the number of tracks used.

The Pathwidth problem is also closely related to Bandwidth and other problems measuring the "width" of total vertex orderings. For a graph $G = (V,E)$, the bandwidth of a total ordering on the vertices $f : V \leftrightarrow \{1,\ldots,|V|\}$ is defined as the maximum distance $|f(u) - f(v)|$ between two adjacent vertices u and v, and the bandwidth of the graph is the minimum bandwidth of all vertex orderings. We show that the bandwidth is an upper bound on the pathwidth of a graph; further relations are listed in [Bod96b]. Given a graph $G = (V,E)$ of bandwidth k and an ordering f, we construct a path decomposition of G by defining bags $B_{f(u)} := \{v : 0 \leq f(v) - f(u) \leq k\}$ and linking $B_{f(u)}$ and $B_{f(v)}$ by an edge if $|f(u) - f(v)| = 1$. If the ordering has bandwidth k, the bags have size at most $k+1$, every edge $(u,v) \in E$ is covered by $B_{\min\{f(u),f(v)\}}$ and each vertex u occurs in the contiguous sequence of bags $B_{\max\{0,f(u)-k\}}, \ldots, B_{f(u)}$; hence the bags linked in this manner constitute a path decomposition of G.

3.2 Interfacing to the Tree-Automaton Technique

The remainder of this chapter is devoted to the construction and analysis of the path-decomposition computation algorithm by Bodlaender and Kloks and our implementation of it. The aim is to "prepare" a linear-time algorithm according to the recipe of Chapter 2, therefore we need to specify constant-size characteristics and constant-time combination algorithms—in other words, the number of characteristics and the time bound of the combination algorithms may depend arbitrarily on the width k of the input tree decomposition and the desired pathwidth ℓ but not at all on the number of vertices of the input graph G. Characteristics are to represent partial solutions, which are path decompositions of width at most ℓ in some subgraph G_x of G; as we saw, characteristics need to carry the information necessary to build from characteristics of siblings the characteristics of their parent.

Given a graph $G = (V, E)$, a tree decomposition $(T = (X, F), \{B_x\}_{x \in X})$ of G and a requested pathwidth ℓ, we plan to provide, for computing a path decomposition of width at most ℓ (or determining that G has pathwidth $> \ell$),

the definition of a characteristic of a path decomposition, such that the number of characteristics is independent of $n = |V|$,

four combination algorithms with a time bound independent of n, which at the four different types of tree nodes $x \in X$ compute characteristics at x from characteristics of the children of x, and

four solution-computing algorithms that expand characteristics to path decompositions.

We let k denote the width of the tree decomposition, and assume that the tree decomposition has only Start, Introduce, Forget, and Join nodes. Partial solutions S_x at tree nodes x are path decompositions of width at most ℓ of the subgraph G_x. To distinguish these path decompositions from the tree decomposition given with the input, we mark components of the former by a hat ($\hat{}$); furthermore, instead of writing $S_x = (\hat{P} = (\hat{X}, \hat{F}), \{\hat{B}_i\}_{i \in \hat{X}})$ for a partial solution at tree node x, we denote such a path decomposition by a sequence $S_x = \langle \hat{B}_i \rangle_{1 \leq i \leq m}$, with the degenerate tree $\hat{P} = (\hat{X}, \hat{F})$ given implicitly by nodes $\hat{X} = \{1, \ldots, m\}$ and edges $\hat{F} = \{(i, i+1) : 1 \leq i < m\}$. A sequence $\langle \hat{B}_i \rangle_{1 \leq i \leq m}$ is a path decomposition of G_x if and only if each vertex $v \in G_x$ occurs precisely in a contiguous subsequence $\langle \hat{B}_i \rangle_{\mathrm{first}(v) \leq i \leq \mathrm{last}(v)}$ and each edge of G_x is covered by some bag (i.e., for each edge there is a bag containing both endpoints).

Given a partial solution S_x at tree node x, how do we derive a suitable characteristic C_x? Using $S_x = \langle \hat{B}_i \rangle_{1 \leq i \leq m}$ itself as characteristic is ruled out

by the fact that S_x grows with the number of vertices of G_x, and hence depends substantially on n. But let us postpone this issue for a moment and consider how the algorithm would work with $C_x = S_x$. Then it will be easier to identify the relevant information about S_x, which needs to be stored in C_x.

A Start node x with vertex v would produce all possible path decompositions of the graph $G_x = (\{v\}, \emptyset)$, e.g.,

$$\langle\{v\}\rangle, \;\; \langle\emptyset, \{v\}\rangle, \;\; \langle\emptyset, \{v\}, \emptyset\rangle, \;\; \langle\emptyset, \emptyset, \{v\}, \{v\}, \emptyset\rangle, \;\; \ldots$$

and so on. For the moment, we ignore that there is an infinite number of such path decompositions. An Introduce node x with child y and new vertex v takes each path decomposition S_y of G_y (produced at node y) and creates candidates $\tilde{S}_x$ for path decompositions of G_x by inserting v into all contiguous subsequences of bags in copies of S_y. If a candidate $\tilde{S}_x$ consists of bags with at most $\ell+1$ elements, and if in $\tilde{S}_x$, all edges between v and vertices in B_y are covered, then $S_x := \tilde{S}_x$ is a path decomposition of G_x of width at most ℓ and is inserted into the output set. A Forget node passes on its child's path decompositions unchanged (note that we must not remove the forgotten vertex); a little more work is needed for Join nodes x with children y and z: We merge only path decompositions S_y and S_z whose paths are of the same length and check whether the pairwise union is a path decomposition of G_x of width at mast ℓ. If there is a path decomposition of width at most ℓ of the entire graph, it is found in principle using the given four combination procedures.

We can limit the number of partial solutions to a finite value by only generating partial solutions S_x in which adjacent bags differ. Start nodes then produce $\langle\{v\}\rangle$, $\langle\emptyset, \{v\}\rangle$, $\langle\{v\}, \emptyset\rangle$ and $\langle\emptyset, \{v\}, \emptyset\rangle$; Introduce nodes optionally duplicate the first and last bags in which the new vertex v is to be put; and, as before, Forget nodes do nothing. At Join nodes, bags in S_y and S_z are repeated in all possible ways to bring S_y and S_z to the same length. After merging the expanded bag sequences bag by bag, we eliminate repetitions of consecutive bags. Since we aim for a finite number of characteristics that is independent of n, we assume in the following that in a partial solution $S_x = \langle\hat{B}_i\rangle_i$ no bag is repeated.

How do we achieve an equivalent computation with constant-size characteristics? Looking for some kind of "compression", we recall the interference property of tree decompositions (Lemma 14): If, at two nodes y and z, partial solutions S_y and S_z share vertices, the shared vertices are in the bag B_x of the lowest common ancestor x of y and z. Thus, at x, only vertices in B_x determine the compatibility of partial solutions at children of x, and

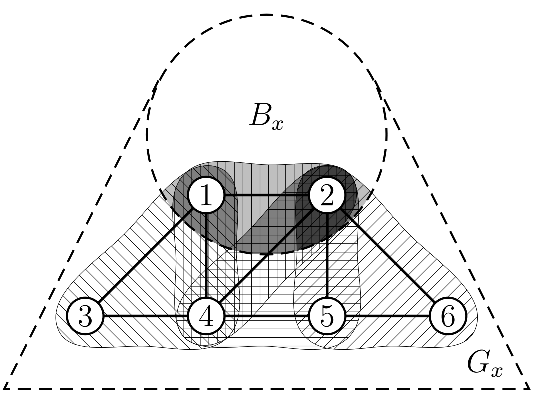


Figure 5: The reduction operation. The bags $\hat{B}_1 = \{1,3,4\}$, $\hat{B}_2 = \{1,2,4\}$, $\hat{B}_3 = \{2,4,5\}$, $\hat{B}_4 = \{2,5,6\}$ of a path decomposition of G_x are projected to $\bar{B}_1 = \{1\}$, $\bar{B}_2 = \{1,2\}$, $\bar{B}_3 = \{2\}$ ($\hat{B}_3$ and $\hat{B}_4$ both map to $\bar{B}_3$).

the size of B_x is bounded by $k+1$. Consequently, to form one part of the characteristic C_x of $S_x = \langle \hat{B}_i \rangle_{1 \le i \le m}$, we eliminate from S_x all vertices not in B_x, getting $\langle \hat{B}_i \cap B_x \rangle_{1 \le i \le m}$, and from this sequence we discard repeated sets, arriving at $\langle \hat{B}_{i_j} \cap B_x \rangle_{1 \le j \le m'} =: \langle \bar{B}_j \rangle_{1 \le j \le m'}$ (see Figure 5). We call $\langle \bar{B}_j \rangle_{1 \le j \le m'}$ a *reduced bag sequence*. Thanks to the removal of equal contiguous sets, the length of such sequences is bounded by $2k+3$: for $k=0$ the bound is 3, and extending a reduced sequence by one vertex, we can duplicate at most two bags, so incrementing k means increasing the length bound by 2. Each of up to $k+1$ vertices occurs in some bag for the first time, and in some other bag for the last time; hence there are at most $\binom{i}{2}^{k+1} \le i^{2k+2}$ ways of placing $k+1$ vertices into i bags, and by summing over i, it follows that there are at most $(2k+3)^{2k+3}$ sequences $\langle \bar{B}_j \rangle_j$, a number independent of n. Also note that $\langle \bar{B}_j \rangle_j$ is a path decomposition of B_x—this will be an important invariant of the final characteristic.

Can reduced sequences $\langle \bar{B}_j \rangle_j$ themselves serve as characteristics C_x? Not quite. Going from $S_x = \langle \hat{B}_i \rangle_i$ to $\langle \bar{B}_j \rangle_j$, we lose too much information about S_x. Notably, we need to supplement the reduced sequences $\langle \bar{B}_j \rangle_j$ with information about how full the bags were before $\langle \hat{B}_i \rangle_i$ was reduced to $\langle \bar{B}_j \rangle_j$. We argue that two simple variants of recording "bag utilization" with the reduced bag sequences do not meet the requirements and show how a sophisticated approach achieves the desired result. Storing with each reduced bag $\bar{B}_j$ the size of the largest original bag $\hat{B}_i$ that was reduced to $\bar{B}_j$ leads to incomplete combination algorithms (see Sections 2.2 and 2.3 for a discussion of correctness and completeness). Suppose at node x, the path decomposition S_x of G_x

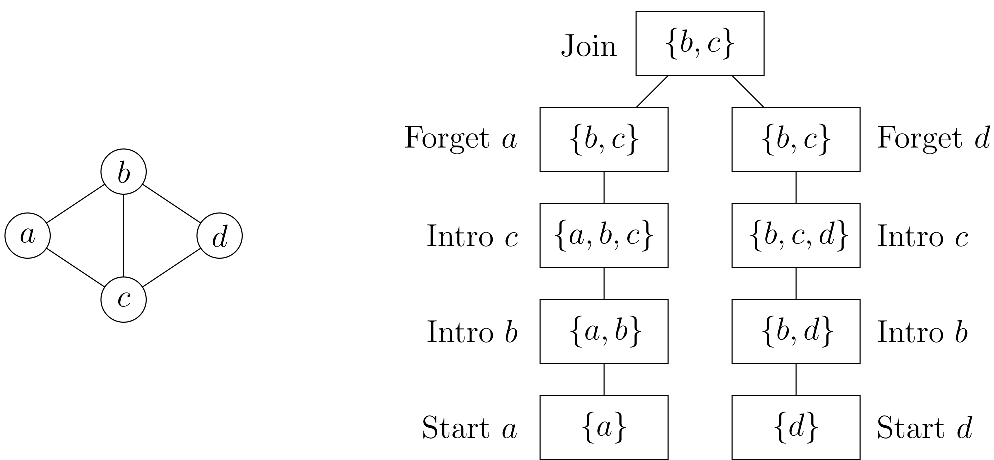


Figure 6: Example graph and tree decomposition

is characterized by a sequence $\langle(\bar{B}_j, u_j)\rangle_{1\leq j\leq m'}$ where $u_j \in \mathbb{N}$ records the size of the largest $\hat{B}_i$ projected to $\bar{B}_j$. The envisaged combination procedure for Join nodes x with children y and z considers only characteristics of children

$$C_y = \langle(\bar{B}_{y,j}, u_{y,j})\rangle_{1\leq j\leq m_y} \qquad \text{and} \qquad C_z = \langle(\bar{B}_{z,j}, u_{z,j})\rangle_{1\leq j\leq m_z}$$

with equal bag sequences, i.e., $m_y = m_z$ and $\bar{B}_{y,j} = \bar{B}_{z,j}$ for $1 \leq j \leq m_y$. After setting $m_x := m_y$, $\bar{B}_{x,j} := \bar{B}_{y,j}$, and $u_{x,j} := u_{y,j} + u_{z,j} - |\bar{B}_j|$, it discards all results $C_x = \langle(\bar{B}_{x,j}, u_{x,j})\rangle_{1\leq j\leq m_x}$ with any $u_{x,j} > \ell + 1$. This procedure is correct because we can merge partial solutions S_y at node y with characteristic C_y and S_z at node z with characteristic C_z to a path decomposition S_x of G_x with characteristic C_x. Unfortunately, there are partial solutions S_y and S_z that can be combined to a solution S_x but for which the combination procedure does not yield a characteristic. Take for example the graph with the tree decomposition of width 2 shown in Figure 6. Characteristics of the left "Intro c" node need to cover the clique $\{a, b, c\}$ (Lemma 11), so they take the form

$$\langle \ldots, (\{a, b, c\}, 3), \ldots\rangle$$

where the other reduced bags are proper subsets of $\{a, b, c\}$. Therefore at the left Forget node, each characteristic will contain a pair $(\{b, c\}, 3)$, as will every characteristic at the Forget node on the right. Merging any characteristics of the two Forget nodes at the Join node yields a pair $(\{b, c\}, 3 + 3 - 2)$, and the result is discarded, even though the underlying graph clearly has pathwidth 2.

The shortcoming of storing the maximum utilization with each bag in the reduced sequence is caused by the fact that $G[\{a,b,c\}]$ has path decompositions whose restriction to $B_{\text{Forget } a} = \{b,c\}$ contains not only bags $\{b,c\}$ with utilization 3 but also some with utilization *less* than 3, namely 2. Those bags $\hat{B}_i = \{b,c\}$ do have room to accommodate node d, but our approach of only remembering the maximum utilization does not take this into account. Instead, we might resort to the other extreme and store with each $\bar{B}_j$ the *utilization sequence* of bag sizes $\langle |\hat{B}_i| \rangle_{i_0 \le i \le i_1}$ for the bags $\hat{B}_i$ with restriction $\hat{B}_i \cap B_x = \bar{B}_j$. While sufficient for showing correctness and completeness of suitable combination procedures, the length of utilization sequences depends on the size of the subgraph G_x and hence on n. However, let us first prove that such "preliminary" characteristics of non-constant size are adequate with regard to correctness and completeness. Later we will find a compromise between size and information content and amend the following proofs for the final form of the characteristic of a path decomposition.

3.3 Preliminary Characteristics

The "preliminary" characteristic of a path decomposition $S_x = \langle \hat{B}_i \rangle_{1 \le i \le m}$ of a subgraph G_x is computed as follows: We assume that consecutive bags in S_x differ by exactly one vertex, otherwise we remove repeated bags and insert new bags between bags that differ in more than one vertex. We set u_i to the size of $\hat{B}_i$ and restrict $\hat{B}_i$ to the bag B_x of tree node x. Proceeding from the left to the right, we remove repeated equal sets $\hat{B}_i \cap B_x$ and build from the corresponding $u_i = |\hat{B}_i|$ a sequence $\langle u_{j,1}, u_{j,2}, \ldots, u_{j,n_j} \rangle$, which is stored with $\bar{B}_j := \hat{B}_i \cap B_x$, giving a characteristic

$$C_x = \langle (\bar{B}_j, \langle u_{j,1}, u_{j,2}, \ldots, u_{j,n_j} \rangle) \rangle_{1 \le j \le m'}$$

The steps of deriving a preliminary characteristic are shown in Figure 7, while the operation of projecting a normalized bag sequence to B_x and constructing the utilization sequences is depicted schematically in Figure 8.

To put the preliminary characteristics to work, we need to show how to combine characteristics at the four different node types so that for each node x and each partial solution S_x (a path decomposition of G_x with width at most ℓ) the characteristic C_x of S_x is built; moreover, for any computed C_x there must be at least one S_x with characteristic C_x.

Start Nodes

A Start node x with vertex v has path decompositions $\langle \{v\} \rangle$, $\langle \emptyset, \{v\} \rangle$, $\langle \{v\}, \emptyset \rangle$ and $\langle \emptyset, \{v\}, \emptyset \rangle$—remember that we decided to discard repeated bags and to

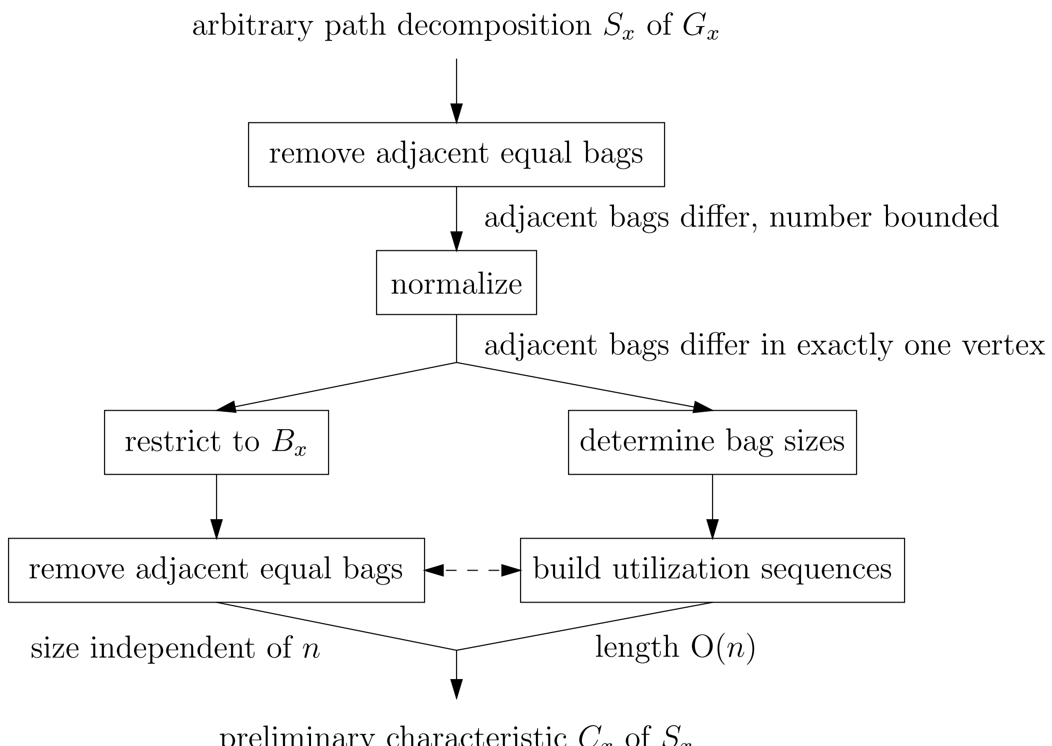


Figure 7: Computation of "preliminary characteristic"

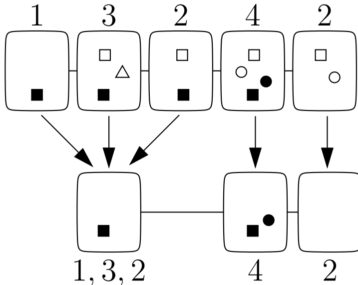


Figure 8: Example of deriving a preliminary characteristic. Vertices in B_x are filled black, i.e., the characteristic is $\langle(\blacksquare, 1, 3, 2), (\blacksquare\bullet, 1), (\emptyset, 2)\rangle$.

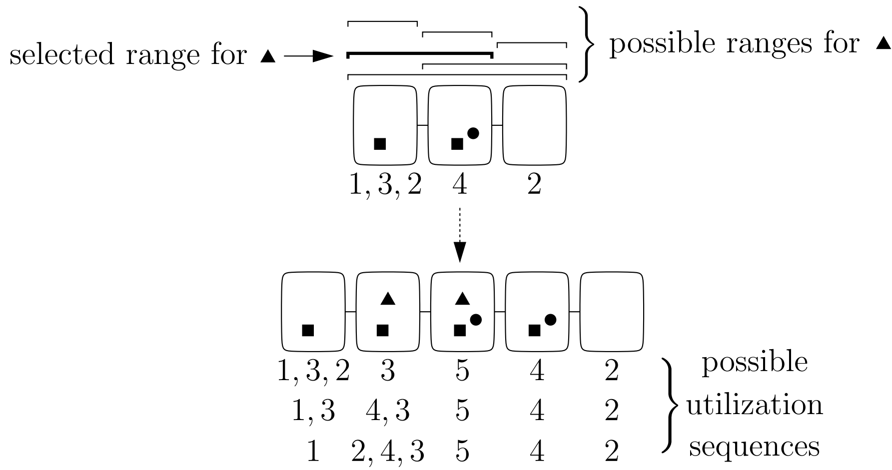


Figure 9: Inserting ▲ into a preliminary characteristic

compensate for this restriction by adapting the combination procedures. The corresponding set of preliminary characteristics is

$$\mathcal{C}_x = \{\langle(\{v\},1)\rangle,\ \langle(\emptyset,0),(\{v\},1)\rangle,\ \langle(\{v\},1),(\emptyset,0)\rangle,\ \langle(\emptyset,0),(\{v\},1),(\emptyset,0)\rangle\}.$$

Note that we have omitted the sequence brackets $\langle\cdot\rangle$ for the utilization sequences to improve readability. There is a one-to-one correspondence between partial solutions and characteristics, which takes care of correctness (a solution for each characteristic) and completeness (a characteristic for each solution), hence $\mathcal{C}_x$ is a full set of characteristics at Start node x.

Introduce Nodes

Introduce nodes x with child y and introduced vertex v take each characteristic C_y produced at y and iterate through combinations of adding v to a range of bags in C_y (see Figure 9 for a depiction of this operation). The first and last bags into which v is put are split into an inner copy with v and an outer copy without v. Furthermore, the utilization sequences within the range are incremented to reflect the new vertex; the utilization sequence of each boundary bag is split in all possible ways into two utilization sequences, which go with the two copies of the boundary bag. The sequence element at which the split is performed is included at the end of the first sequence and the beginning of the second sequence, and the sequence in v's range is incremented. Each resulting C_x must pass two checks in order not to be discarded: All edges between v and some other node of the subgraph G_x must be covered; otherwise, C_x is not a valid path decomposition of B_x and thus

cannot be a characteristic of a path decomposition of G_x. And secondly, none of the utilization values may exceed the upper limit of $\ell+1$. Correctness and completeness of this operation are proved by induction on the tree, taking as induction hypothesis that correctness and completeness hold for the child: For each C_x, there is a C_y from which C_x was constructed; by induction, there exists a normalized S_y with characteristic C_y. By the definition of the preliminary characteristic, there is a one-to-one correspondence between utilization values u_i in C_y and bags in the path decomposition S_y. Therefore repeating two reduced bags in C_y and adding v to a range of reduced bags induces an equivalent operation on S_y, yielding a sequence of bags that we call S_x. In S_x, all edges of G_x are covered and vertices occur only in contiguous ranges, either because of the corresponding property of S_y or because of the way v was added. Hence S_x is a path decomposition of G_x. Moreover, S_x obviously has characteristic C_x, which completes the proof of correctness: For each characteristic C_x at x, there exists a path decomposition S_x of G_x.

To prove completeness, we must show that the characteristic of every partial solution S_x is computed, given every characteristic at y. The restriction of any S_x to G_y—formed by removing the new vertex v—is a partial solution at y, which we call S_y. By the induction hypothesis, we know that the characteristic C_y of S_y is computed at y. As outlined in Figure 10, we will show

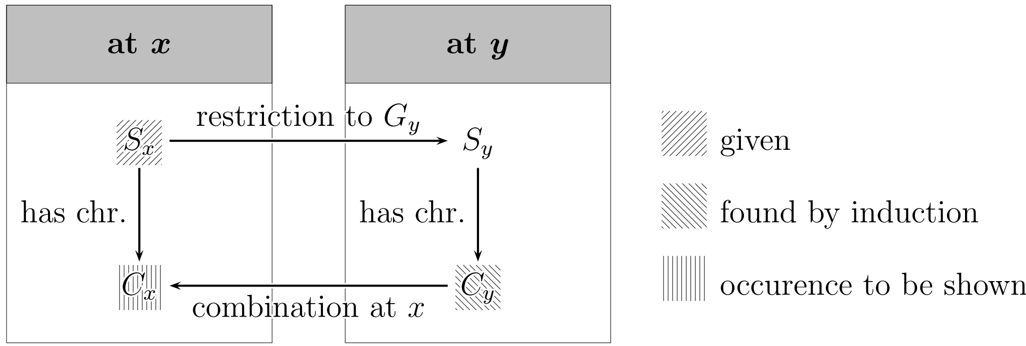


Figure 10: The approach taken by the completeness proofs

that the combination procedure on input C_y will produce the characteristic C_x of S_x. Without loss of generality, we may assume that in S_x, consecutive bags differ in exactly one vertex. Removing v from the first bag in which it appears makes this bag equal to its predecessor. Likewise, the last bag in which v appears coincides with its successor when v is deleted. Thus S_y has repeated bags at the beginning and at the end of the range into which v is inserted to get S_x, but all other bags still differ in exactly one vertex. In the characteristic C_y of S_y, these repeated bags get contracted, but none else.

The combination algorithm checks for all ranges of reduced bags whether inserting v will cover all edges between v and other vertices in B_x, so it will also consider adding v to from the first contracted to the last contracted bag. For this choice of first and last bags, all edges will be covered because S_x is a valid path decomposition. The combination algorithm then duplicates the first and last bag, thus undoing the contraction, and inserts v into the interior of the range. This gives the characteristic C_x of S_x and we have shown the completeness of the Introduce node combination algorithm.

Forget Nodes

The combination algorithm for Forget nodes x with child y and forgotten vertex v transforms characteristics C_y into characteristics C_x by removing v from all bags in C_y, deleting repeated bags, and concatenating their utilization sequences. Correctness: Given a characteristic C_x, there exists a C_y from which C_x was constructed. By induction, there is a partial solution S_y on G_y with characteristic C_y. Since $G_x = G_y$, S_y is also a partial solution of G_x; hence for each C_x, there is a partial solution. As for completeness, we follow again the outline of Figure 10; any partial solution S_x with characteristic C_x is also a partial solution at y, hence the characteristic C_y of $S_y = S_x$ is computed at y. Performing the Forget node algorithm on C_y yields a characteristic $\tilde{C}_x$ of S_x, and since characteristics are unique, $\tilde{C}_x = C_x$.

Join Nodes

Let x be a Join node with children y and z; remember, $B_x = B_y = B_z$ for Join nodes. Combination of characteristics C_y and C_z at node y and z, respectively, will only be attempted when their reduced bag sequences coincide. By the interference property of tree decompositions, vertices shared by partial solutions S_y and S_z are in B_x, therefore utilization values beyond the size of the reduced bag in C_y and C_z refer to *different* forgotten vertices and thus must be added. Even when the reduced bag sequences of C_y and C_z are equal, the corresponding utilization sequences in C_y and C_z do not necessarily have the same length. We can bring two utilization sequences to the same length by repeating some of the utilization values. This corresponds to repeating bags in partial solutions, an operation that maintains the path-decomposition property. Each way of expanding every pair of utilization sequences in $C_y = \langle(\bar{B}_j, \langle u_{y,j,1}, \ldots, u_{y,j,n_{y,j}}\rangle)\rangle_{1\leq j\leq m_y}$ and $C_z = \langle(\bar{B}_j, \langle u_{z,j,1}, \ldots, u_{z,j,n_{z,j}}\rangle)\rangle_{1\leq j\leq m_z}$ to the same length gives rise to a candidate C_x of a characteristic at x: C_x has the same reduced bag sequence as C_y and C_z, and its utilization sequences are formed by summing the ex-

panded sequences of C_y and C_z element by element and subtracting the size of the corresponding bag, which would otherwise be counted twice. Hence we have

$$C_x = \langle(\bar{B}_j, \langle u^*_{y,j,1} + u^*_{z,j,1} - |\bar{B}_j|, \ldots, u^*_{y,j,n_j} + u^*_{z,j,n_j} - |\bar{B}_j|\rangle)\rangle_{1\le j\le m_y}$$

where the sequences $\langle u^*_{y,j,i}\rangle_{1\le i\le n_j}$ derive from $\langle u_{y,j,i}\rangle_{1\le i\le n_{y,j}}$ by repeating elements, and the $\langle u^*_{z,j,i}\rangle_{1\le i\le n_j}$ from $\langle u_{z,j,i}\rangle_{1\le i\le n_{z,j}}$. If a candidate C_x has all utilization values bounded by $\ell + 1$, it is inserted into the output set and discarded otherwise.

Let us consider the correctness of this algorithm: Given C_x, there are characteristics C_y at y and C_z at z from which C_x was built. Let S_y and S_z be the corresponding partial solutions, which exist by the induction hypothesis. We can merge S_y and S_z by first repeating bags according to the expansions of the utilization sequences, and then computing the pairwise union. The resulting S_x is a path decomposition of G_x: each edge is covered, and each vertex only occurs in a contiguous range of bags. Its width is bounded by ℓ since the utilization sequences accurately reflect the bag sizes in S_y and S_z.

To prove completeness, we start from any partial solution S_x, which can be restricted to G_y and G_z giving partial solutions S_y and S_z. By induction, the characteristics C_y of S_y and C_z of S_z are computed at y and z; the Join-node algorithm combines them, creating as output a set of characteristics $\mathcal{C} = \{\tilde{C}_{x,i}\}_{i\in I}$, among which must be the characteristic C_x of S_x. Because the restrictions of S_x, S_y, and S_z to $B_x = B_y = B_z$ are identical, C_x, C_y, and C_z have the same reduced bag sequences, so we only have to show that the utilization sequences of C_x can be built by expanding and summing corresponding sequences of C_y and C_z. Expansion is needed when S_x restricted to G_y or G_z contains repeated bags, which are removed in computing the characteristics C_y and C_z. The expansion of the utilization sequences that corresponds to restoring the deleted repeated bags leads to a C'_x that accurately reflects the bag sizes of S_x, hence $C_x = C'_x \in \mathcal{C}$.

This completes the construction of combination procedures for computing path decompositions of G of width at most ℓ using "preliminary" characteristics. From the characteristic C_{root} at the root and from the characteristic that was used at each other node to construct the characteristic of the parent, we can derive a path decomposition S_{root} of $G_{\text{root}} = G$ by executing the insert and merge operations that were imitated by combining preliminary characteristics. Indeed, computing preliminary characteristics C_x instead of entire partial solutions S_x, as on page 33, did not cause much change to the combination algorithms because the relevant information for combination—the structure of the restriction of S_x to $G[B_x]$ and the original bag sizes—were

conserved in the characteristic. As we saw earlier, size and number of reduced bag sequences are independent of the number n of vertices in G, in contrast to the utilization sequences, whose total length equals the number of bags of the characterized path decomposition and is therefore linear in n.

We already observed that substituting the maximum utilization for each utilization sequence falls short with regard to completeness. Since we require that no characteristic being computed at the root node implies that the graph has pathwidth greater than ℓ, we have to find a way to reduce the size of the characteristics without sacrificing completeness. Any attempt of going from one utilization value to a fixed-length sequence—say three values, one for the first element, the greatest element, and the last element of the actual utilization sequence—is doomed as well: In the next section, we will give a class $\mathcal{T}$ of $\Omega(2^\ell)$ utilization sequences and show that to achieve correctness and completeness, they must map to distinct compressed utilization representations. However, with a fixed number of values in the range $0, \ldots, \ell+1$, we cannot represent 2^ℓ objects. After this result of our own, we resume the construction by Bodlaender and Kloks and show that representing arbitrary utilization sequences by elements of $\mathcal{T}$ is sufficient and that the size of $\mathcal{T}$ is in $\mathrm{O}(2^{2\ell})$, that is, independent of the number of vertices n.

3.4 Compressing Utilization Sequences

In the following, $\mathcal{U}$ will denote the class of finite sequences of nonnegative integers, which we call utilization sequences; $\mathcal{U}_\ell$ stands for the $\mathcal{U}$-sequences with elements in the range $0,\ldots,\ell+1$. We define a subset $\mathcal{T}$ of utilization sequences and its restriction $\mathcal{T}_\ell$ to $\mathcal{U}_\ell$: The defining property of sequences $\tau \in \mathcal{T}$ is that between any two non-consecutive sequence elements, there is an element that is either greater or smaller than both of them. For example, $\langle 1, 5, 3, 4\rangle$ conforms to this condition, whereas $\langle 1, 3, 5, 4\rangle$ does not because $1 \leq 3 \leq 5$. Let us derive a bound on the number of such $\tau \in \mathcal{T}_\ell$. For integers $0 \leq u_1 < u_2 < \ldots < u_s \leq \ell+1$, the sequence $\langle u_1, u_s, u_2, u_{s-1}, u_3, u_{s-2}, \ldots\rangle$ is in $\mathcal{T}_\ell$. There are $2^{\ell+2} - 1$ ways to choose non-empty subsets $\{u_1, \ldots, u_s\}$ from $\{0, \ldots, \ell+1\}$, and each choice leads to a different sequence, therefore $|\mathcal{T}_\ell| = \Omega(2^\ell)$. On the other hand, it can be shown that every $\mathcal{T}_\ell$-sequence that starts with its minimum is of the above form, and every $\mathcal{T}_\ell$-sequence starting with its maximum is of the form $\langle u_s, u_1, u_{s-1}, u_2, u_{s-2}, u_3, \ldots\rangle$. In every $\mathcal{T}$-sequence, either the maximum or the minimum occurs only once; if we split a sequence at this hinge element so that it ends up in both parts, we get a right part, which is in one of the forms above, and a left part, whose reverse is in this form. Thus we have reduced counting the number of $\mathcal{T}_\ell$-sequences

to counting subsets of $\{0, \dots, \ell+1\}$. A detailed calculation leads to a precise count of $T_\ell := \frac{32}{3}4^\ell - \frac{2}{3} = \Theta(2^{2\ell})$.

To see why in compressing utilization sequences of preliminary characteristics, two sequences from $\mathcal{T}$ must never have the same compressed image, we need to introduce a little apparatus. A sequence $\alpha \in \mathcal{U}$ can be expanded by repeating elements; expansions will be denoted by an asterisk, e.g., a possible expansion of $\alpha = \langle 1, 3, 2, 2, 5 \rangle$ is $\alpha^* = \langle 1, 3, 3, 2, 2, 2, 5, 5 \rangle$. Remember that those numbers stand for bag sizes, and bags in a path decomposition can be repeated without destroying the path decomposition; moreover, repeating bags and utilization values is necessary in merging partial solutions and characteristics. We write $\alpha \leq \beta$ when α and β have the same length and each element a_i of α is at most as great as the corresponding element b_i of β. We extend $\leq$ to a partial order $\preccurlyeq$ on sequences of different length: $\alpha \preccurlyeq \beta$ shall hold if there exist expansions α^* and β^* of the same length with $\alpha^* \leq \beta^*$. Informally, $\alpha \preccurlyeq \beta$ expresses that merging operations that work with β also work with α. Equivalence with respect to the combination operations is conveyed by the equivalence relation $\asymp$: We set $\alpha \asymp \beta$ if and only if $\alpha \preccurlyeq \beta$ and $\alpha \succcurlyeq \beta$. Actual merging is reflected in the addition operation; for expansions of the same length, $\alpha^* + \beta^*$ is the pairwise sum of the sequences, and $\alpha \oplus \beta$ is the set of the sums of all expansions of common length. One criterion of the "quality" of a utilization sequence α is its maximum $\max \alpha$, the value of its greatest element. The maximum matters, for example, at Introduce nodes, where a new vertex is added to a range of bags and we must ensure that the maximal bag utilization does not exceed $\ell + 1$. At Join nodes, the best fit of two sequences with respect to the maximum utilization value is measured by

$$\min\max(\alpha \oplus \beta) := \min\{\max(\alpha^* + \beta^*) : \alpha^*, \beta^* \text{ same-length expansions}\}$$

The following two lemmas help to establish that fixed-size utilization representations cannot exist. We first claim that sequences α and β indistinguishable by $\min\max(\cdot \oplus \gamma)$ are equivalent and then argue that distinct $\mathcal{T}$-sequences are never equivalent; Theorem 19 summarizes the conclusion that distinct $\mathcal{T}$-sequences are distinguishable.

Lemma 17. Let α and β be utilization sequences. If for all utilization sequences γ, $\min\max(\alpha \oplus \gamma) = \min\max(\beta \oplus \gamma)$, then $\alpha \asymp \beta$.

Lemma 18. For $\sigma, \tau \in \mathcal{T}$, $\sigma \asymp \tau$ implies $\sigma = \tau$.

Putting together the contrapositions of Lemma 18 and 17 yields

Theorem 19. For $\sigma, \tau \in \mathcal{T}$, if $\sigma \neq \tau$, then there is a utilization sequence $\gamma \in \mathcal{U}$ with $\min\max(\sigma \oplus \gamma) \neq \min\max(\tau \oplus \gamma)$. □

Any compressed utilization sequence must contain information about the maximum utilization of the represented utilization sequences, since this information is necessary to know whether at an Introduce node a vertex can be inserted into the whole range of bags that have the same projection $\bar{B}_j$. Only utilization sequences with the same maximum can map to the same compressed representation; otherwise, for correctness, the greater of the values had to determine the maximum stored in the representation, defeating completeness: The characteristic claims that less vertices can be added than for which actually is room. Therefore Theorem 19 means that each element of $\mathcal{T}_\ell$ must be projected to a different representation, bloating their number to $\Omega(2^\ell)$, beyond the capacity of a fixed number of utilization values. Before proceeding, we give proofs of the preceding lemmas.

Proof of 17. Let $\alpha = \langle a_i \rangle_i$, $A = \max \alpha$ and $\gamma = \langle A - a_i \rangle_i$. Then $\min \max(\alpha \oplus \gamma) = A$. If $\min \max(\beta \oplus \gamma) \leq A$, then $\beta \preccurlyeq \alpha$. Switching the role of α and β yields $\alpha \preccurlyeq \beta$ and therefore $\alpha \asymp \beta$. □

Proof of 18. We show that there are expansions σ^* and τ^* with $\sigma^* = \tau^*$. Undoing the repetition of values, we then get $\sigma = \tau$. Let $L := \mathrm{len}(\sigma) + \mathrm{len}(\tau)$. Note that

(1) because of $\sigma \asymp \tau$, we have $\sigma^* \asymp \tau^*$ for any expansions σ^* and τ^*,

(2) for expansions $\sigma^* = \langle s_i^* \rangle_i$ and $\tau^* = \langle t_i^* \rangle_i$ with

$$\mathrm{len}(\sigma^*) = \mathrm{len}(\tau^*) > L,$$

there must be positions where elements have been repeated both in σ^* and τ^*, i.e., there is an index i with $s_i = s_{i-1}$ and $t_i = t_{i-1}$.

We construct inequalities of expansions σ_i, τ_i,

$$\sigma_1 \leq \tau_1 \leq \sigma_2 \leq \tau_2 \leq \ldots \leq \sigma_j \leq \tau_j$$

invoking (1) repeatedly and expanding earlier σ_i, τ_i to the length of the latest pair. By (2) we can assume that all σ_i, τ_i have length L. Eventually, equality must hold because there is only a finite number of expansions of σ and τ of length L. □

We now move along to prove that representing utilization sequences with a maximum of at most $\ell+1$ by $\mathcal{T}_\ell$-sequences yields a linear-time algorithm for path decomposition. To do so, we introduce the projection $\tau : \mathcal{U} \to \mathcal{T}$, use it to define the final characteristics, adjust the combination algorithms and extend the correctness and completeness proofs. We have already seen that

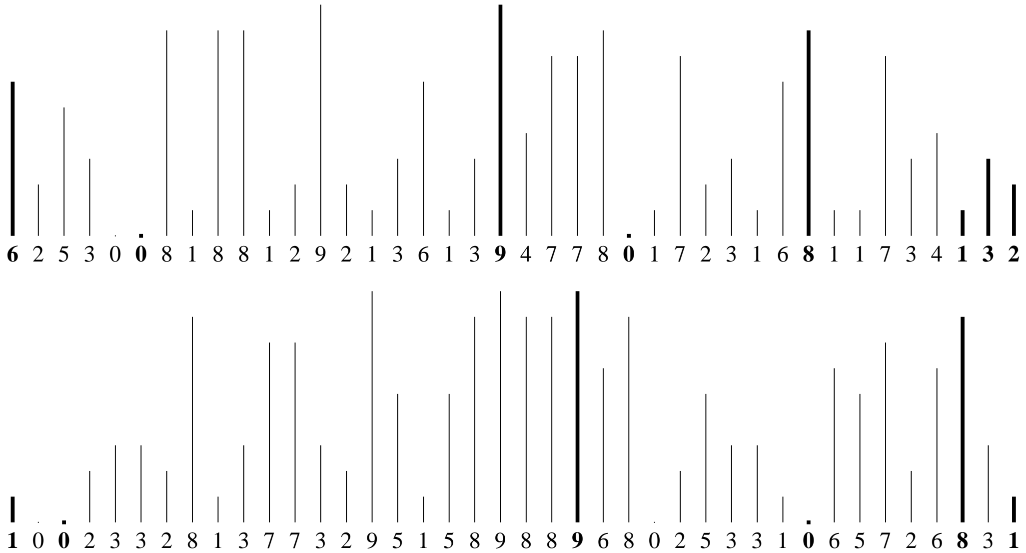


Figure 11: Two utilization sequences and their $\mathcal{T}$-projection $\tau(\cdot)$ (bold). Note that the ranges to be deleted are not uniquely determined.

$\alpha \asymp \beta$ reflects one aspect of similarity between utilization sequences α and β, namely, the behavior under min max tests. We will show that $\alpha \asymp \beta$ implies that we can interchange α and β in any characteristic without sacrificing the correctness or completeness of the combination algorithms. Hence we can safely replace utilization sequences α by small representatives $\tau(\alpha)$ of the equivalence class $[\alpha]_\asymp$. It is a natural choice to choose $\tau(\alpha)$ from $\mathcal{T}$, since those sequences have been shown to be mutually inequivalent. Moreover, we will see that for each α with $\max\alpha \leq \ell + 1$, there is a $\tau(\alpha) \in \mathcal{T}_\ell$ with $\tau(\alpha) \asymp \alpha$. Therefore we can uniquely represent any valid utilization sequence using $\mathcal{T}_\ell$.

For $\alpha = \langle a_i \rangle_i$, $\tau(\alpha)$ is defined by repeatedly deleting offending ranges of elements in α until the definition of a $\mathcal{T}$-sequence is met: While there are indices i and j with $i < j$ so that for *all* elements a_k between i and j holds $\min\{a_i, a_j\} \leq a_k \leq \max\{a_i, a_j\}$, remove all elements between i and j (i.e., the a_k with $i < k < j$, see Figure 11 for examples). Obviously, $\tau(\alpha) \in \mathcal{T}$. We can extend $\tau(\alpha)$ to the length of α by repeating the greater boundary of each deleted range, leading to a $(\tau(\alpha))^*$ with $(\tau(\alpha))^* \geq \alpha$, so $\tau(\alpha) \succcurlyeq \alpha$. Similarly, we can construct a lower bound $(\tau(\alpha))_*$ with $(\tau(\alpha))_* \leq \alpha$, hence $\tau(\alpha) \preccurlyeq \alpha$ and $\tau(\alpha) \asymp \alpha$. As a side effect of this relation, we get that $\tau(\alpha)$ is well-defined: If α gets reduced by different deletions to $\sigma \in \mathcal{T}$ and $\tau \in \mathcal{T}$, then $\sigma \asymp \alpha \asymp \tau$, so by Lemma 18, $\sigma = \tau$.

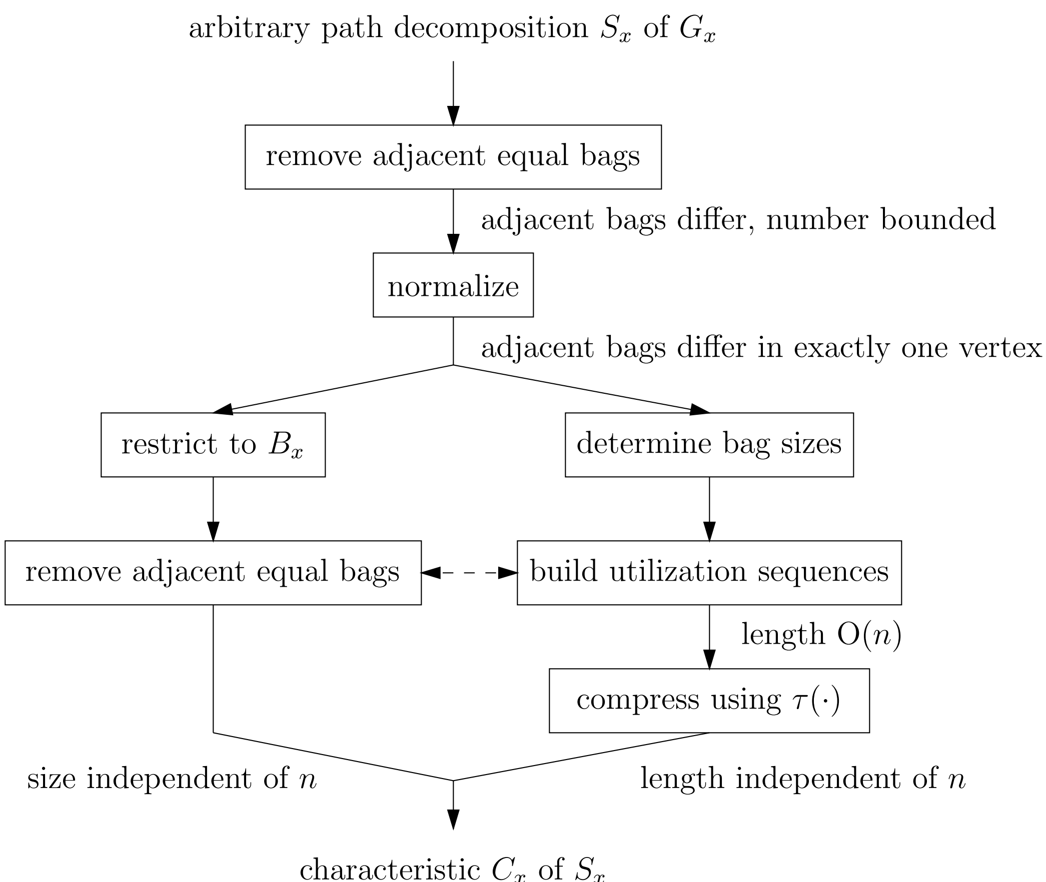


Figure 12: The final characteristic

As a compression operation, $\tau(\cdot)$ is the last ingredient of the final characteristic. Figure 12 shows how $\tau(\cdot)$ fits into the procedure for computing the unique characteristic of a partial solution S_x in the subgraph G_x of node x; note that from now on, C_x will denote a final characteristic of a path decomposition of subgraph G_x. $C_x = \langle(\bar{B}_j, \tau_j)\rangle_{1\leq j\leq m'}$ consists of a sequence of reduced bags $\bar{B}_j$ and $\mathcal{T}_\ell$-sequences τ_j with the essential information about the sizes of the bags $\hat{B}_i$ of $S_x = \langle\hat{B}_i\rangle_{1\leq i\leq m}$ that are reduced to $\bar{B}_j$. What does C_x tell us about S_x? Every pair of consecutive utilization values $t_{j,p}$, $t_{j,p+1}$ in sequence τ_j corresponds to a contiguous range of bags $\langle\hat{B}_i\rangle_{q\leq i\leq q'}$, which have intersection $\bar{B}_j$ with B_x. About the sizes of those $\langle\hat{B}_i\rangle_{q\leq i\leq q'}$, we know the precise number of vertices in the first and last bag: $|\hat{B}_q| = t_{j,p}$ and $|\hat{B}_{q'}| = t_{j,p+1}$. Furthermore, due to $\tau_j \in \mathcal{T}$, the sizes of the bags between q and q' vary only between $\min\{t_{j,p}, t_{j,p+1}\}$ and $\max\{t_{j,p}, t_{j,p+1}\}$.

3.5 The Final Characteristic at Work

By completeness, we previously understood that for any partial solution S_x at a tree node x, the combination algorithm will compute its characteristic C_x. To succeed in proving completeness with final characteristics, we need to relax this requirement so that for any S_x, C_x does not need to be computed, but there exists at least some "better" partial solution S'_x for which a characteristic C'_x is computed at x. This is still sufficient to guarantee that whenever a solution exists on the entire graph, a characteristic of one is really found—indeed, it would suffice to prove that whenever a partial solution at x exists, any characteristic at x is computed at all. Earlier we argued that for utilization sequences, $\alpha \preccurlyeq \beta$ implies that α can be used wherever β fits; building on this, we write $C'_x \preccurlyeq C_x$ if C_x and C'_x have the same reduced bag sequence and if for each reduced bag, the associated compressed utilization sequences satisfy $\tau'_j \preccurlyeq \tau_j$. In this case, C'_x subsumes C_x, and our new notion of completeness means that for the partial solution S_x with characteristic C_x, some C'_x with $C'_x \preccurlyeq C_x$ is computed; Figure 13 shows the impact on the completeness proofs for combination procedures.

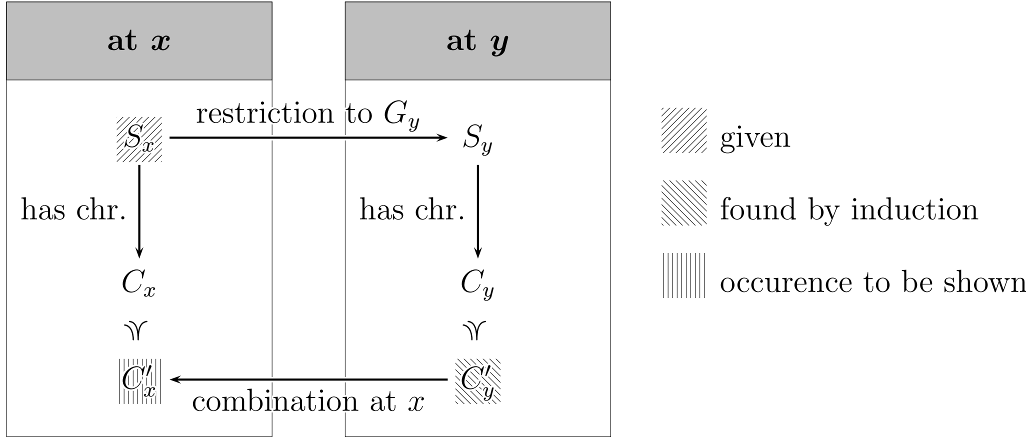


Figure 13: The revised approach for completeness proofs. Compared to Figure 10, the characteristics of the solutions are replaced by "better" characteristics. The diagram shows the case where x has one child; for Join nodes, there are accordingly two restricted solutions S_y and S_z.

With regard to the full set of characteristics, the consequence of the shift is that full sets are no longer unique, though there still is a minimal full set. Observe that if a combination procedure at some node x produces characteristics C'_x and C''_x with $C''_x \succcurlyeq C'_x$, we may in fact discard C''_x. In contrast

to the preceding subsequent refinements of the characteristics, this is a property of the entire (full) set of characteristics at a tree node x. Whether redundant characteristics are eliminated at each node or not does not affect the desired linear running time, because even without elimination, the number of characteristics is independent of n. However, redundancy in the full set will be addressed in detail when we discuss the implementation of the path-decomposition algorithm.

In the following, we will iterate one last time over the four tree-node types, giving combination algorithms and proving correctness and completeness. From time to time, it might be useful to skip ahead to Figures 18 and 20 on pages 61 and 63 to see how the combination algorithms work on concrete characteristics.

Start and Introduce Nodes

Of the four node types, only Start nodes behave exactly as before: They produce the four characteristics

$$\langle(\{v\},1)\rangle,\ \langle(\emptyset,0),(\{v\},1)\rangle,\ \langle(\{v\},1),(\emptyset,0)\rangle \text{ and } \langle(\emptyset,0),(\{v\},1),(\emptyset,0)\rangle$$

whose utilization sequences are already $\mathcal{T}$-sequences. No further work is needed to show correctness and completeness. Note that we can do without the first three characteristics, because every path decomposition building on them can be extended to contain empty bags in front or in the back.

As for Introduce nodes x with child y, our approach is exactly as before: For every input characteristic $C_y = \langle(\bar{B}_j, \tau_j)\rangle_{1\le j\le m'}$,

- we determine all possible ranges $\langle \bar{B}_j\rangle_{q\le j\le q'}$ into which the new vertex v can be put so that all edges between v and other vertices in B_x are covered,

- split the boundary sequences $\tau_q = t_1,\dots,t_p$ and $\tau_{q'} = t'_1,\dots,t'_{p'}$ at all positions i, i' into

$$\tau_{q,\text{left}} = t_1,\dots,t_i \qquad \tau_{q,\text{right}} = t_i,\dots,t_p$$
$$\tau_{q',\text{left}} = t'_1,\dots,t'_{i'} \qquad \tau_{q',\text{right}} = t'_{i'},\dots,t'_{p'}$$

- and increment the elements of the sequences associated with bags inside the range of v.

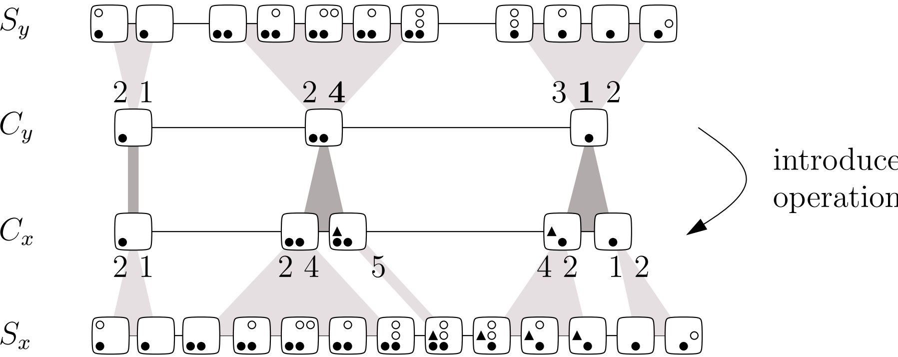


Figure 14: Example of deriving a partial solution at an Introduce node. The chosen range for the new vertex ▲ starts at the second bag of C_y and ends at the third; at the second bag, the utilization sequence is split at 4, while at the third, the split is performed at 1.

The steps lead to several

$$
\begin{array}{ll}
C_x = \langle\ (\bar{B}_1, \tau_1), \dots, (\bar{B}_{q-1}, \tau_{q-1}), & \\
\qquad (\bar{B}_q, \tau_{q,\text{left}}), (\bar{B}_q \cup \{v\}, \tau_{q,\text{right}} + 1), & \text{beginning,} \\
\qquad (\bar{B}_{q+1} \cup \{v\}, \tau_{q+1} + 1), \dots, (\bar{B}_{q'-1} \cup \{v\}, \tau_{q-1} + 1), & \text{interior,} \\
\qquad (\bar{B}_{q'} \cup \{v\}, \tau_{q',\text{left}} + 1), (\bar{B}_{q'}, \tau_{q',\text{right}}), & \text{end of range} \\
\qquad (\bar{B}_{q'+1}, \tau_{q'+1}), \dots, (\bar{B}_{m'}, \tau_{m'})\ \rangle &
\end{array}
$$

and again we discard any C_x where the maximum of any τ_j exceeds $\ell + 1$. Splitting as well as adding constants to $\mathcal{T}$-sequences gives $\mathcal{T}$-sequences, so the procedure maps final characteristics to final characteristics. Figure 14 shows an example of how we can reconstruct a partial solution S_x from a characteristic C_x: We assume that at tree node y, we have a partial solution S_y for C_y (which is the characteristic from which C_x was computed). We locate in S_y the two bags that correspond to the splits performed to get from C_y to C_x, duplicate those bags and insert v into the range of bags that correspond to the incremented utilization values. This yields a path decomposition S_x of G_x; it has width at most ℓ, which completes the induction argument for correctness.

Now we start from some partial solution S_x at x and prove that the combination algorithm computes a characteristic C'_x that subsumes the characteristic C_x of S_x. We assume that in S_x, consecutive bags differ in exactly one vertex. Let S_y be the restriction of S_x to G_y; by the induction hypothesis, we know that at tree node y, either the characteristic C_y of S_y or some C'_y with $C'_y \preccurlyeq C_y$ is computed. For preliminary characteristics, we had a one-to-one correspondence between bags and utilization values, so that inserting v into

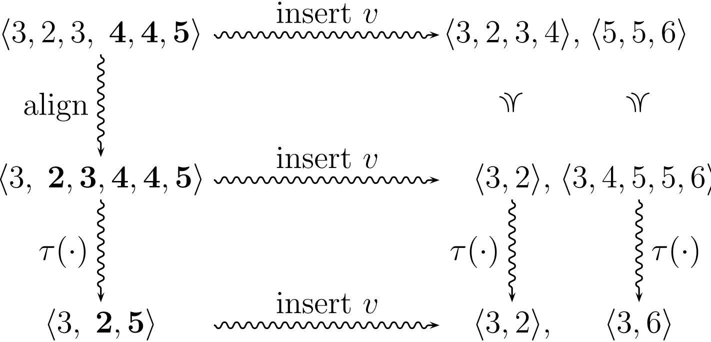


Figure 15: Moving misaligned insertion limits. The figure shows utilization sequences of a solution and the corresponding $\mathcal{T}$-sequence of its characteristic. Note that the sequence element at the split position is repeated, and that the $\mathcal{T}$-sequence from C_y was split into two $\mathcal{T}$-sequences in $\tilde{C}_x$.

a partial solution had a counterpart in adding it to a characteristic and vice versa. For the final characteristic, we have to specify what happens if the utilization value of a boundary bag of the range of v in S_y is eliminated in C_y: We change in S_y the range of v by moving in S_y any "misaligned" insertion limit to the bag that corresponds to the next lower value in the compressed utilization sequence (Figure 15). This results in splits where on both sides, the utilization sequence is not greater than the corresponding sequence at the old split position. Inserting v into the aligned range yields a partial solution $\tilde{S}_x$. For its characteristic $\tilde{C}_x$, we have $\tilde{C}_x \preccurlyeq C_x$ and on input C_y, the combination algorithm does find $\tilde{C}_x$ (Figure 16). If C_y is produced at tree node y, we are done; in general, however, merely a characteristic C'_y with $C'_y \preccurlyeq C_y$ is computed at y. Since $C'_y \preccurlyeq C_y$, we can expand the compressed utilization sequences in $C'_y = \langle(\bar{B}_j, \tau_j)\rangle_{1\leq j\leq m'}$ and $C_y = \langle(\bar{B}_j, \sigma_j)\rangle_{1\leq j\leq m'}$ so that for all j, we have $\tau_j^* \leq \sigma_j^*$. Adding v to C_y induces a way of adding v to the expansion of C_y and hence to the expansion of C'_y; adding v to the expansion of C'_y induces a way of adding v to C'_y, yielding a characteristic C'_x with $C'_x \preccurlyeq \tilde{C}_x \preccurlyeq C_x$. This concludes the completeness proof for the Introduce-node combination operation.

Forget Nodes

Forget nodes cause little trouble. Let x be a Forget node with child y, and let v be the forgotten vertex. The combination operation for final characteristics is a straightforward extension from the same procedure for preliminary characteristics: We transform every input C_y to an output C_x

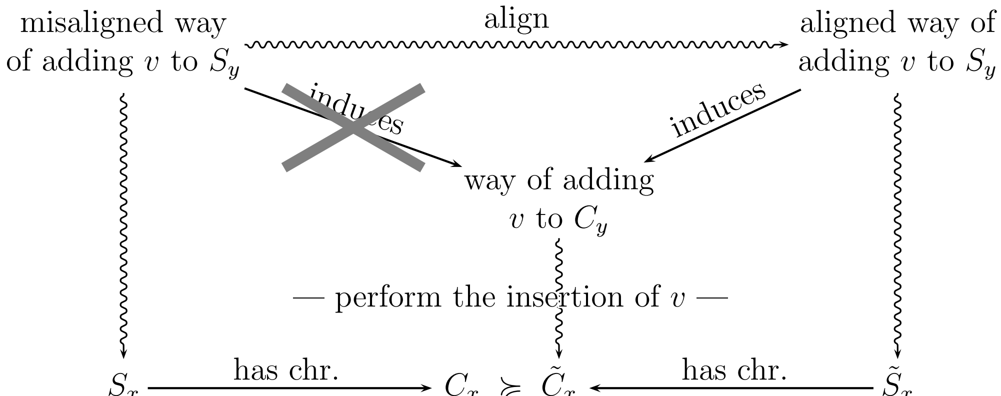


Figure 16: Outline of showing how for S_x with characteristic C_x, a $\tilde{C}_x$ with $\tilde{C}_x \preccurlyeq C_x$ is found that gets computed from characteristic C_y. S_y is the restriction of S_x to G_y and C_y is the characteristic of S_y.

(1) by removing v from the reduced bag sequence,

(2) removing the two repeated bags,

(3) concatenating the corresponding compressed utilization sequences,

(4) and recompressing them.

For $C_y = \langle(\bar{B}_j, \tau_j)\rangle_{1\leq j\leq m'}$ where v is in the bags q through q', C_x takes the form

$$\begin{aligned} C_x = \langle\ & (\bar{B}_1, \tau_1), \ldots, (\bar{B}_{q-2}, \tau_{q-2}), & \\ & (\bar{B}_{q-1}, \tau(\tau_{q-1} \circ \tau_q)), & \text{beginning,} \\ & (\bar{B}_{q+1} \setminus \{v\}, \tau_{q+1}), \ldots, (\bar{B}_{q'-1} \setminus \{v\}, \tau_{q'-1}), & \text{interior,} \\ & (\bar{B}_{q'+1}, \tau(\tau_{q'} \circ \tau_{q'+1})), & \text{end of } v\text{'s range} \\ & (\bar{B}_{q'+2}, \tau_{q'+2}), \ldots, (\bar{B}_{m'}, \tau_{m'})\ \rangle & \end{aligned}$$

where $\circ$ stands for sequence concatenation and $\tau(\cdot)$ is the projection to $\mathcal{T}$. Note that due to the normalization, $\bar{B}_{q-1} = \bar{B}_q \setminus \{v\}$ and $\bar{B}_{q'} \setminus \{v\} = \bar{B}_{q'+1}$. Now take any C_x produced at x from C_y at y. By induction, there is a partial solution at y with characteristic C_y. Since $G_x = G_y$, this partial solution is also a partial solution at x that has characteristic C_x. Thus we have shown that the combination is correct; completeness relies on the fact that for arbitrary utilization sequences α, β, γ, δ we have

$$\alpha \preccurlyeq \beta \text{ and } \gamma \preccurlyeq \delta \Longrightarrow \tau(\alpha \circ \gamma) \preccurlyeq \tau(\beta \circ \delta). \qquad (*)$$

Let S_x be the partial solution for which we want to show that a C'_x at most as great as the characteristic C_x of S_x is computed. Again, the restriction S_y to G_y is just S_x itself; for $S_y = S_x$ with characteristic C_y, we know by induction that a characteristic C'_y with $C'_y \preccurlyeq C_y$ is found at y. "Forgetting" v from both C'_y and C_y yields some C'_x and the C_x of S_x; by $(*)$, we have $C'_x \preccurlyeq C_x$, which proves the completeness of the Forget-node combination procedure.

Join Nodes

Finally, let us consider Join nodes. As usual, we denote the Join node by x and its two children by y and z. With preliminary characteristics, we could achieve completeness even though we only merged characteristics C_y and C_z with the same reduced bag sequence. Since preliminary characteristics differ from final characteristics only in the compression of the utilization sequences, we maintain this restriction. Accordingly, we merely need to specify how the $\mathcal{T}$-sequences of C_y and C_z can be added to reflect the bag sizes of a merged partial solution at x. The degree of freedom we have in merging partial solutions S_y and S_z is repeating bags; the corresponding repetition of utilization sequences maps utilization sequences α to expansions α^*. Merging two bags of S_y and S_z is reflected in the utilization by adding their sizes without counting shared vertices twice; merging utilization sequences α_j and β_j associated with bag $\bar{B}_j$ in all possible ways yields the sequences

$$(\alpha_j \oplus \beta_j) - |\bar{B}_j| = \left\{ \left(\alpha_j^* + \beta_j^*\right) - |\bar{B}_j| : \alpha_j^*, \beta_j^* \text{ same-length expansions} \right\}.$$

The utilization sequences in C_y and C_z are compressed and the utilization sequences of C_x must be from $\mathcal{T}$ as well. In general, the sum $\sigma \oplus \tau$ of $\mathcal{T}$-sequences contains elements that are not from $\mathcal{T}$, but applying the compression operation $\tau(\cdot)$ to all elements of $\sigma \oplus \tau$ appears to be a reasonable approach to produce compressed sequences for characteristics at x. Thus, from $C_y = \langle(\bar{B}_j, \sigma_j)\rangle_{1 \le j \le m'}$ and $C_z = \langle(\bar{B}_j, \tau_j)\rangle_{1 \le j \le m'}$, we let the algorithm produce the characteristics

$$C_x = \langle(\bar{B}_j, \rho_j)\rangle_{1 \le j \le m'} \quad \text{with} \quad \rho_j \in \tau\big((\sigma_j \oplus \tau_j) - |\bar{B}_j|\big) \ \text{ for } 1 \le j \le m'. \qquad (**)$$

Of course, every combination of choosing the ρ_j gives rise to one C_x, and all C_x with $\max \rho_j > \ell + 1$ for any j are weeded out. To convince ourselves that for a C_x thus computed, we can find a matching partial solution S_x, we rely once more on the induction hypothesis that partial solutions S_y on G_y and S_z on G_z with characteristics C_y and C_z exist. We derive instructions from C_x, C_y, and C_z on how to merge S_y and S_z: Let α_j and β_j be the uncompressed utilization sequences of S_y and S_z, so $\sigma_j = \tau(\alpha_j)$ and $\tau_j = \tau(\beta_j)$. Since

$\sigma_j \asymp \alpha_j$ and $\tau_j \asymp \beta_j$, we can choose expansions σ_j^* and τ_j^* that satisfy $\sigma_j^* \geq \alpha_j$ and $\tau_j^* \geq \beta_j$. In σ_j^*, we can identify each value in the sequence with a bag of S_y of at most that size; the same holds for τ_j^* and S_z. By $(**)$, we know that ρ_j originates from particular expansions σ_j^+ and τ_j^+ of σ_j and τ_j with $\rho_j = \tau(\sigma_j^+ + \tau_j^+ - |\bar{B}_j|)$; we would like to translate these expansion steps to expansions of σ_j^* and τ_j^*, because operations on the latter have a counterpart in operations on bags of S_y and S_z. Therefore we compute σ_j^{*+} as in Figure 17 by repeating the expanded ranges of σ_j^* just as the elements

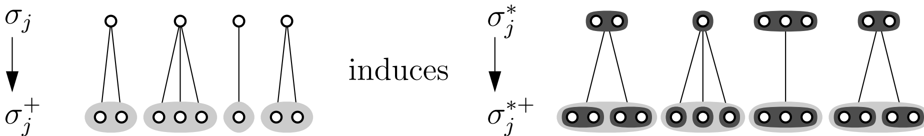


Figure 17: Imitating the sum expansion of σ_j with σ_j^*.

of σ_j were repeated in producing σ_j^+. By the same means, we expand τ_j^* to τ_j^{*+} using the expansion from τ_j to τ_j^+ as a model. Then we have

$$\tau\big(\sigma_j^{*+} + \tau_j^{*+} - |\bar{B}_j|\big) = \rho_j,$$

which means that if we can construct a partial solution at x with utilization sequences $\sigma_j^{*+} + \tau_j^{*+} - |\bar{B}_j|$, we have accomplished our goal. Through the same expansions that take the σ_j^* to σ_j^{*+}, we expand S_y to S_y^+ with utilization sequences α_j^+ satisfying $\alpha_j^+ \leq \sigma_j^{*+}$. By the same means, we obtain S_z^+ with utilization sequences β_j^+, where $\beta_j^+ \leq \tau_j^{*+}$. Merging S_y^+ and S_z^+ bag by bag, we get a path decomposition $\tilde{S}_x$ of G_x with utilization sequences

$$\gamma_j = \alpha_j^+ + \beta_j^+ - |\bar{B}_j| \leq \sigma_j^{*+} + \tau_j^{*+} - |\bar{B}_j|.$$

Clearly, we have nearly obtained the desired result—proving the existence of a partial solution S_x with characteristic C_x—but we have to be a little careful in concluding the argument. Looking closely, we see that so far we have only proved that the sequences $\langle \sigma_j^{*+} + \tau_j^{*+} - |\bar{B}_j| \rangle_{1 \leq j \leq m'}$ dominate the utilization sequences of $\tilde{S}_x$; however, some of the compressed utilization sequences $\tau(\gamma_j)$ might be strictly smaller than the corresponding ρ_j of C_x. To obtain an S_x with utilization sequences $\sigma_j^{*+} + \tau_j^{*+} - |\bar{B}_j|$, we enlarge bags in $\tilde{S}_x$ that are too small by taking vertices from larger neighboring bags. The resulting S_x has characteristic C_x because it has the right reduced bag sequence and compressed utilization sequences $\rho_j = \tau(\sigma_j^{*+} + \tau_j^{*+} - |\bar{B}_j|)$.

To recapitulate: we expand the compressed utilization sequences of C_y and C_z to dominate the actual utilization sequences of S_y and S_z, and then

expand them further to reflect the way the compressed utilization sequences of C_x were constructed. These expansions tell us how the bags of S_y and S_z have to be repeated to produce a S_x with the given characteristic C_x.

To prove the completeness of the Join-node combination procedure, we start from a partial solution S_x on G_x and show that the characteristic C_x of S_x or a characteristic C'_x of some better partial solution S'_x is computed. By the now familiar reasoning, we define S_y and S_z to be the restrictions of S_x to G_y and G_z and let C_y and C_z be the characteristics of S_y and S_z; by the induction hypothesis, the tree nodes y and z produce characteristics C'_y and C'_z with $C'_y \preccurlyeq C_y$ and $C'_z \preccurlyeq C_z$. Our task is to exhibit a C'_x that on the one hand is among the output of combining C'_y and C'_z, and on the other hand satisfies $C'_x \preccurlyeq C_x$. Writing $C'_y = \langle(\bar{B}_j, \sigma_j)\rangle_{1\le j\le m'}$ and $C'_z = \langle(\bar{B}_j, \tau_j)\rangle_{1\le j\le m'}$ (C_y and C_z, and hence C'_y and C'_z, must have the same reduced bag sequence), we know that there are, for each j, expansions σ^*_j and τ^*_j so that $\sigma^*_j \le \alpha_j$ and $\tau^*_j \le \beta_j$, where α_j and β_j are the uncompressed utilization sequences of S_y and S_z. Since S_y and S_z are both restrictions of S_x, they have the same number of bags; hence α_j and β_j have the same length, and we can add σ^*_j and τ^*_j element by element. Defining

$$\rho_j = \tau\big(\sigma^*_j + \tau^*_j - |\bar{B}_j|\big)$$

then will do the job; in other words, $C'_x := \langle(\bar{B}_j, \rho_j)\rangle_{1\le j\le m'}$ will turn out to be a characteristic at x that does get computed by our algorithm and for which $C'_x \preccurlyeq C_x$ holds. When we recall that in the combination algorithm, the candidates for the j-th combination sequence come from $\tau((\sigma_j \oplus \tau_j) - |\bar{B}_j|)$, we see immediately that our C'_x will be produced as an intermediate result. It might get rejected if its maximum utilization exceeds $\ell + 1$, so proving $C'_x \preccurlyeq C_x$ will not only establish that C'_x is a sufficient surrogate for C_x, but also serves to bound the maximum of C'_x. Looking at how S_x results from merging S_y and S_z, we see that the j-th uncompressed utilization sequence of S_x is $\alpha_j + \beta_j - |\bar{B}_j|$, which is lower bounded by $\sigma^*_j + \tau^*_j - |\bar{B}_j|$. The compression step—projecting to $\mathcal{T}$—preserves this inequality, i.e.,

$$\begin{aligned} & \sigma^*_j + \tau^*_j - |\bar{B}_j| \le \alpha_j + \beta_j - |\bar{B}_j| \\ \Longrightarrow \quad & \tau\big(\sigma^*_j + \tau^*_j - |\bar{B}_j|\big) \preccurlyeq \tau\big(\alpha_j + \beta_j - |\bar{B}_j|\big)\,, \end{aligned}$$

therefore ρ_j is smaller than the corresponding compressed utilization sequence of S_x, or $C'_x \preccurlyeq C_x$. As an aside, note that from the correctness proof above follows that the ominous "better" partial solution with characteristic C'_x really exists.

This concludes our description of the [BK96] algorithm for computing path decompositions of graphs of bounded treewidth. Before we refine the

analysis and discuss our implementation, a few remarks on the construction are in order:

Solutions can be computed by recursing on the tree and combining partial solutions of the children by following the correctness proofs. To every characteristic at the root, a solution with this or a smaller characteristic can be found; nonetheless, enumerating all path decompositions given all characteristics at the root requires further effort.

No Solution will be found when, at any tree node, no characteristic is computed. By the completeness of the combination algorithms, this implies that the graph does not have a path decomposition of width ℓ.

Simplicity seems to be lacking in the overall construction of the algorithm. However, we have argued in several places that the present level of complexity cannot be avoided in interfacing to the tree-automaton technique: The reduced bag sequence is necessary to determine the ways a partial solution can be extended and by Theorem 19, further compression of the utilization sequences is impossible.

3.6 Analyzing the Algorithm

How many final characteristics can there be? A characteristic consists of a reduced bag sequence and a $\mathcal{T}_\ell$-sequence for each reduced bag; we know exactly how many $\mathcal{T}_\ell$-sequences there are, but our earlier approximation of the number of reduced bag sequences (on page 34) was rather coarse. To refine it, we recall that consecutive bags $\hat{B}_i$ and $\hat{B}_{i+1}$ of a normalized path decomposition $S_x = \langle \hat{B}_i \rangle_{1 \leq i \leq m}$ of the subgraph G_x at tree node x differ in exactly one vertex. So do any reduced bags $\bar{B}_j$ and $\bar{B}_{j+1}$, which result from restricting the path decomposition to the bag B_x of tree node x and removing consecutive equal sets. We determine by induction the number r_k of reduced bag sequences in which exactly k different vertices occur and in which adjacent bags differ in exactly one vertex. For $k = 0$, there is one sequence, which has length 1 and consists of the empty set, so $r_0 = 1$. For $k > 0$, we construct all sequences from the sequences with $k-1$ vertices. These have length $s_{k-1} = 2(k-1)+3$, and there are $\binom{s_{k-1}}{2} + s_{k-1} = \frac{1}{2}(s_{k-1}^2 + s_{k-1})$ ways to choose the subrange for the k-th vertex, giving

$$r_k = \frac{1}{2}(s_{k-1}^2 + s_{k-1})r_{k-1} = \frac{1}{\sqrt{\pi}}\, 4^k\, k!\, \Gamma\!\left(k + \frac{1}{2}\right)$$

where $\Gamma(\cdot)$ is Euler's gamma function, which generalizes the factorial function to arbitrary real arguments. The number R_k of reduced bag sequences over a set of $k+1$ fixed vertices then is

$$R_k = \sum_{i=0}^{k+1} \binom{k+1}{i} \cdot r_i$$

Given a tree decomposition of width k and a desired pathwidth of ℓ, the number of different characteristics can be up to (T_ℓ is the number of $\mathcal{T}_\ell$-sequences)

$$\begin{aligned}
C_{k,\ell} &\le \sum_{i=0}^{k+1} \binom{k+1}{i} \cdot r_i \cdot T_\ell{}^{s_i} \\
&= \sum_{i=0}^{k+1} \binom{k+1}{i} \left(\frac{1}{\sqrt{\pi}}\, 4^i\, i!\, \Gamma\!\left(i+\frac{1}{2}\right)\right) \left(\frac{32}{3}4^\ell - \frac{2}{3}\right)^{2i+3} \\
&= \sum_{i=0}^{k+1} \binom{k+1}{i} \cdot 2^{\Theta(i \log i)} \cdot 2^{\Theta(i \cdot \ell)} \\
&= 2^{\Theta(k \log k + k \cdot \ell)}.
\end{aligned}$$

Due to the requirement that consecutive bags differ in exactly one vertex, the difference between the last and the first element of the $\mathcal{T}_\ell$-sequences of consecutive reduced bags is 1; to get an asymptotic lower bound, we observe that if we choose every second $\mathcal{T}_\ell$-sequence at will, the gaps can be filled with simple $\mathcal{T}_\ell$-sequences; hence

$$C_{k,\ell} \ge \sum_{i=0}^{k+1} \binom{k+1}{i} \cdot r_i \cdot T_\ell{}^{\lceil s_i/2 \rceil} = 2^{\Theta(k \log k + k \cdot \ell)}.$$

Altogether, we obtain $C_{k,\ell} = 2^{\Theta(k \log k + k \cdot \ell)}$. At every tree node, we have to process at most $C_{k,\ell}$ many characteristics, which can be combined using table lookups in time proportional to the size of characteristics, $\Theta(k+\ell)$, so, as promised, the entire algorithm will run in time $2^{\Theta(k \log k + k \cdot \ell)} \cdot \mathrm{O}(n) = \mathrm{O}(2^{\mathrm{poly}(k,\ell)} \cdot n)$. For a few concrete values of k and ℓ, Table 1 shows how many characteristics can arise at any tree node. The values shown do not represent a loose upper bound—in a totally disconnected graph with an arbitrary tree decomposition of width k, there really are $C_{k,\ell}$ many characteristics at every tree node when we try to compute a path decomposition of width ℓ. However, to dismiss the algorithm based on this evidence as completely impractical is

		$\ell = 1$ $T_1 = 42$	$\ell = 2$ $T_2 = 170$	$\ell = 3$ $T_3 = 682$	$\ell = 4$ $T_4 = 2,730$
$k = 1$	$R_1 =$ 9	$4.48 \cdot 10^5$	$2.95 \cdot 10^7$	$1.90 \cdot 10^9$	$1.22 \cdot 10^{11}$
$k = 2$	$R_2 =$ 112	$2.81 \cdot 10^8$	$7.52 \cdot 10^{10}$	$1.94 \cdot 10^{13}$	$4.99 \cdot 10^{15}$
$k = 3$	$R_3 =$ 2,921	$3.30 \cdot 10^{11}$	$3.58 \cdot 10^{14}$	$3.71 \cdot 10^{17}$	$3.82 \cdot 10^{20}$
$k = 4$	$R_4 =$ 126,966	$6.24 \cdot 10^{14}$	$2.73 \cdot 10^{18}$	$1.14 \cdot 10^{22}$	$4.69 \cdot 10^{25}$

Table 1: Some values of the number of reduced bag sequences R_k, the number of $\mathcal{T}_\ell$-sequences T_ℓ, and the *lower* bound on the number of characteristics $C_{k,\ell}$. The number of vertices in a reduced bag sequence is $k+1$ and the maximum of the utilization sequences is at most $\ell + 1$.

premature for two reasons: The degenerate case just cited is actually very easy to handle: if we pipeline the computation, then a single characteristic at every node will suffice for finding a characteristic at the root. In general, computing characteristics "on demand" improves the running time on sparse graphs. Secondly, we already observed that many characteristics are redundant because they are subsumed by smaller characteristics of "better" solutions. The effect of these optimizations will be investigated in the following section.

3.7 The Implementation

We implemented the Bodlaender-Kloks path-decomposition algorithm by substituting the definition of the characteristic and the combination algorithms into the generic tree automaton "template" described in Section 2.4. We preserved the generality of the algorithm by setting up the desired path-width ℓ as a runtime parameter (as opposed to a compile-time parameter) just as the input graph and tree decomposition. The width k of the tree decomposition does not occur in the description of the algorithm nor in our implementation—as a bound on the maximum bag size, it appears only in the analysis of the algorithm. The parts of the resulting program specific to path decomposition comprise data structures for

- utilization sequences,
- $\mathcal{T}$-sequences,
- reduced bag sequences,
- characteristics of path decompositions, and
- partial solutions

as well as procedures for the combination of characteristics and of partial solutions at the different tree-node types. The functionality of the data structures and the algorithms used closely follow the description in the previous sections. In particular, partial solutions are merged by imitating the correctness proofs.

Integer Sequences

Bag-utilization values ($\mathcal{U}$-sequences) and $\mathcal{T}$-sequences are implemented as arrays of integers that support the following operations ($\alpha, \beta \in \mathcal{U}$, and $\sigma, \rho \in \mathcal{T}$):

len($\boldsymbol{\alpha}$) returns the number of elements in sequence α.

$\boldsymbol{\alpha}[\boldsymbol{i}]$ queries the ith element of α.

max $\boldsymbol{\alpha}$ returns the maximum of α; this takes time O(1) as the maximum is maintained in a variable.

$\boldsymbol{\alpha + c}$ adds a constant $c \in \mathbb{N}$ to all elements of α.

$\boldsymbol{\alpha \oplus_m \beta}$ computes for all same-length expansions α^*, β^*, the pairwise sum $\gamma = \alpha^* + \beta^*$, retaining only sums γ with $\max \gamma \leq m$. Not all expansions are considered, but only those where at each position, either a new element from α occurs in α^* or a new element from β occurs in β^*. The other sums of expansions, especially those of length greater than $\mathrm{len}(\alpha) + \mathrm{len}(\beta)$, are necessarily expansions of smaller sums, hence superfluous for our purpose. The elements of $\alpha \oplus_m \beta$ are computed on demand to allow pipelining with higher-level functions; elimination of duplicates does not occur to avoid storing all previous sums.

$\boldsymbol{\tau(\alpha)}$ projects α to $\mathcal{T}$ using a straightforward quadratic-time algorithm.

$\boldsymbol{\tau(\alpha \oplus_m \beta)}$ projects the sum of $\mathcal{U}$-sequences to $\mathcal{T}$, thereby discarding duplicates. This is a simple composition of the $\alpha \oplus_m \beta$ operation, $\tau(\cdot)$ and a set data structure. Merging $\mathcal{T}$-sequences at Join nodes, i.e., computing

$$\tau((\sigma_j \oplus \rho_j) - |\bar{B}_j|),$$

can be implemented with the procedures presented so far if we omit the elimination of redundant characteristics.

$\boldsymbol{\sigma \preccurlyeq \rho}$ compares two $\mathcal{T}$-sequences and determines whether σ and ρ are equal or incomparable or which of σ and ρ is strictly smaller. The linear-time algorithm employed originates from an idea by Hagerup [Hag98b].

$\tau^*(\alpha \oplus_m \beta)$ supersedes the $\tau(\alpha \oplus_m \beta)$ operation by purging non-minimal $\mathcal{T}$-sequences from the output. This is achieved by computing the elements $\gamma \in \alpha \oplus_m \beta$ one by one and comparing $\tau(\gamma)$ against the list of previously computed compressed sums. $\tau(\gamma)$ replaces an earlier greater sum, or is discarded if an earlier smaller sum is found, or is appended to the list if it is found to be incomparable to all list elements. Hence this operation computes the minimum number of elements of

$$\tau((\sigma_j \oplus \rho_j) - |\bar{B}_j|)$$

necessary to ascertain the completeness of the Join-node combination algorithm.

Computing Characteristics

The characteristic of a path decomposition of some G_x was defined as a list of reduced bags with associated compressed utilization sequences. In our implementation, we chose not to store a list of vertex sets for the bags; instead we opted for a more compact representation by giving for each vertex the number of the first and the last bag in which it occurs. Accordingly, a characteristic consists of two lists, one of length at most $|B_x| \leq k+1$ containing the vertex intervals and a list with the $\mathcal{T}$-sequences, whose length equals the number of bags in the reduced bag sequence. Although very convenient, our representation is somewhat less efficient for computing solutions from characteristics than the "implicit" representation given by Bodlaender and Kloks.

Introducing new vertices thus means to add one vertex interval, adjust the others to reflect the split, and to increment the $\mathcal{T}$-sequences within the new range. Forgetting a vertex amounts to deleting the corresponding interval, accommodating the elimination of two bags in the other intervals, and concatenating two pairs of $\mathcal{T}$-sequences. Merging two characteristics is performed by first comparing the lists of intervals—the reduced bag sequences must coincide—and then outputting all combinations of choosing one $\mathcal{T}$-sequence from

$$\tau^*((\sigma_j \oplus_m \rho_j) - |\bar{B}_j|)$$

for every j and $m = k + 1 + |\bar{B}_j|$. For fixed input characteristics, all four combination algorithms produce only incomparable characteristics: This is evident for Start and Forget nodes, which yield only a single characteristic; for Introduce nodes, we ensure this by never splitting at a sequence maximum—it can be shown that only splits at maxima lead to redundant characteristics. Characteristics computed at Join nodes are mutually incomparable by virtue

of the same property of the sets $\tau^*((\sigma_j \oplus_m \rho_j) - |\bar{B}_j|)$. However, in most cases, the combination algorithm at a tree node is called for multiple combinations of children characteristics; therefore it may happen that two different input characteristics lead to the same output or to comparable characteristics. In this event, redundant characteristics get optionally removed by the generic framework.

Figures 18, 19 and 20 show some details of an exemplary path-decomposition computation. To maintain coherence with the presentation of the theory, the characteristics in Figure 20 have been converted to lists of pairs of a bag and a $\mathcal{T}$-sequence; so at the root,

$$\langle(\emptyset, 0\ 1), (\{2\}, 2\ 1\ 4), (\emptyset, 3), (\{4\}, 4\ 2\ 3), (\emptyset, 2\ 0)\rangle$$

represents a reduced bag sequence of an empty bag, followed by a bag with the vertex labeled "2", followed by an empty bag, a bag with vertex "4", and another empty bag. The first bag has the $\mathcal{T}$-sequence $\langle 0, 1\rangle$, the second $\langle 2, 1, 4\rangle$, and so on.

Benchmarks

The benchmarks were run on a Sun Enterprise 10000 computer [Cha98], where up to eight tests could be executed simultaneously on as many 333 MHz Ultra-2 processors, which shared two gigabytes of main memory. The programs were written in C++ and compiled using the GNU C++ compiler. Details about the software and the development environment are given in the appendix.

Our test cases with a known upper bound on the pathwidth are created as in Figure 21 by using paths and cacti as "skeletons" for ℓ-tree constructions, similar to the triangle construction in Chapter 1: We maintain a mapping between tree nodes and $(\ell + 1)$-cliques in the growing graph; starting with an $(\ell + 1)$-clique identified with an arbitrary node of the tree, children of tree nodes get their counterparts in the graph by inserting one new vertex and making it adjacent to all vertices of the parent's clique except for one vertex, which is chosen uniformly at random. Graphs with path skeletons have pathwidth ℓ: turning the $(\ell + 1)$-cliques into bags linked in the order of construction, we get a path decomposition of width ℓ. Furthermore, these graphs are maximal in the sense that adding any new edge will increase the treewidth and hence the pathwidth of the graph (Proposition 8). Graphs generated from cacti have pathwidth at most $\ell + 1$, because in the tree decomposition of width ℓ, bags of inner nodes can be replaced by two consecutive bags of size $\ell + 2$, yielding a path decomposition of width $\ell + 1$.

From these graphs, sparser and less regular graphs are obtained by deleting edges at random. The necessary tree decompositions were computed

using the algorithm from [ACP87], which offered acceptable performance for the graphs that the path-decomposition algorithm could handle. Table 2 shows the effect of the different optimizations described earlier; Figures 22 to 29 show the performance of the fastest configuration of the final implementation. The following observations were made:

- Memory consumption rather than running time proved in many cases to be the limiting factor. This is especially poignant for sparse graphs.
- Therefore it is of utmost importance to reduce the number of characteristics produced at each tree node (Table 2).
- Figure 23 shows linearly growing worst-case running time for a large number of samples; the experiments appear to indicate a constant of ca. 30 s/node for $\ell = 2$. As the test cases from Table 2 show, the performance is often much better. To improve performance on sparse graphs, a good heuristic would be to handle each connected component separately.
- As expected, the time for computing solutions grows faster than linearly (Figure 24); the results are inconclusive as to whether the bound is quadratical as predicted by theory.
- There is no particular bottleneck in the program (Figure 22); the memory management would greatly benefit from a restriction of the values of ℓ—e.g., it would then be possible to store entire $\mathcal{T}_\ell$-sequences in machine words instead of relying on dynamically allocated arrays.
- For $\ell = 2$, the performance of the algorithm is acceptable. Beyond that, practicality is questionable (Figure 29); it is likely that the Bodlaender-Kloks algorithm cannot compete with the algorithms for the special cases $\ell = 2$, $\ell = 3$, and $\ell = 4$, such as the one by Sanders [San96].

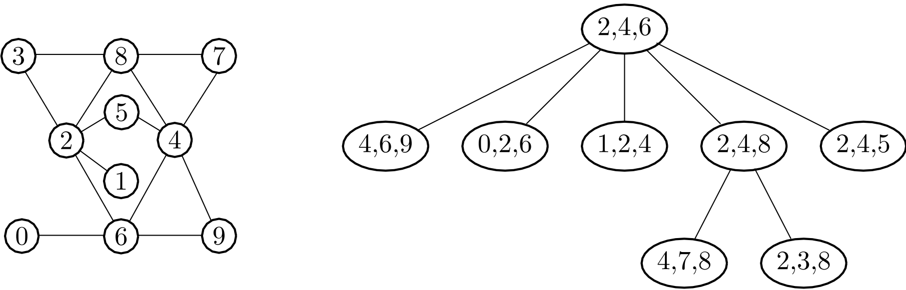


Figure 18: Graph `cactus2t-03.gml` and the width-2 tree decomposition that will be used in the following examples.

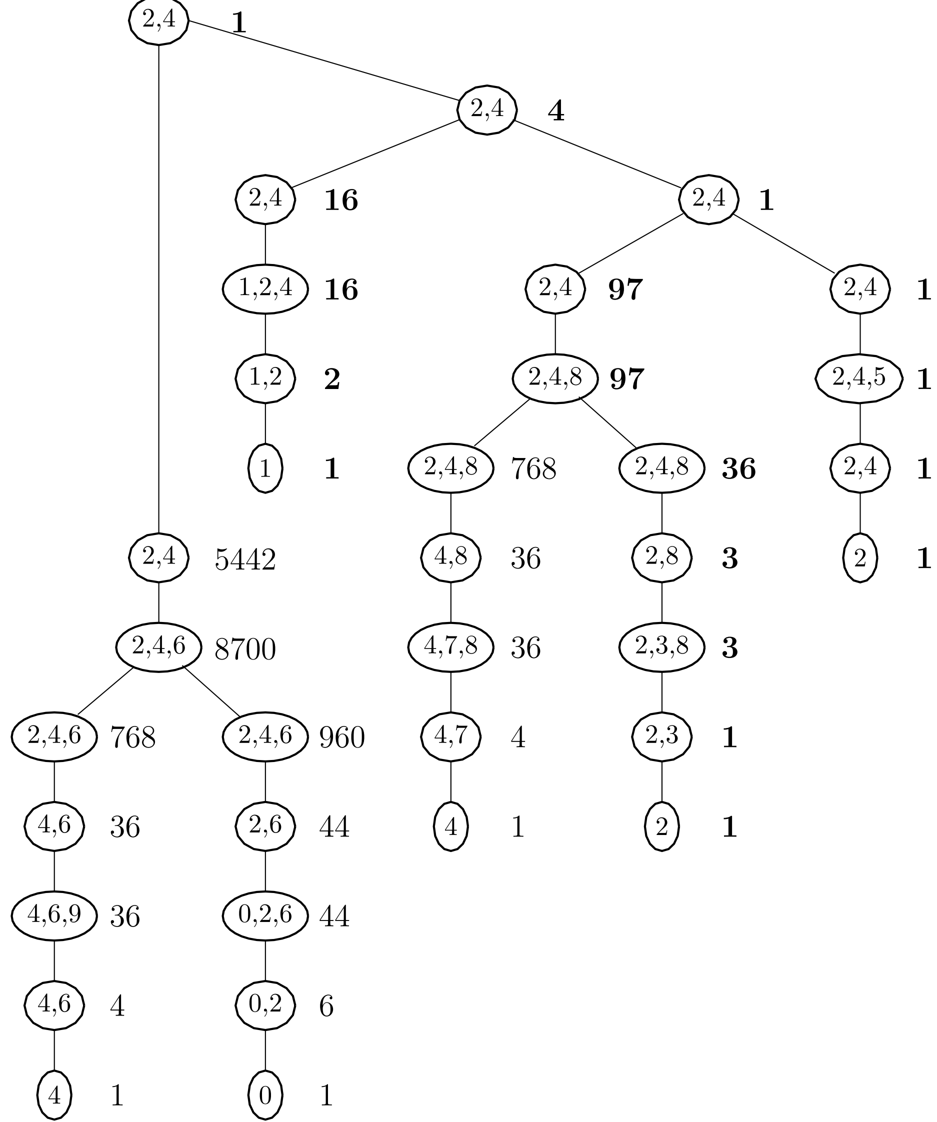


Figure 19: A tree decomposition of graph `cactus2t-03.gml` annotated with the number of characteristics computed at every node. This tree decomposition is the result of converting the tree decomposition from Figure 18 to the Start-Introduce-Forget-Join-node format; the bold numbers indicate bags where the algorithm did not compute a full set of characteristics or did not notice that it did.

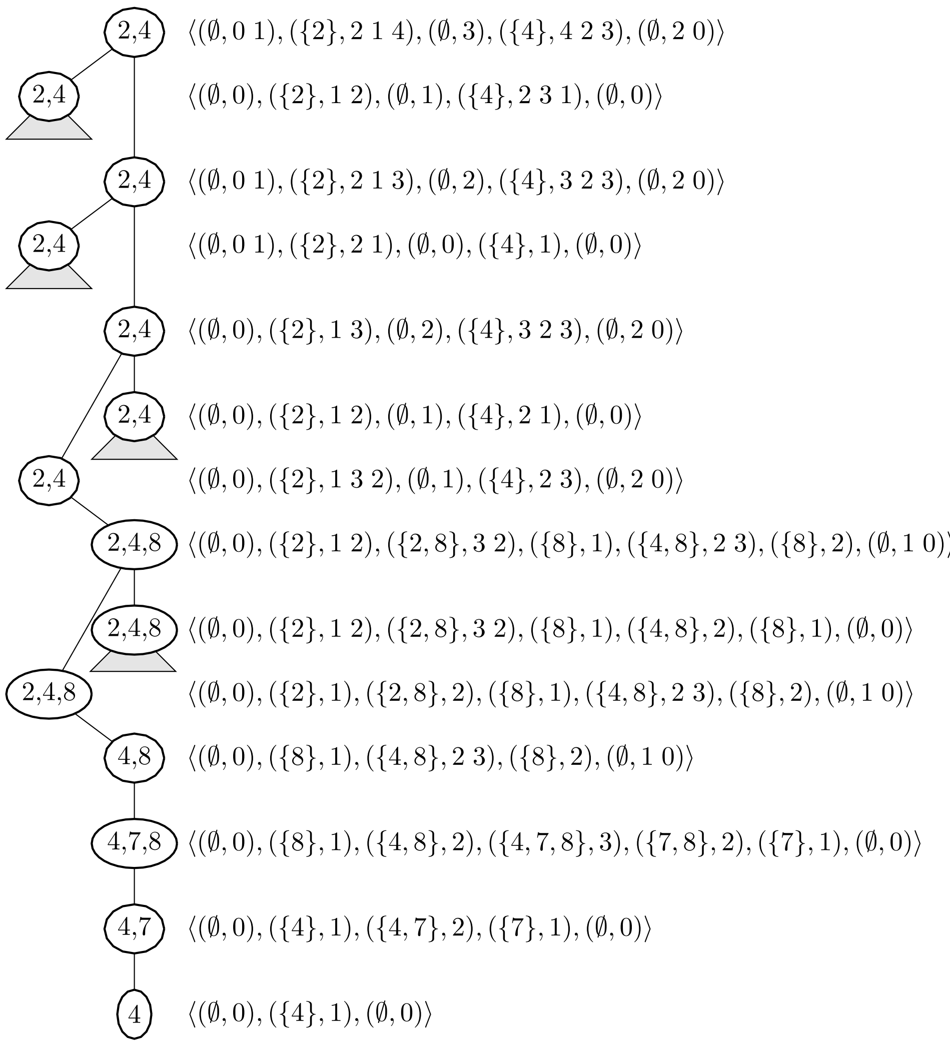


Figure 20: One path of the tree decomposition of `cactus2t-03.gml` annotated by the characteristics that lead to a solution (see the remarks on page 60)

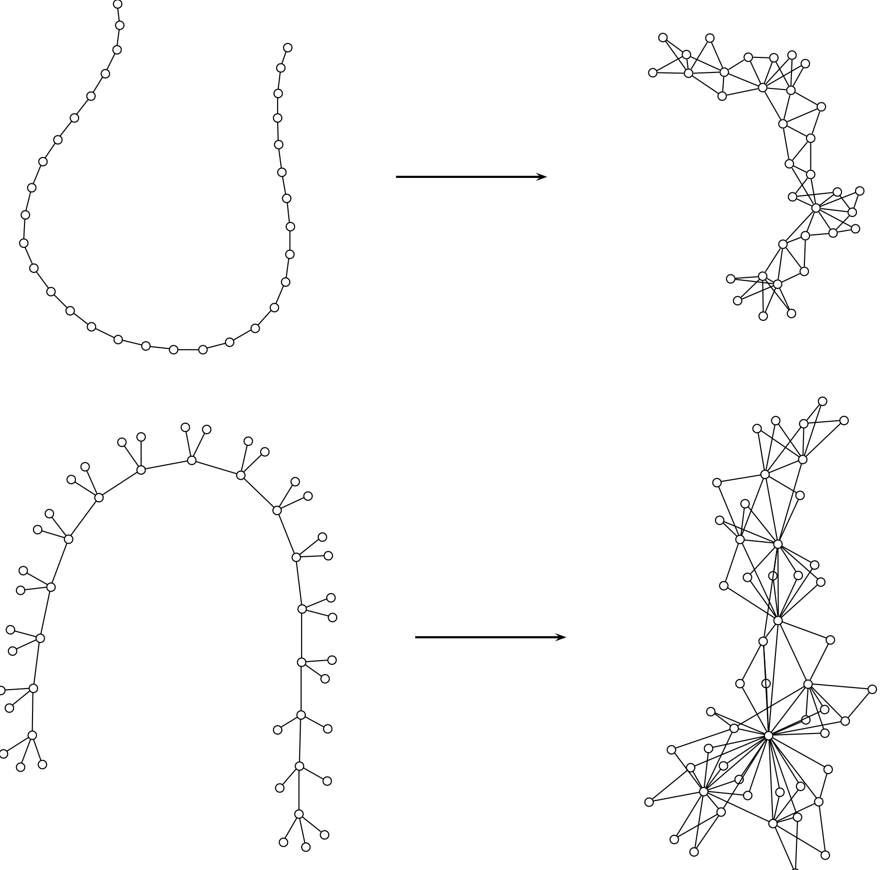


Figure 21: Generating test cases by “blowing up” paths and cacti. Simple trees guide a 2-tree construction, yielding graphs with predictable bounds on the pathwidth.

	test 1	test 2	test 3	test 4	test 5	test 6	test 7	test 8
final, no compiler optimization	4.1 s 6 MB	3.9 s 5 MB	3.3 s 6 MB	5.7 s 6 MB	327.5 s 34 MB	106.2 s 13 MB	3.0 s 6 MB	5251.9 s 242 MB
final	1.9 s 6 MB	1.5 s 5 MB	1.2 s 5 MB	2.0 s 6 MB	141.9 s 34 MB	56.7 s 13 MB	1.6 s 6 MB	2454.2 s 242 MB
no sum optimization	2.0 s 6 MB	1.5 s 5 MB	1.2 s 5 MB	2.0 s 6 MB	138.6 s 36 MB	51.6 s 14 MB	1.7 s 6 MB	4146.0 s 328 MB
no split optimization	2.8 s 6 MB	1.6 s 5 MB	1.5 s 5 MB	2.4 s 6 MB	161.4 s 36 MB	65.5 s 14 MB	1.7 s 6 MB	4185.9 s 340 MB
no redundancy elimination	8.4 s 15 MB	5.0 s 8 MB	8.0 s 12 MB	62.3 s 33 MB	776.0 s Ⓕ 1027 MB	1278.5 s 80 MB	12.1 s 22 MB	4483.7 s Ⓕ 1028 MB
no caching	Ⓘ 75 h 179 MB	Ⓝ	Ⓝ	Ⓝ	Ⓝ	Ⓝ	Ⓝ	Ⓝ
no redundancy elim., no sum opt.	9.4 s 17 MB	5.0 s 8 MB	8.1 s 13 MB	64.1 s 39 MB	765.8 s Ⓕ 1027 MB	3373.0 s 165 MB	19.2 s 33 MB	5720.4 s Ⓕ 1028 MB

Ⓕ — tests that could not be completed due to memory exhaustion

Ⓝ — tests that were not conducted

Ⓘ — tests that were interrupted

Table 2: The effect of the various optimizations.

test case	graph	tree decomposition		requested pathwidth
		tree	width	
1			2	2
2			3	3
3				3
4				3
5			2	2
6			2	2
7			2	2
8				3

Table 3: The test cases used for evaluating the optimizations.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %     self
 time  seconds    calls  name
 49.32  269.35           __mcount_internal
 11.10   60.62           mcount
  2.68   14.66  9743318  chunk_free
  2.20   11.99  9754826  chunk_alloc
  0.96    5.24  9743318  cfree
  0.95    5.17  5439583  gen_array::clear
  0.85    4.63  9754826  malloc
  0.69    3.79 16714204  leda__access<pdc::chrctr::vinfo>
  0.69    3.75  4594919  memory_manager::allocate_vector
  0.66    3.60 27216762  dlist::entry
  0.56    3.07  1884714  gen_array::gen_array
  0.53    2.88  4594919  memory_manager::deallocate_vector
  0.52    2.85 13229150  leda_array<char>::clear_entry
  0.52    2.83 24739802  leda_access<char>
  0.47    2.59  1798807  gen_array::init
  0.47    2.58  3367102  lex_compare<leda_list<pdc::chrctr::vinfo> >
  0.47    2.55 11168570  leda_array<char>::operator[]
  0.47    2.54  7054689  compare
  0.43    2.37           ___builtin_new
  0.43    2.36  8928161  iseq::operator[]
  0.43    2.34 10159550  dlist::first_item
  0.42    2.31  8152263  ref<pdc::chrctr>::operator*
  0.42    2.30  5883299  dlist::clear
  0.41    2.23 14119442  leda_list<pdc::chrctr::vinfo>::contents
  0.39    2.15  8422410  leda_create<char>
  0.39    2.13 14413062  leda__access<char>
  0.39    2.13  4474456  gen_array::~gen_array
  0.37    2.01 14119442  leda_list<pdc::chrctr::vinfo>::inf
  0.36    1.99  5687167  dlist::length
  0.35    1.90  4704435  ref<pdc::chrctr>::discard
  0.34    1.86  2124187  dlist::append
  0.34    1.85  8928161  leda_array<char>::operator[]
  0.33    1.81    11519  tree_automaton<pdc>::ta_join::iter::next1
```

Figure 22: Excerpt from a profile of test case 6 by the gprof utility. The first two entries indicate that profiling incurred a 60% performance penalty; they are followed by the memory management routines, and low-level LEDA functions; from the algorithm proper, only some comparison function, and the generic Join operation show up in this list. Even though similar results were found with other test cases, these figures are to be treated with caution; e.g., suppressing the inlining functions may have significantly distorted the distribution of CPU cycles.

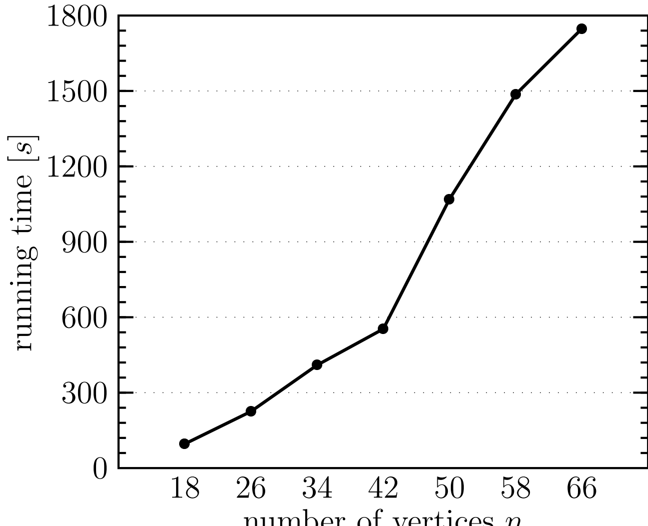


Figure 23: Running time against growing graph size. For every n, 32 maximal pathwidth-2 graphs were generated by using a path as skeleton. After randomly removing multiples of four edges, the time for computing *all* characteristics was taken, always using the tree decomposition that derives naturally from the construction of the maximal graph. The diagram above shows for each n the greatest running time encountered.

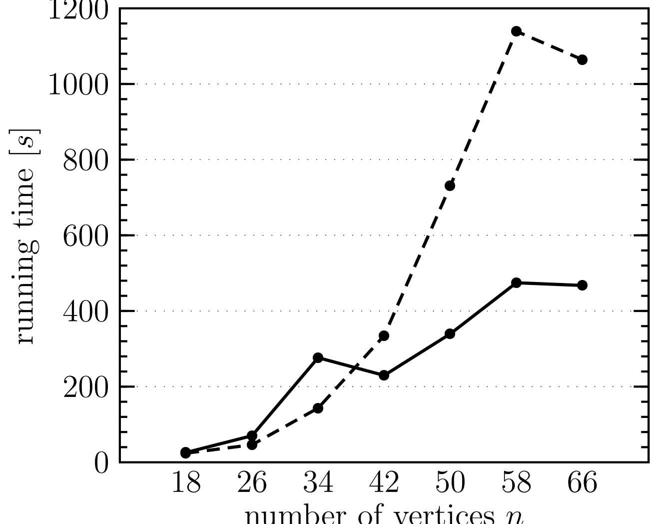


Figure 24: For the same experiment as in Figure 23, this figure shows the maximum time for computing the first characteristic at the root (solid line) and for computing a solution (dashed line) for each n.

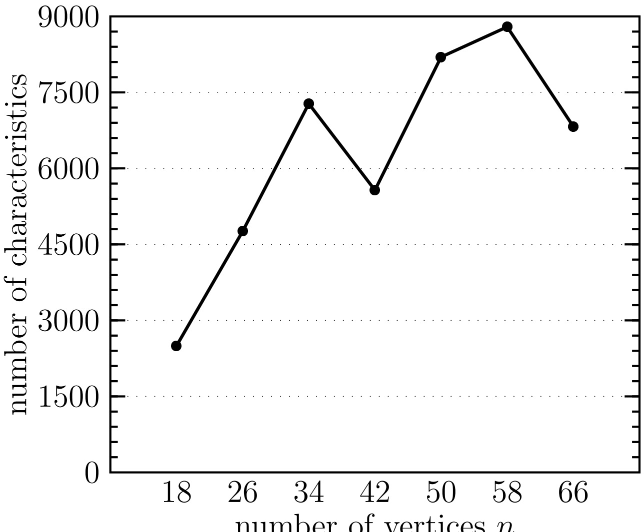


Figure 25: Maximal number of characteristics at any tree node, plotted against growing n (same experiment as in Figures 23 and 24).

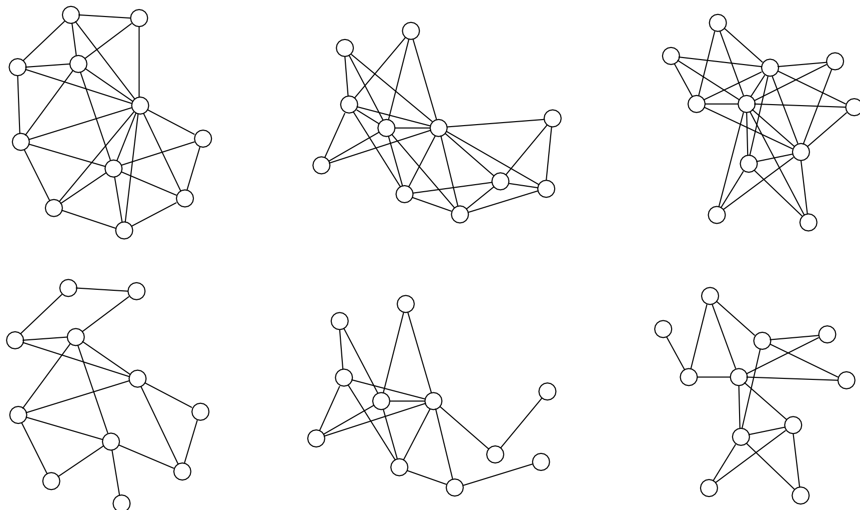


Figure 26: The three maximal pathwidth-3 graphs used for investigating the influence of graph density on the running time, and the same graphs with ten edges removed at random. The results of this experiment are shown in Figures 27 and 28.

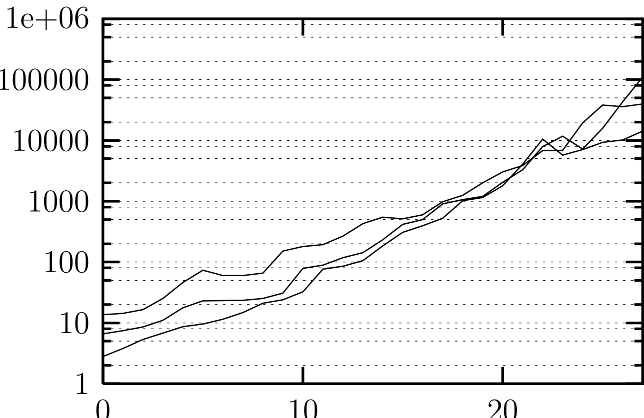


Figure 27: Running time against density of graph: From each of the three graphs of Figure 26, one edge after the other was removed in random order and after each deletion, a path decomposition was computed, using the tree decomposition of the original graph. In the diagram, the number of deleted edges is plotted against the time (in seconds) to compute all characteristics at the root node. The number of characteristics grows roughly exponentially with the number of deleted edges.

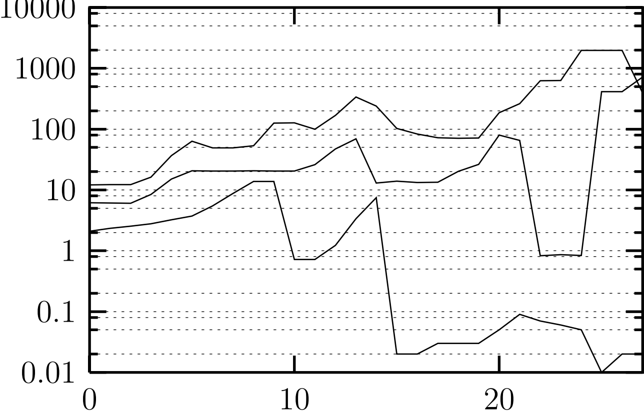


Figure 28: Running time against density of graph: For the same experiment as in Figure 27, the time for computing the first characteristic at the root and the corresponding solution is shown. This benchmark remains inconclusive.

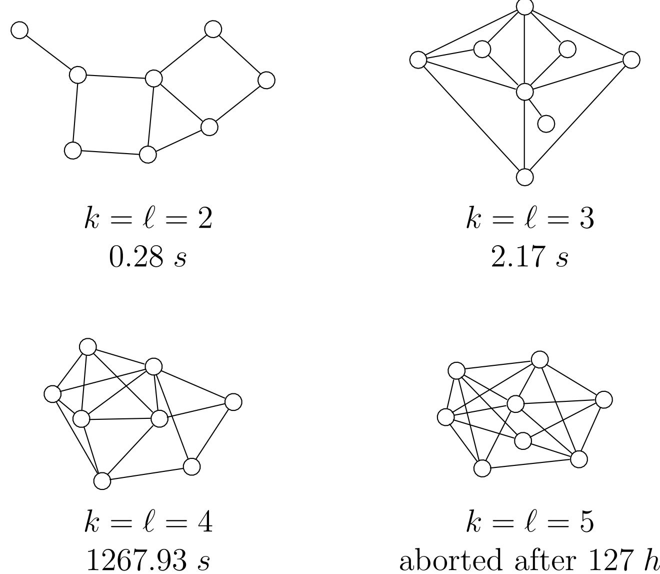


Figure 29: Running time for fixed n and growing k and ℓ.

Chapter 4

Tree-Decomposition Algorithms

In this chapter, we review algorithms for computing tree decompositions. As input, these algorithms take a graph G and an integer k. If G has treewidth at most k, they compute a tree decomposition of G of width at most k; otherwise they correctly state that G has treewidth greater than k. Building on such a procedure, it is easy to find a tree decomposition of optimal width, for example, by calling the procedure with $k = 1, 2, \ldots$ until a tree decomposition is found. Using a tree-decomposition algorithm with time bound $\mathrm{O}(g(k) \cdot n^c)$—i.e., one that exhibits the property of fixed-parameter tractability—and imposing an upper bound on the treewidth, the running time for finding the treewidth is $\mathrm{O}(n^c)$.

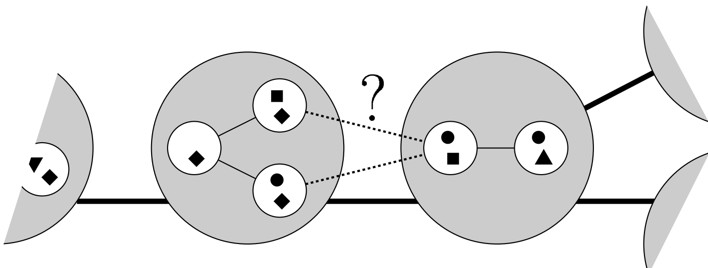
While discussing the computation of tree decompositions, we will assume that the input graph G is connected; for graphs with more than one connected component, tree decompositions of the individual components can be merged by linking the trees at arbitrary tree nodes. Section 4.1 provides an important subroutine for many tree-decomposition algorithms. In Section 4.2, we present tree-decomposition algorithms that rely on computing separators and which culminate in Reed's $\mathrm{O}(n \log n)$ algorithm [Ree92]. Section 4.3 is devoted to a different approach, by which Bodlaender [Bod96a] succeeded in devising a linear-time algorithm for computing minimum-width tree decompositions of graphs of bounded treewidth. In this chapter, we maintain a theoretical perspective and lay the groundwork for Chapters 5 and 6, where we discuss issues of practicality.

4.1 Shrinking Tree Decompositions

We present in this section an algorithm for shrinking tree decompositions: The algorithm takes as input a graph G, a linear-size tree decomposition $(T = (X, F), \{B_x\}_{x \in X})$ of width k, and an integer $\ell < k$. It checks whether G has

treewidth ℓ and if so, computes a tree decomposition of width ℓ. The running time of the algorithm is $\mathrm{O}(2^{\mathrm{poly}(k,\ell)} \cdot n)$, so for fixed k, the bound is linear in the input size. Invented by Bodlaender and Kloks [BK96], the algorithm is an essential constituent of Bodlaender's linear-time algorithm [Bod96a], though it is also needed for post-processing the output of procedures, such as most of the algorithms presented in Section 4.2, which compute from scratch a tree decomposition of constant but non-optimal width.

Shrinking tree decompositions is not straightforward. For example, we cannot turn a tree decomposition of width k into a tree decomposition of width ℓ by decomposing large bags locally and linking the resulting trees—most of the time, it would not be possible to join the tree decompositions of adjacent bags:



Nevertheless, the problem fits into the framework from Chapter 2 for solving problems on graphs of bounded treewidth. By plugging characteristics and combination procedures into the generic algorithm, we solve the problem not so much by taking a wide tree decomposition as a starting point for a shrinking process, as by using the wide tree decomposition as a guide in computing a narrow tree decomposition from scratch. The algorithm by Bodlaender and Kloks can be thought of as an extension of the path-decomposition algorithm presented in Chapter 3: computing tree decompositions via a tree automaton is a generalization of computing path decompositions this way, and we will be able to transfer many of the earlier results. From the projected time bound $\mathrm{O}(2^{\mathrm{poly}(k,\ell)} \cdot n)$, we see that the running time at each node of the tree automaton should again be independent of n; therefore the maximum number of characteristics at any tree node must not depend on G.

The Characteristic of a Tree Decomposition

Recall that at a tree node x, the characteristic of a path decomposition $\langle \hat{B}_i \rangle_{1 \le i \le m}$ of the subgraph G_x was made up of two constituents: the reduced bag sequence $\langle \bar{B}_j \rangle_{1 \le j \le m'}$—the projection of $\langle \hat{B}_i \rangle_{1 \le i \le m}$ to the bag B_x at tree node x, with repeated bags removed—and compressed utilization sequences

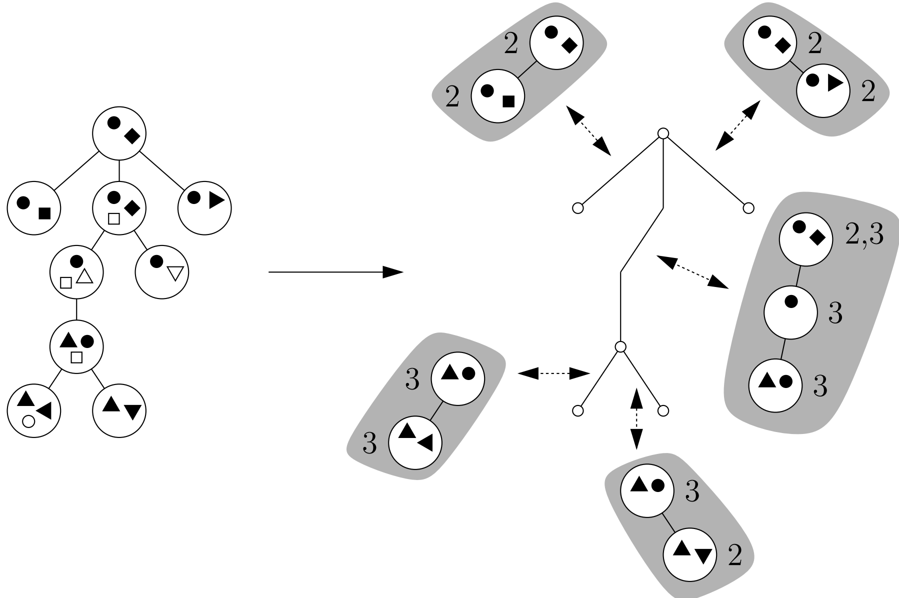


Figure 30: The characteristic of an exemplary tree decomposition. Black vertices are in B_x, white vertices in $G_x \setminus B_x$; the characteristic of the tree decomposition on the left consists of the trunk on the right whose edges are annotated with path-decomposition characteristics.

associated with each reduced bag $\bar{B}_j$, which convey the essential information about what was lost in the projection. Likewise, the characteristic of a tree decomposition $S_x = (\hat{T} = (\hat{X}, \hat{F}), \{\hat{B}_{\hat{x}}\}_{\hat{x} \in \hat{X}})$ of graph G_x is constructed by restricting S_x to B_x, removing "repeated" bags, and annotating the restriction with utilization values. Note that we follow again the convention of marking components of partial solutions by a hat ($\hat{}$) and parts of characteristics by a bar ($\bar{}$), while leaving objects of the backbone tree decomposition unmarked.

Let $S_x = (\hat{T} = (\hat{X}, \hat{F}), \{\hat{B}_{\hat{x}}\}_{\hat{x} \in \hat{X}})$ be a partial solution at tree node x; we now develop the definition of the tree-decomposition characteristic C_x of S_x. To simplify the reuse of the path-decomposition procedures, we proceed as follows to create a (tree-)*trunk* $\bar{T}$ to which we can affix characteristics of path decompositions (see Figure 30). We reduce $\hat{T}$ to $\bar{T} = (\bar{X}, \bar{F})$ with $\bar{X} \subseteq \hat{X}$,

(1) by repeatedly removing leaves $\hat{x}$ for which $\hat{B}_{\hat{x}} \cap B_x$ is a subset of $\hat{B}_{\hat{y}} \cap B_x$ at the single neighbor $\hat{y}$, and

(2) by removing all nodes $\hat{x}$ of degree two and making their neighbors adjacent.

The first reduction rule is necessary to bound the size of $\bar{T}$, whereas the second rule removes chains of nodes, which we plan to treat differently: Every

edge $\bar{e} = (\bar{x}, \bar{y})$ in $\bar{T}$ can be associated with the path from $\bar{x}$ to $\bar{y}$ in $\hat{T}$, and we define $\hat{P}_{(\bar{x},\bar{y})} = \langle \hat{B}_{\bar{x}}, \ldots, \hat{B}_{\bar{y}} \rangle$ to be the sequence of the corresponding bags in the original tree decomposition. For any trunk edge $\bar{e} = (\bar{x}, \bar{y}) \in \bar{F}$, $\hat{P}_{\bar{e}}$ is a path decomposition of a subgraph of G_x, and we let $\bar{P}_{\bar{e}}$ be the path-decomposition characteristic of $\hat{P}_{\bar{e}}$, so $\bar{P}_{\bar{e}} = \langle (\bar{B}_{\bar{e},j}, \tau_{\bar{e},j}) \rangle_{1 \leq j \leq m'}$ consists of the sequence $\langle \hat{B}_{\bar{x}} \cap B_x, \ldots, \hat{B}_{\bar{y}} \cap B_x \rangle$ with repetitions removed and the corresponding compressed utilization sequences. Labeling the trunk edges with these path decompositions completes the construction of the tree decomposition characteristic $C_x = (\bar{T} = (\bar{X}, \bar{F}), \{\bar{P}_{\bar{e}}\}_{\bar{e} \in \bar{F}})$. Note that again we have the invariant that inserting the reduced bags into the trunk—i.e., building the tree of bags that results from substituting the reduced bag sequences for the trunk edges—gives a tree decomposition of bag B_x; furthermore, the path decompositions at edges that share an endpoint, share the last bag and the last element of the compressed utilization sequence of the last bag. However, there is a characteristic in which the trunk $\bar{T}$ does not have any edge at all—such a characteristic C_x at node x with a *degenerate trunk* represents all tree decompositions of G_x where all vertices of B_x occur in some bag: in this and only this case, reduction rule (2) leaves only a single $\bar{x}$ with bag $\hat{B}_{\bar{x}} \supseteq B_x$.

How many tree-decomposition characteristics are there? By reduction rule (1), every bag $\hat{B}_{\bar{x}}$ of a trunk leaf $\bar{x}$ contains a vertex from B_x that is in no bag $\hat{B}_{\bar{y}}$ of any other trunk node $\bar{y} \in \bar{X}$, $\bar{y} \neq \bar{x}$. Since B_x is a bag of the input tree decomposition and thus contains at most $k+1$ vertices, the trunk $\bar{T}$ can have at most $k+1$ leaf nodes and hence at most $2k$ nodes in total. Each of the at most $2k-1$ edges can be labeled with one of $2^{\Theta(k \log k + k \cdot \ell)}$ characteristics of path decompositions (Section 3.6), implying a bound of

$$\mathrm{O}\Big(\big(2^{\mathrm{O}(k \log k + k \cdot \ell)}\big)^{2k-1}\Big) = 2^{\mathrm{O}(k^2 \log k + k^2 \cdot \ell)}$$

on the number of characteristics and a bound of $2^{\mathrm{O}(k^2 \log k + k^2 \cdot \ell)} \cdot n$ on the running time $S(n, k, \ell)$. With a little more effort, it can be seen that the number of characteristics is $2^{\Theta(k^2 \log k + k^2 \cdot \ell)}$.

We can easily extend the partial order $\preccurlyeq$ from path-decomposition characteristics to entire characteristics of tree decompositions: for tree-decomposition characteristics C_x and C'_x shall hold $C_x \preccurlyeq C'_x$ if they have the same trunk, so $C_x = (\bar{T} = (\bar{X}, \bar{F}), \{\bar{P}_{\bar{e}}\}_{\bar{e} \in \bar{F}})$ and $C'_x = (\bar{T} = (\bar{X}, \bar{F}), \{\bar{P}'_{\bar{e}}\}_{\bar{e} \in \bar{F}})$ and if $\bar{P}_{\bar{e}} \preccurlyeq \bar{P}'_{\bar{e}}$ for all trunk edges $\bar{e} \in \bar{F}$. Since we are using $\mathcal{T}$-sequences instead of uncompressed utilization sequences, we aim again for the weaker kind of completeness, where for each partial solution S_x with characteristic C_x, we only require that a characteristic C'_x of a better solution S'_x with $C'_x \preccurlyeq C_x$ is computed. This also permits us to transfer the elimination of redundant

characteristics from the path-decomposition case to the tree-decomposition case.

Start Nodes

The combination algorithms are extensions of the corresponding procedures for PATHWIDTH, except for the Start-node combination algorithm. At a Start node x with $B_x = \{v\}$, partial solutions can have only the degenerate trunk $\bar{T} = (\{\bar{x}\}, \emptyset)$, so that $C_x = (\bar{T}, \emptyset)$ is the only characteristic generated at x.

Introduce Nodes

At Introduce nodes x with child y and introduced vertex v, the input is a characteristic $C_y = (\bar{T}, \{\bar{P}_{\bar{e}}\})$ of a partial solution S_y at node y (i.e., S_y is a tree decomposition of G_y) and we are asked to compute all characteristics C_x that stand for a partial solution S_x at x, where S_x is an extension of S_y by the introduced vertex v. Essentially, the combination procedure will construct all characteristics C_x that meet the utilization bound of $\ell + 1$ and which reduce to C_y when v is removed. However, it takes a little struggle to make this explicit.

When we manipulate a path-decomposition characteristic $\bar{P}_{(\bar{x},\bar{y})}$ that is part of a tree-decomposition characteristic C_x of a partial solution S_x, i.e.,

$$S_x = (\hat{T} = (\hat{X}, \hat{F}), \{\hat{B}_{\hat{x}}\}_{\hat{x} \in \hat{X}}),$$
$$C_x = (\bar{T} = (\bar{X}, \bar{F}), \{\bar{P}_{\bar{e}}\}_{\bar{e} \in \bar{F}}),$$
$$\bar{P}_{(\bar{x},\bar{y})} = \langle (\bar{B}_{(\bar{x},\bar{y}),j}, \tau_{(\bar{x},\bar{y}),j}) \rangle_{1 \leq j \leq m'},$$

we must bear in mind that some of the reduced bags $\bar{B}_{(\bar{x},\bar{y}),j}$ may originate from a subsequence of $\hat{P}_{(\bar{x},\bar{y})} = \langle \hat{B}_{\bar{x}}, \ldots, \hat{B}_{\bar{y}} \rangle$ containing bags $\hat{B}_{\hat{z}}$ where $\hat{z}$ has in $\hat{T}$ a branch that gets eliminated by the reduction rules (1) and (2). In Figure 30, the first inner bag from the top at the long edge is an example of such a $\hat{z}$. Which operations on the partial solution S_x correspond to inserting v into $\bar{P}_{(\bar{x},\bar{y})}$ using the Introduce-node combination algorithm for path-decomposition characteristics? Let $\bar{P}'_{(\bar{x},\bar{y})}$ denote the result of such an insertion; the correctness proof of the algorithm gave instructions on how to repeat bags in $\hat{P}_{(\bar{x},\bar{y})}$ and where to add v so that the resulting $\hat{P}'_{(\bar{x},\bar{y})}$ has characteristic $\bar{P}'_{(\bar{x},\bar{y})}$. So far so good, but what happens to the branches that were cut off in deriving the characteristic C_x from S_x? We just attach them to exactly one of the (possible) repetitions of $\hat{z}$, thus preserving the validity of the tree decomposition with respect to vertices other than v. Similarly, merging

two path-decomposition characteristics $\bar{P}^{(1)}_{(\bar{x},\bar{y})}$ and $\bar{P}^{(2)}_{(\bar{x},\bar{y})}$ with the Join-node combination algorithm induces an equivalent operation on the corresponding $\hat{P}^{(1)}_{(\bar{x},\bar{y})}$ and $\hat{P}^{(2)}_{(\bar{x},\bar{y})}$, and we can reattach branches to the result without any problem.

We begin the discussion of the Introduce-node combination algorithm for tree-decomposition characteristics with the computation of characteristics C_x that have the same trunk as C_y. If C_y has a degenerate trunk without edges, $C_x := C_y$ is a valid characteristic at x—meaning "all vertices of B_x in a bag in some partial solution at x"—if $|B_x| \leq \ell+1$. If the trunk of C_y has at least one edge, we check to which reduced bags in path-decomposition characteristics $\bar{P}_{\bar{e}}$ the new vertex v can be added: all edges between v and other vertices of B_x have to be covered, the bags containing v must be connected, and the utilization limit $\ell + 1$ must be respected. For each such legal way of adding v, the compressed utilization sequences are split and updated accordingly. Thus we get all characteristics C_x where the trunk coincides with the trunk of C_y; correctness is immediate since paths in some S_y behave just like path decompositions and, as argued above, truncated branches do not pose a problem.

To obtain the characteristics C_x whose trunks differ from C_y, observe that the trunk only changes when removing v from C_x leads to a trunk leaf whose bag is a subset of the neighbor's bag—this can only happen if v is in the bag of the leaf, but in no other bag. Conversely, assume that all edges of v are covered by putting it in a single reduced bag $\bar{B}$ from any reduced bag sequence in C_y. We create all possible decreasing chains of bags

$$\langle \bar{B}, \bar{B} \setminus \{u_1\}, \bar{B} \setminus \{u_1, u_2\}, \dots, \bar{B} \setminus \{u_1, \dots, u_{r-1}\}, \bar{B} \setminus \{u_1, \dots, u_r\} \rangle$$

that start with $\bar{B}$ and end with some subset $\bar{B} \setminus \{u_1, \dots, u_r\}$ with all neighbors of v in B_x; inserting v into the last bag of such a chain,

$$\langle \bar{B}, \bar{B} \setminus \{u_1\}, \dots, \bar{B} \setminus \{u_1, \dots, u_{r-1}\}, (\bar{B} \setminus \{u_1, \dots, u_r\}) \cup \{v\} \rangle,$$

makes reduction rule (1) inapplicable, so that we can extend the trunk by a new leaf $\bar{y}$, and associate the edge $\bar{e}$ from $\bar{y}$ to its neighbor with the path-decomposition characteristic $\bar{P}_{\bar{e}}$ formed by the bags of the chain and arbitrary $\mathcal{T}$-sequences τ_j with $\min \tau_j \geq |\bar{B} \setminus \{u_1, \dots, u_j\}|$ for $0 \leq j < r$, $\min \tau_r \geq |\bar{B} \setminus \{u_1, \dots, u_r\}) \cup \{v\}|$, and $\max \tau_j \leq \ell + 1$ for $0 \leq j \leq r$:

$$\bar{P}_{\bar{e}} = \langle (\bar{B}, \tau_0), \dots, (\bar{B} \setminus \{u_1, \dots, u_r\}) \cup \{v\}, \tau_r) \rangle.$$

If $\bar{B}$ corresponds to a node $\bar{x}$ in the trunk (i.e., it is a last or first bag of a reduced bag sequence), we make $\bar{y}$ adjacent to $\bar{x}$, i.e., $\bar{e} = (\bar{x}, \bar{y})$. Otherwise,

$\bar{B}$ corresponds in some $\hat{T}$ to a node $\hat{x}$ that was removed by reduction rule (2); we put a new $\bar{x}$ corresponding to $\hat{x}$ into the trunk; then we split the $\mathcal{T}$-sequence of $\bar{B}$ at all possible places to create two path-decomposition characteristics, which go with the two edges that replace the old trunk edge "through $\hat{x}$". We attach $\bar{P}_{\bar{e}}$ to $e := (\bar{x}, \bar{y})$ as before and have syntactically a characteristic C_x. If C_y has a degenerate trunk, we create characteristics C_x by setting $\bar{B} = B_y$ and applying the same construction.

Given such a C_x and a partial solution S_y at y with the input characteristic C_y, we can easily identify the bag in S_y that corresponds to $\bar{B}$ and add the same chain to it. Since in G_x, v can have only edges to other vertices from B_x (and by Lemma 14 not from $G_x \setminus B_x$), all edges are covered and v only occurs in a connected subgraph of the tree decomposition. Such a S_x evidently has characteristic C_x, hence the algorithm behaves correctly in the case of a trunk extension.

Let us address completeness: given a partial solution S_x, which has characteristic C_x, we have to show that a characteristic C'_x with $C'_x \preccurlyeq C_x$ is computed by the presented procedure. By the induction hypothesis, we get a characteristic C'_y from y that is smaller than the characteristic C_y of the restriction S_y of S_x. Inasmuch as $C'_y \preccurlyeq C_y$, they are comparable and hence have the same trunk and reduced bag sequences. So if we can show that v can be added to C_y yielding C_x, then v can be inserted into C'_y in the same way—splits of $\mathcal{T}$-sequences σ in C_y are imitated by the corresponding $\mathcal{T}$-sequence τ in C'_y by first building the common-length expansions σ^* and τ^* with $\tau^* \leq \sigma^*$, and splitting τ^* at the, say, leftmost position where the split element from σ occurs in σ^*. Obviously, the C'_x constructed this way is smaller than C_x.

We conclude the proof of completeness by showing that on input C_y, the combination algorithm indeed outputs C_x. Going from S_y to S_x, where can v appear? If C_x and C_y have the same trunk, the completeness of the introduce operation for path-decomposition characteristics guarantees us that the $\mathcal{T}$-sequences of C_x or smaller ones really are computed. If the trunks of C_x and C_y differ, then v can only occur in the path-decomposition characteristic of a single new edge in the trunk; the combination algorithm finds all possible nodes in the trunk of C_y to which the new node can be joined, and labels the new edge with all possible path-decomposition characteristics.

Forget Nodes

Let x be a Forget node with child y and forgotten vertex v. Each characteristic C_y at y gives rise to one characteristic C_x at x: v is removed from all path-decomposition characteristics and then the reduction rules (1) and (2)

are applied again to shrink the trunk where necessary, concatenating reduced bag sequences and concatenating and recompressing the $\mathcal{T}$-sequences of the boundary bags. For every such C_x, there is a C_y from which it originates and by the induction hypothesis, there is a partial solution S_y at y with characteristic C_y. Clearly, $S_x := S_y$ has at x characteristic C_x, which proves correctness. Given an S_x with characteristic C_x, we show that a C'_x with $C'_x \preccurlyeq C_x$ is computed: At y, $S_y := S_x$ is a partial solution with characteristic C_y. By induction, a C'_y with $C'_y \preccurlyeq C_y$ is found; since C'_y and C_y are comparable, they have the same trunk and the same reduced bag sequences. So "forgetting" v from both C'_y and C_y results in some C'_x and the characteristic C_x of S_x. They satisfy $C'_x \preccurlyeq C_x$, and since C'_x is the output of the combination algorithm on input C'_y, completeness is proved.

Join Nodes

Merging characteristics C_y and C_z at Join node x with children y and z can be largely reduced to merge operations on path-decomposition characteristics. If C_y and C_z differ in their trunks or reduced bag sequences, no characteristic C_x is produced. Otherwise, the path decompositions at each edge of the trunk are combined individually and every way of choosing one merged path-decomposition characteristic at each edge yields one characteristic C_x at x—the trunk of C_x is that of C_y and C_z. If C_y and C_z have a degenerate trunk, the characteristic with a degenerate trunk is produced at x if $|B_x| \le \ell + 1$.

The correctness and completeness proofs profit from the fact that pairs of path decompositions are combined independently. Given a characteristic C_x computed at x, we construct a solution S_x with this characteristic: By induction, there exist partial solutions S_y and S_z with characteristics C_y and C_z, respectively. The paths corresponding to path decompositions $\bar{P}_{\bar{e}}$ at trunk edges $\bar{e}$ are merged like path decompositions, except that in repeating bags, branches attached in $G_y \setminus B_y$ or $G_z \setminus B_z$ are not repeated. The resulting S_x is a partial solution at x and has characteristic C_x.

As for completeness, we are given, as usual, some partial solution S_x and want to exhibit a characteristic C'_x that is at least as good as the characteristic C_x of S_x and which gets computed from some C'_y and C'_z in the full sets of characteristics at y and z, respectively. Restricting S_x to G_y and G_z gives partial solutions S_y at y and S_z at z; to their characteristics C_y and C_z, characteristics $C'_y \preccurlyeq C_y$ and $C'_z \preccurlyeq C_z$ are computed by the induction hypothesis. The completeness of the path-decomposition join operation implies that at each edge $\bar{e} \in \bar{F}$, a path-decomposition characteristic $\bar{P}'_{\bar{e}}$ will be computed from C'_y and C'_z that is smaller than the corresponding path-decomposition characteristic in C_x. Labeling the trunk edges with these $\bar{P}'_{\bar{e}}$ produces a char-

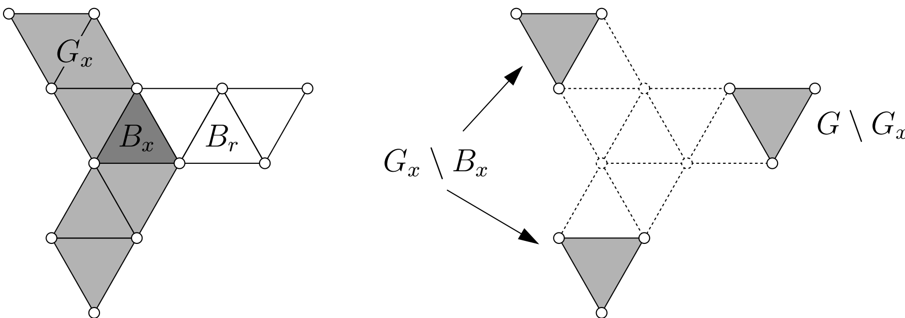


Figure 31: Removing the vertices of bag B_x separates the graph into a component $G \setminus G_x$ towards the root r of the tree decomposition and two components in $G_x \setminus B_x$.

acteristic $C'_x \preccurlyeq C_x$. If the characteristic C_x of S_x has a degenerate trunk, then C_y and C_z will have degenerate trunks as well. By the induction hypothesis, C_y and C_z are computed at y and z, respectively, because there is only a single characteristic with a degenerate trunk. All vertices of B_x must occur in a single bag in S_x, so have $|B_x| \leq \ell + 1$ and the combination procedure produces C_x.

Computing Solutions

Once a characteristic of a tree decomposition has been computed at the root of the backbone tree decomposition, we can follow the constructions of the correctness proofs to construct a tree decomposition of width ℓ. The partial solutions thus computed have size proportional to the number of vertices of the respective subgraphs; therefore the tree decomposition at the root has size $\mathrm{O}(n)$. Moreover, Bodlaender and Kloks show how to compute this tree decomposition in time $\mathrm{O}(n)$ by using a suitable representation for path decompositions. This completes our discussion of the algorithm for shrinking tree decompositions; armed with this important subroutine, we now attack the tree-decomposition problem proper.

4.2 The Separator Approach

An outstanding property of graphs of treewidth k is that they have small *separators*. In a rooted tree decomposition $(T = (X, F), \{B_x\}_{x \in X})$ of $G = (V, E)$ there is a bag B_x so that removing all vertices in B_x disconnects the graph into a component $G \setminus G_x$ and several components in $G_x \setminus B_x$ (Figure 31). Intuitively, choosing tree node x to be "near the center" of T means that

removing B_x decomposes G into parts of balanced size. Indeed, if we define a balanced separator to be a set of vertices S whose deletion from G leaves components of at most $\frac{1}{2}|V \setminus S|$ vertices, then G has a balanced separator of size $k+1$: assume that for all adjacent tree nodes x and y we have $|B_x \setminus B_y| = 1$ and $|B_y \setminus B_x| = 1$ and that all bags have size $k+1$ (a tree decomposition of this form exists because G is a partial k-tree, see Proposition 7). Every inner tree node of such a tree decomposition disconnects G; we start at any inner x and check for each neighbor y of x whether removing B_y from G gives more balanced component sizes than removing B_x from G. As long as an improvement can be made, we repeat the procedure with this neighbor. At termination, no more than $\frac{1}{2}(|V| - (k+1))$ of the vertices can be in any component. It can also be shown that in any graph of treewidth k, there exists a vertex set of size k whose removal leaves components of size at most $\frac{2}{3}(|V| - k)$ (see [Bod96b] for an overview of the relations between different kinds of balanced separators and treewidth).

So graphs of bounded treewidth have small separators, and some of those separators are "central" bags of tree decompositions. A number of tree-decomposition algorithms are based on this observation, among them the ones described in [ACP87, Lag90, MT91, Ree92]. A naïve approach might be as follows: Find a separator of size $k+1$, recursively compute tree decompositions of each component, and glue the resulting tree decompositions together using the separator as common root. The catch is that we have to ensure that in each partial tree decomposition, the vertices of the separator occur all in one bag; otherwise we have the same problem as when shrinking tree decompositions by decomposing large bags locally. We will now look into two ways of dealing with this issue.

The Algorithm by Arnborg, Corneil, and Proskurowski

Arnborg, Corneil, and Proskurowski [ACP87] obtained the first algorithm algorithm for computing tree decompositions of width k with running time polynomial in n by using dynamic programming on components of the input graph $G = (V, E)$. In a first stage, their algorithm determines all size-k vertex sets $S_i \subseteq V$ whose removal disconnects the graph; many such (not necessarily balanced) separators S_i exist in every G of treewidth at most k, since during the construction of a k-tree supergraph of G, every new vertex v is made adjacent to all vertices of a k-clique K and deleting K disconnects v from other vertices built on K. If S_i coincides with such a K, then the components of $G[V \setminus S_i]$ have treewidth k; the idea is to decompose them by creating a table of all components of all separators $\{S_i\}_{1 \leq i \leq s}$ and to use

dynamic programming to determine the decomposability of the components from the small ones up to the largest.

Let $\{C_{i,j}\}_{1\leq j\leq c_i}$ be the connected components of $G[V\setminus S_i]$ and $G_{i,j} := G[C_{i,j}\cup S_i]$. If S_i is the root bag of a tree decomposition, then for all $1\leq j\leq c_i$, there exists a rooted tree decomposition of $G_{i,j}$ with S_i in the bag of root $r_{i,j}$. Conversely, if for some i, all $G_{i,j}$ with $1\leq j\leq c_i$ can be decomposed in this way, a tree decomposition of G can be constructed: create a new root node r and a bag $B_r = S_i$, and for $1\leq j\leq c_i$, link r to the root $r_{i,j}$ of the tree decomposition of $G_{i,j}$. To find a tree decomposition of $G_{i,j}$ comprising a bag with S_i, we create graphs $G'_{i,j}$ from $G_{i,j}$ by making the subgraph induced by S_i complete; by Lemma 11, every tree decomposition of $G'_{i,j}$ will be a tree decomposition of $G_{i,j}$ with all vertices of S_i occurring in some bag. For all i and j, we submit $(G'_{i,j}, S_i)$ to a list of subproblems.

When all S_i have been found, there is a lot of overlapping to be expected among the $\{G'_{i,j}\}_{i,j}$, and the trick is to exploit the overlaps by solving the subproblems $(G'_{i,j}, S_i)$ in the order of increasing size. If a $G'_{i,j}$ has size at most $k+1$, it has a one-bag rooted tree decomposition; we attempt to cover larger $G'_{i,j}$ with clique S_i with a number of smaller graphs $G'_{p,q}$ that are already known to be decomposable. In particular, we check for each $v\in G'_{i,j}\setminus S_i$ whether $G'_{i,j}$ can be covered by a family of decomposable subgraphs $\{G'_{p,q}\}_{(p,q)\in D}$ with separators $S_p\subset S_i\cup\{v\}$ for all $(p,q)\in D$, so that the $G'_{p,q}$ only overlap on $S_i\cup\{v\}$. In this case, we create a root $r_{i,j}$ and a bag $B_{r_{i,j}} = S_i\cup\{v\}$, and for each $(p,q)\in D$, we link $r_{i,j}$ to the root $r_{p,q}$ of the tree decomposition of $G'_{p,q}$. This yields a rooted tree decomposition of $G'_{i,j}$.

We prove by induction on the size of the subproblems $G'_{i,j}$ that every $G'_{i,j}$ of treewidth k will be decomposed: If $G'_{i,j}$ has size at most $k+1$ the trivial one-bag tree decomposition is found. Otherwise, let S_i be the separator that gave rise to $G'_{i,j}$; since $G'_{i,j}$ is a partial k-tree, a k-tree supergraph of $G'_{i,j}$ can be obtained by taking S_i as the initial basis for adding some vertex v and constructing k-trees $\{H_\ell\}_{1\leq\ell\leq m}$ based on the k-cliques $K_u = (S_i\cup\{v\})\setminus\{u\}$, $u\in S_i$. If K_u is used as basis for H_ℓ, then removing K_u separates G. Hence there exists a $p = p(\ell)$ such that $S_p = K_u$ and a $q = q(\ell)$ such that $G'_{p,q}$ derives from H_ℓ by edge deletion. $G'_{p,q}$ is smaller than $G'_{i,j}$ because it does not include u; by the induction hypothesis, it is successfully decomposed. The $\{G'_{p(\ell),q(\ell)}\}_{1\leq\ell\leq m}$ cover $G'_{i,j}$ and overlap only on $S_i\cup\{v\}$, therefore the algorithm finds a tree decomposition of $G'_{i,j}$. If G has treewidth at most k, then it is a subgraph of a k-tree with a basis S_i that separates G; we proved that a tree decomposition is found for each of the connected components $G'_{i,j}$ and that these tree decompositions can be merged into a tree decomposition of G. We have to consider $\binom{n}{k}$ candidates for

separators, and for each candidate S, we have to do work of order n to check whether removing S leads to more than one connected component. There are $O(n \cdot \binom{n}{k}) = O(n^{k+1})$ subproblems $(G'_{i,j}, S_i)$, and potential covers can be examined in time $O(n)$ by using appropriate pointer structures. Therefore the time complexity of the algorithm is $O(n^{k+2})$. This is not quite as devastating as it may appear at first, as we will see in Section 5.2.

Using Balanced Separators

We now return to balanced separators and introduce the framework underlying some of the more sophisticated algorithms for computing tree decompositions. When each component has size less than a constant fraction of the original graph's size, then using the procedure recursively on the components leads to a recursion depth of $O(\log n)$. The best algorithm to date for finding suitable separators has been discovered by Reed [Ree92], who shows how to find "approximate" separators in time $O(n)$. To beat his $O(n \log n)$ algorithm, a new idea like the one presented in the next section appears to be necessary.

Instead of adding edges to enforce that certain vertices end up in the same bag, the notion of a W*-separator* will allow us to specify the treatment of the distinguished vertices when the graph is cut. Given a graph $G = (V, E)$ and a vertex set $W \subseteq V$, we call $S \subseteq V$ a W-separator if every component of $G[V \setminus S]$ contains at most two thirds of the vertices from W. In other words, the component size is measured not by the total number of vertices but by the number of vertices from W. The following theorem from [Ree92] sets the stage for a procedure to compute a tree decomposition of width at most $4k + 3$ or to decide that the graph has treewidth greater than k.

Theorem 20.

(1) If $G = (V, E)$ has treewidth k, then for any $W \subseteq V$, there is a W-separator of size $k + 1$.

(2) If G contains for all $W \subseteq V$ a W-separator of size $k + 1$, then G has treewidth at most $4k + 3$.

□

The first part of the theorem follows from the fact that either many vertices of W are in a bag of some tree decomposition (choose that bag as W-separator of order $k + 1$) or we can find a bag in the tree decomposition such that at most half of the vertices of W are in any subtree. We prove the second part constructively by assembling a recursive function that actually computes a tree decomposition of width at most $4k + 3$. With each invocation, we pass

as parameters a graph G and a vertex set W of size at most $3k+3$ that must be contained in a bag of the returned tree decomposition.

We start by calling a subroutine that computes a W-separator S; the way S is computed is the distinguishing feature of the algorithms based on this approach. If there is no W-separator, then by case 1 of the theorem, G has treewidth greater than k. Otherwise, let C_i be the components of $G[V \setminus S]$ and let $G_i := G[C_i \cup S]$, i.e., the G_i are the different components including their overlap S. Obviously, these are the graphs for the next recursion; the W_i for the recursive invocations consist of the vertices that W shares with G_i plus all of S, so $W_i := (W \cap G_i) \cup S$. This is a reasonable definition since we can link the bags with W_i to a new bag consisting of $W \cup S$ to combine the tree decompositions of the subproblems; luckily, the size of W_i is bounded by

$$|W \cap G_i| + |S| \leq \frac{2}{3}|W| + (k+1) \leq \frac{2}{3}(3k+3) + k + 1 = 3k+3$$

and that of $W \cup S$ is bounded by

$$(3k+3) + (k+1) \leq 4k+4$$

These inequalities provide an explanation for the "magic" values for the width $4k+3$ in Theorem 20 and the bound $3k+3$ on $|W|$. It can also be seen that balancing the W-separators more precisely can get the bound on the width close to $3k+2$, but not smaller. Hence we need to shrink the output tree decomposition to the optimal width using one application of the algorithm by Bodlaender and Kloks described in Section 4.1. Before we move on, we recapitulate the significance of using W-separators instead of "plain" separators. The latter do tear up the graph in suitably sized chunks, but they fail to keep the separators of subsequent recursion levels close together in the tree decomposition, so that S can serve both as a knot for the G_i and as an interface to the other components of the graph at the next higher level.

4.3 The Algorithm by Bodlaender

A substantially different approach for computing tree decompositions was discovered by Bodlaender [Bod96a]; our presentation is based on notes by Hagerup [Hag98a]. The idea is to construct a recursive function that for instances (G, k) with a graph $G = (V, E)$ computes a tree decomposition of G by calling itself at most once. If $n = |V|$ is greater than some constant C with $C > k$, the algorithm proceeds in four stages, namely,

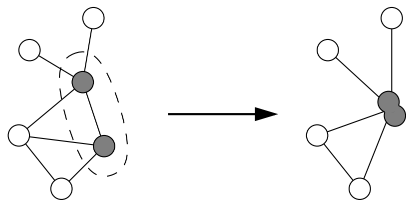
(1) reducing the input to a smaller graph,

(2) recursively computing an optimal tree decomposition of the reduced graph,

(3) patching the tree decomposition of the reduced graph into a non-optimal tree decomposition of the input graph,

(4) applying the shrinking procedure from Section 4.1 to convert the non-optimal tree decomposition into an optimal one.

This is to be the skeleton of a linear-time algorithm; reducing the graph in step (1) and operating on the tree decomposition in step (4) requires linear time per recursion step, so we must ensure that the graph is reduced sufficiently in each step to guarantee that the total work remains linear. Bodlaender meets this requirement by eliminating a constant fraction $1/d$ of the vertices in step (1), so that there are $\mathrm{O}(\log n)$ recursion steps and for a bound of cn on the work per recursion step on a graph of n vertices, the total work is bounded by $\sum_{i \geq 0} c\,(1/d)^i\, n = \mathrm{O}(n)$. The key to the reduction step is the observation that fusing pairs of adjacent vertices like this



allows a tree decomposition of the reduced graph to be transformed into a tree decomposition of the original graph by replacing the new vertex by the two old vertices in all bags. This gives a tree decomposition of the original graph because all restored edges are internal to a bag, and all other edges are covered as before; every vertex occurs in a connected component of the tree of bags, since restored vertices occur in the same component as the fused vertex. If fused vertices do not participate a second time in a fusion, then a width-k tree decomposition of the reduced graph gives rise to a transformed tree decomposition of width $2(k+1)-1 = 2k+1$.

Selecting pairs of adjacent vertices that can be fused simultaneously amounts to computing a matching in G, that is, a set of edges $M \subseteq E$ in which no two edges share an endpoint. Finding many pairs can be done by using a greedy algorithm to compute a matching M to which no further edges can be added. Does such a maximal matching always have size $\mathrm{O}(n)$, so that contracting all edges in M reduces the size of graph by a constant fraction? It does not. In fact, we need to handle the case of a small $|M|$ separately. As contracting edges does not help for small maximal matchings M, we resort in this case to another operation for reducing the size of the

graph. We try to identify vertices v in G that are *leaves* in the k-tree of which G is a subgraph; "leaves" in the sense that none of the k k-cliques that result from adding v during the construction of the k-tree is used to add a further vertex. Such vertices v have the property that there exists a tree decomposition of G where all of the (at most k) neighbors of v are contained in a single bag. The idea is to remove v and augment the reduced graph by auxiliary edges to enforce that the neighbors of v are in any width-k tree decomposition of the reduced graph. When we have a tree decomposition of the reduced graph, a new bag with v and its neighbors can be linked to the bag containing the neighbors, thus getting a tree decomposition of G.

To make this work, we have to show that for small maximal matchings M, a large number of leaf vertices can be identified. Vertices of degree one certainly qualify as leaf vertices, and we can repeatedly remove degree-one vertices until none remain. Since large matchings also benefit from this reduction, we perform this elimination step *before* the computation of M. We seek further vertices for which all neighbors occur in a single bag in all tree decompositions of graph G. In Chapter 2, we presented two lemmas that give sufficient conditions for vertices to occur together in a bag in any tree decomposition: Lemma 11 postulated that every clique occurs in a bag, irrespective of the size of the clique; and by Lemma 12, we know that for vertex sets V_1 and $V_2 \subseteq V$ that induce a complete bipartite subgraph of G, we can find in any tree decomposition either a bag containing V_1 or a bag containing V_2. The usefulness of Lemma 12 becomes obvious in the following consequence:

Lemma 21. If $u, v \in V$ have at least $k+1$ common neighbors, then in any tree decomposition of width at most k, some bag contains both u and v.

Proof. Vertices u and v on the one side and their common neighbors on the other side induce a complete bipartite subgraph of G. By Lemma 12, either u and v are in one bag, or their neighbors are. However, in the latter case, we can add edges between their neighbors without destroying the tree decomposition; in particular, we can turn them into a complete subgraph of size at least $k+1$. Since u is adjacent to all of them, we actually have a clique of size at least $k+2$, which contradicts the existence of a tree decomposition of width k: by Lemma 11, a $(k+2)$-clique would be contained in a bag. □

The plan is this: We will use Lemma 21 to single out vertices whose neighbors occur in a single bag in all tree decompositions of G, since such vertices certainly are leaf vertices. Removing those vertices may add unwanted degrees of freedom to the neighbors, which we combat by adding new edges to G and invoking Lemma 11. Remember that we are considering the case

of a small matching M; we let U denote the set of endpoints of edges in M and treat vertices in U as an immutable skeleton from which we try to pluck vertices outside of U—note that the neighbors of any vertex $w \in V \setminus U$ all lie in U because M is a maximal matching. For the purpose of analysis, we fix a rooted tree decomposition $(T = (X, F), \{B_x\}_{x \in X})$ of G and give names to the vertices outside of U: $w \in V \setminus U$ is a *bridge* vertex if it has neighbors $u, v \in U$ that in the fixed tree decomposition do not occur together in any bag; we call the pair $\{u, v\}$ a witness for w. The other vertices from $V \setminus U$ are called *internal* vertices; for every pair $\{u, v\}$ of neighbors of an internal vertex, there exists a bag containing both u and v. Our goal is to identify many vertices that are internal vertices in *every* tree decomposition of G. We proceed as follows:

(1) Let A denote a table of integers indexed by unordered pairs $\{u, v\}$ of vertices from U; in $A[\{u, v\}]$ we count the number of common neighbors of u and v that are in $V \setminus U$; we assume that A is initialized to contain all zeroes.

(2) We step through all vertices $w \in V \setminus U$. If the degree of w is at least $2k$, we ignore it. Otherwise, we consider all pairs $\{u, v\}$ of neighbors of w and increment $A[\{u, v\}]$. The cut-off value on the degree of w serves to bound the running time; e.g., processing a w of degree $\omega(\sqrt{n})$ takes time $\omega(n)$, compromising the linear running time of one recursion step and of the entire algorithm.

(3) We make a pass through A, adding an edge between each pair of nodes $\{u, v\}$ for which $A[\{u, v\}] \geq k + 1$. By Lemma 21, this does not invalidate any tree decomposition of G.

(4) We step a second time through all $w \in V \setminus U$, again skipping vertices of degree at least $2k$. If for all pairs $\{u, v\}$ of w's neighbors we have $A[\{u, v\}] \geq k + 1$, then the neighbors form a clique because we have added the necessary edges in (3). Knowing that the neighbors will stick together in any tree decomposition, we check whether w has degree at most k and remove it in this case. In the other case—if w has degree greater than k and for all pairs of its neighbors holds $A[\{u, v\}] \geq k+1$—we have found a proof that the graph has treewidth greater than k.

Even a small maximal matching M can have size $\Omega(n)$. If implemented in a straightforward manner, A therefore has size $\Omega(n^2)$, so that initializing and iterating over A would take time $\Omega(n^2)$. It is easy to avoid initializing A and to iterate only over non-zero entries, so we get a linear time bound. Moreover, Bodlaender gives a slightly more complicated data structure that manages with space $\mathrm{O}(n)$.

In a moment, we are going to take stock of how many vertices are left in the graph. In anticipation of a constant reduction factor, we review the complete algorithm: it eliminates degree-one vertices, finds a maximal matching, and checks whether the matching is sufficiently large. If it is, contracting the edges in the matching, invoking the procedure recursively, and expanding the edges again gives a tree decomposition of width $2k + 1$, which gets narrowed down to width k using the shrinking algorithm. All these operations take linear time. If the matching is too small, we add some edges to G and remove a number of vertices, recurse, and patch the previously deleted vertices back. The second case, too, is of linear complexity, so the algorithm computes tree decompositions in time linear in $n = |V|$ for k fixed. We can amend the algorithm not only to compute tree decompositions, but also to decide TREEWIDTH: If a graph has treewidth greater than k, then at some level in the recursion the shrinking will fail, or a removable vertex has degree greater than k, or the constant-size instance has treewidth greater than k.

To define the threshold size of M for branching into the first or the second case of the algorithm as well as to analyze the influence of k on the running time, we have to wrap up the argumentation for small matchings; then we know the fraction by which the graph is actually reduced in each recursive call. We bound the number of vertices left: We did not touch vertices in the matching, therefore we need to count all of them. Defining $m := |M|$ as the number of edges in the matching, this accounts for $|U| = 2m$ vertices. In step (2), we skipped vertices outside of M of degree at least $2k$. If in all of G, more than half of the vertices had such a high degree, then G would have more than nk edges, which by Proposition 8 is impossible.

After these bounds on the number of vertices we explicitly disregard, we estimate the number of vertices that rightly or inadvertently get ignored in step (4). Bridge vertices will never qualify for removal in step (4), because their neighbors are in at least one tree decomposition in different bags, so that Lemma 21 cannot be applicable. We assume that the tree decomposition is rooted and again write $T(v)$ for the subtree of T formed by the tree nodes whose bag contains $v \in V$. Since $T = (X, F)$ is rooted, $T(v)$ has a well-defined root $r(v) \in X$. For a fixed $u \in U$, consider the bridge vertices w with witnesses $\{u, v\}$ (for arbitrary $v \in U$) where w is in the bag $B_{r(u)}$ of the root $r(u)$ of $T(u)$. Since $B_{r(u)}$ contains u, it can contain at most k bridge vertices w; however, every bridge vertex must be in $B_{r(u)}$ for some u, hence there can be at most $|U|k = 2mk$ bridge vertices.

Internal vertices escape our scrutiny if their neighbors cannot be turned into a clique. We count the number of docking locations in U for internal vertices. Within one bag, there are at most $\binom{k+1}{2}$ possibilities to connect to two vertices. How many different bags can there be? We ignore bags that

are subsets of other bags and count the number of maximal bag sets with respect to vertices from U. There can be no more than $2m$ maximal bag sets since there are only $2m$ elements to start with. Therefore there are at most $2m\binom{k+1}{2} = mk(k+1)$ ways to dock an internal vertex, and to overlook an internal vertex, its docking location must not be used by more than k internal vertices in total, leading to a bound of $mk^2(k+1)$ on the total number of undiscovered internal vertices. Our calculation is summarized below:

vertices in U:	exactly $2m$
high-degree vertices:	at most $n/2$
bridge vertices:	at most $2mk$
overlooked internal vertices:	at most $mk^2(k+1)$
total:	at most $m(k^3+k^2+2k+2)+n/2$

We have traded one half the vertices of G against an expression depending linearly on m. By solving

$$m(k^3+k^2+2k+2) = \frac{n}{4}$$

for m, we get a threshold value of $m_t \approx n/(4k^3)$ to choose between the two ways to shrink the graph. If the size m of the maximal matching M does not exceed m_t, then the reduction amounts to

$$\frac{m(k^3+k^2+2k+2)+n/2}{n} \le \frac{m_t(k^3+k^2+2k+2)+n/2}{n} = \frac{3}{4},$$

whereas if $m > m_t$, contracting the edges of the matching reduces the size of the graph by a factor of

$$\frac{n-m/2}{n} \le \frac{n-m_t/2}{n} = 1 - \frac{1}{8(k^3+k^2+2k+2)} =: \frac{1}{d},$$

that is, in both cases the factor is bounded away from 1 ($1 > 1/d > 3/4$). Examining the individual computation steps once more, we see that each step can be done in time $\mathrm{O}(kn + S(n, 2k+1, k))$, where $S(n,k,\ell)$ is the time complexity of shrinking a tree decomposition of width k to width ℓ in a graph with n vertices (see page 75). Altogether, this leaves us with a bound of

$$\begin{aligned}
&\mathrm{O}\left(\sum_{i\ge 0} k\frac{n}{d^i} + S\left(\frac{n}{d^i}, 2k+1, k\right)\right)\\
&= \mathrm{O}\left(k^3\left(k + 2^{\mathrm{O}((2k+1)^2\log(2k+1)+(2k+1)^2\cdot k)}\right)\cdot n\right)\\
&= 2^{\mathrm{O}(k^3)}\cdot n
\end{aligned}$$

on the running time of Bodlaender's algorithm for computing tree decompositions. Even though our analysis is far from being tight, we clearly see that the bottleneck is the shrinking of tree decompositions. Observing that shrinking is only needed in the case of large matchings, we could bias the algorithm towards treating more matchings as small by raising the threshold m_t. Furthermore, Bodlaender remarks in [Bod96a] that it is possible to trade to certain extent the complexity in k against the complexity in n by restoring the contracted edges of the matching M in multiple iterations. If in each bag of the tree decomposition of the shrunken graph, only one fusion is reversed, then the resulting tree decomposition has width at most $k+1$—so we select up to one edge in each bag, restore it, and shrink the resulting tree decomposition of width at most $k+1$ to width at most k. We repeat these steps until all edges have been restored. How many iterations are required? We cannot always freely select one vertex in each bag to expand, therefore we will, in general, need more than $k+1$ iterations. Yet we can find a $(k+1)$-coloring of the expandable vertices in time $I(n,k) = nk$ using a greedy algorithm and extract from it an independent set of expandable vertices that has size $|M|/(k+1)$. Since $|M| = \mathrm{O}(n)$ and we chop off a factor of $k+1$ in each iteration, $\mathrm{O}(\log n)$ rounds suffice. The running time of this algorithm is

$$\mathrm{O}\big(k^3 \log n \,(kn + I(n,k) + S(n,k+1,k))\big) = 2^{\mathrm{O}(k^3)} \cdot n \log n,$$

but with smaller constants than before—in the shrinking subroutine, the path-decomposition characteristics on the trunk edges now come from a set of size $C_{k+1,k}$ instead of $C_{2k+1,k}$. Substituting $k = 2$ gives lower bounds $C_{3,2} \geq 3.58 \cdot 10^{14}$ as opposed to $C_{5,2} \geq 3.07 \cdot 10^{22}$ (see also Table 1 on page 57).

Chapter 5

Computing Tree Decompositions

5.1 Generating Test Cases

Evaluating the implementation of an algorithm means measuring the running time on a large number of "uniformly" selected samples from "typical" inputs. Since we set out without a specific application of tree decompositions, we could not make any assumptions about the source of inputs. Therefore we had to generate "random" graphs of a given size and a given bound on the treewidth, so that we could investigate how tree-decomposition algorithms compare on arbitrary input for various graph sizes and treewidths.

We begin our discussion on the generation of test cases with a few remarks on the selection of random inputs in general; we write I_n for the set of binary representations of input objects of size n, assuming that every $x \in I_n$ has length $\mathrm{O}(n \log n)$, i.e., x can represent $\mathrm{O}(n)$ numbers of magnitude $\mathrm{O}(n)$. In most problems in computer science, several different input bit strings are considered equivalent: for SAT, we pose essentially the same problem when we rename variables or reorder the clauses of a CNF formula. Similarly, in many graph problems, the order in which the vertices are listed in the input is irrelevant, that is, isomorphic graphs constitute the same problem instance. Formally, the set I_n is partitioned into equivalence classes $[x]$ of the objects y that are equivalent to x; choosing a random input of a given length then amounts to selecting an equivalence class $[x]$ uniformly at random and returning an arbitrary $y \in [x]$. Graphs $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$ are called *isomorphic*, if there exists an isomorphism $\sigma : V_1 \to V_2$ such that $(u, v) \in E_1 \Leftrightarrow (\sigma u, \sigma v) \in E_2$. The equivalence classes of general (*labeled*) graphs under graph isomorphism are called *unlabeled* graphs; likewise, there are unlabeled trees and unlabeled rooted trees, the latter with the isomorphisms restricted to leave the root vertex r invariant, $\sigma r = r$. In Figure 32, a representative of each unlabeled tree with four vertices is shown; Figure 33 lists the unlabeled rooted trees with four vertices. For these three types of

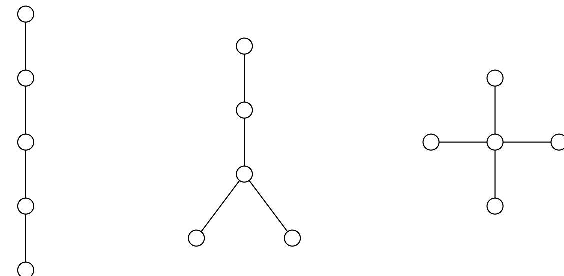


Figure 32: The unlabeled trees with four vertices.

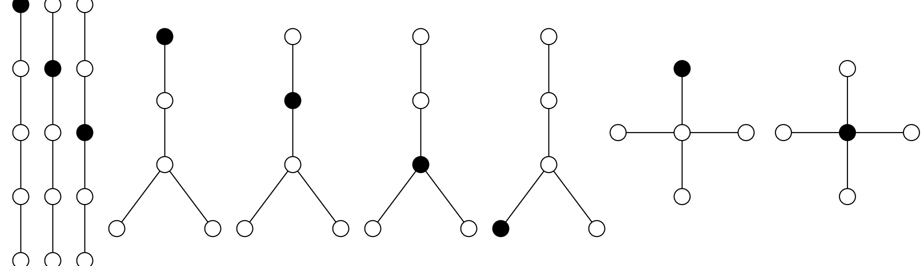


Figure 33: The unlabeled rooted trees with four vertices (roots shown in black).

unlabeled graphs as well as for labeled trees, there are efficient selection algorithms [NW78, Wil81, Tin90], that is, given a source of random numbers, these algorithms produce an arbitrary labeled graph (not always the same) of an equivalence class that is selected uniformly at random.

As nice as such selection procedure may appear from a theoretical point of view, judging the performance of an algorithm based on inequivalent test cases entails a certain danger: an actual implementation is likely to depend significantly on the order in which the input is presented. Even the [ACP87]-algorithm (discussed in the next section), which iterates through all separators of the graph, shows deviations of almost 100% when the input graph is permuted, as can be seen in Figure 34. Nevertheless, we did extend selection algorithms for labeled trees and unlabeled rooted trees to produce labeled k-trees and unlabeled "rooted" k-trees uniformly at random, assuming k and the number of vertices n to be fixed. However, our results do not generalize to partial k-trees, and due to their minor theoretical and practical significance, we refrain from presenting our approach in detail; suffice it to say that there is a bijection between a labeled k-tree G and a tuple (T, R, l) consisting of a labeled tree $T = (X, F)$, the specification of a "root" $(k+1)$-clique R and an edge labeling $l : F \to \{1, \ldots, k\}$, which assigns an integer to every edge of

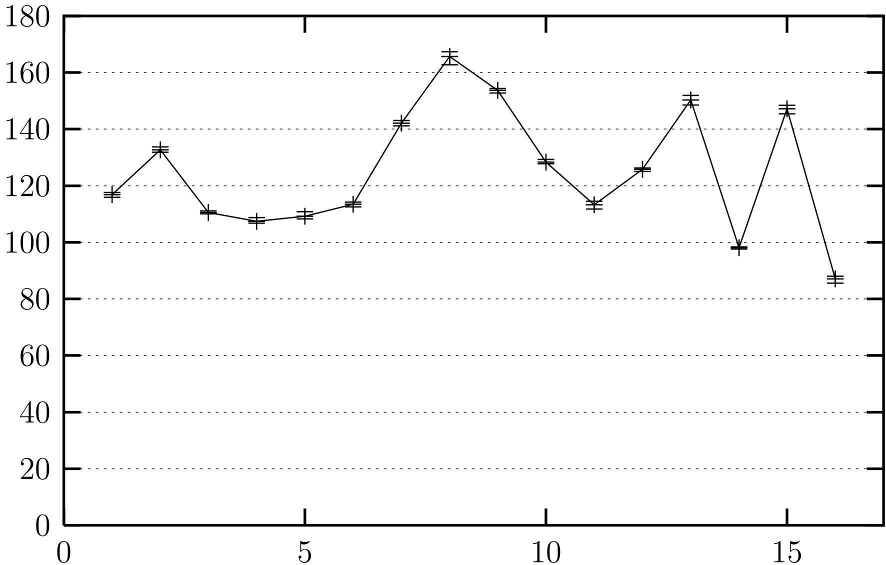


Figure 34: The running time of the [ACP87]-algorithm for 16 random permutations of the graph depicted in Figure 35. Error bars indicate the greatest, smallest and average time for running the same instance multiple times; the deviations are caused by transient changes in the operating environment and are obviously negligible.

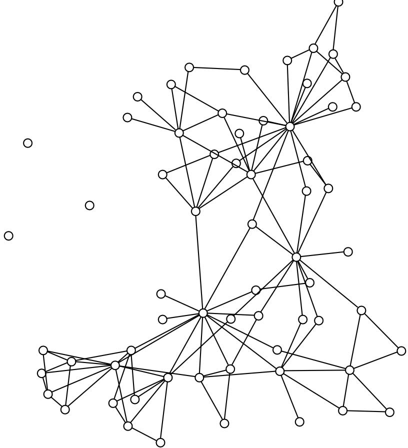


Figure 35: A graph of treewidth 3 for measuring the dependency on the input order of the [ACP87]-algorithm

T. Because of this mapping, selecting a labeled k-tree uniformly at random amounts to generating such a tuple uniformly at random.

By a unlabeled rooted k-tree, we mean the equivalence class of labeled k-trees G under isomorphisms σ that map the vertices of a given $(k+1)$-clique R in G unto themselves, i.e., $\sigma v = v$ for all $v \in R$; it can be shown that representatives of every unlabeled rooted k-tree occur equally often in a certain set of tuples (T, R, l) consisting of a representative T of an unlabeled rooted tree, an arbitrary root clique, and an edge labeling l from a certain subset of permitted edge labelings. Choosing such a tuple uniformly at random and constructing the corresponding labeled k-tree then gives a representative of a uniformly chosen unlabeled rooted k-tree.

Our "grut" package of graph utilities has programs for creating labeled and unlabeled rooted trees uniformly at random, for creating random k-trees using labeled trees as skeleton, for deleting a given number of edges at random, and for randomly permuting the vertices of a graph. Further information on this software is provided in Section A.1 in the appendix.

5.2 The Algorithm by Arnborg, Corneil, and Proskurowski

Only the first tree-decomposition algorithm described in the previous chapter, the $O(n^{k+2})$ procedure from [ACP87], can do without the Bodlaender-Kloks shrinking algorithm. In spite of its enormous asymptotic bound, our implementation of the [ACP87]-algorithm did work quite well for $k = 2$, and meaningful results could be obtained for k up to 4. Figures 36 and 37 show the results on a few benchmarks. The test cases were constructed by generating for each n three random unlabeled rooted trees with $n - k$ nodes, and using these as skeletons for random k-trees. From those "maximal" graphs, edges were deleted in bunches of 10% of the k-tree's edges, getting 11 test cases from each of the three skeletons, or 33 for each n. The tests were run on the same computer as the path-decomposition algorithm (see page 60), but with limits on running time (30 minutes) and on the main memory (597 megabytes to allow three tests to run concurrently). The statistics in Figures 36 and 37 show for each n and k the greatest measurement—no entry means that either for this choice of n and k, at least one test run violated the limits or that all runs had too small running time to yield meaningful measurements.

Except for interrupting the tabulating of subproblems as soon as a solution is found and running the [ACP87]-procedure separately on each connected component of the input graph, no further optimizations were implemented; in particular, no bounds on k were assumed.

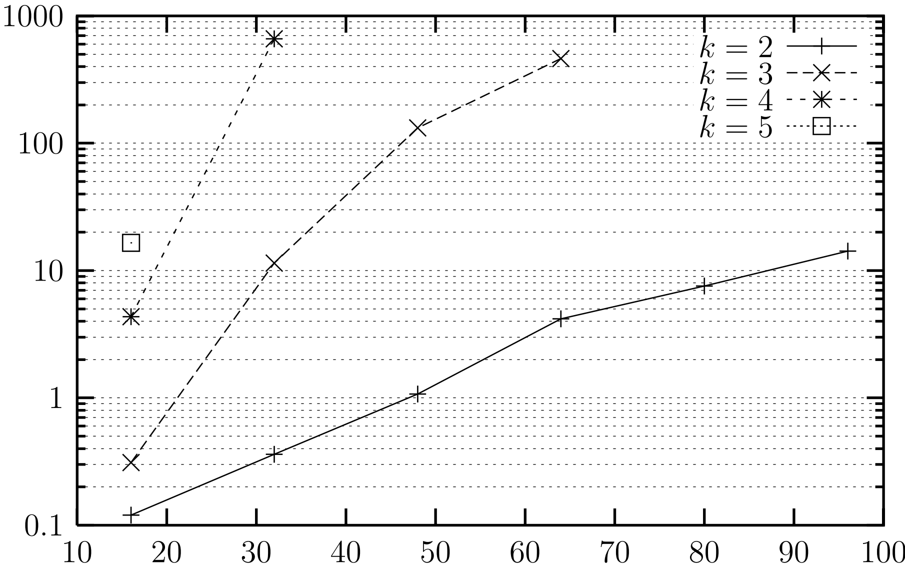


Figure 36: Running time in seconds of the [ACP87]-algorithm for different treewidths k, plotted against the number of vertices n.

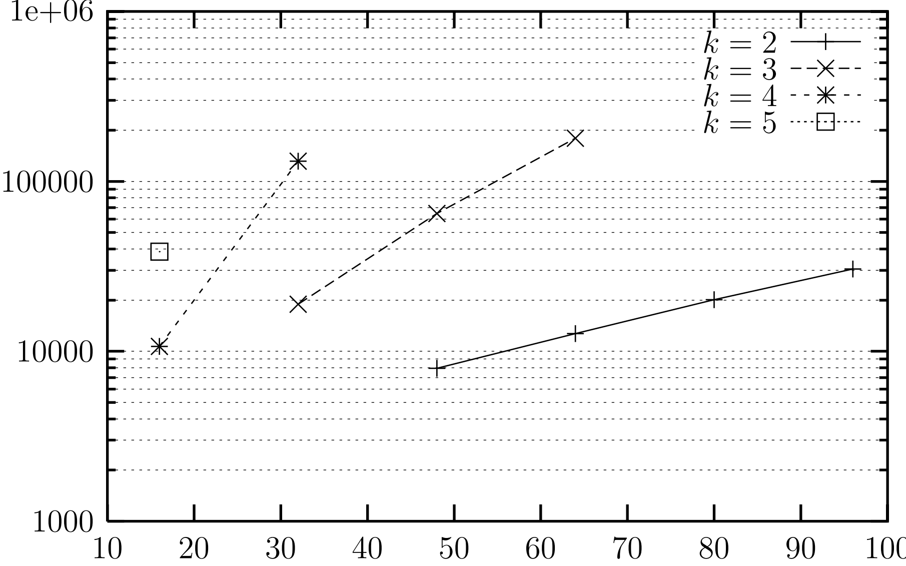


Figure 37: Memory consumption in kilobytes of the [ACP87]-algorithm.

Chapter 6

Conclusions

6.1 Shrinking Tree Decompositions Is Not Feasible

The more advanced separator algorithms and Bodlaender's linear-time algorithm depend heavily on a procedure for reducing tree decompositions from non-optimal bounded width to the minimum width. This procedure was provided in Section 4.1 in form of the shrinking algorithm by Bodlaender and Kloks, which is an extension of their path-decomposition algorithm for graphs supplied with a bounded-width tree decomposition. In Chapter 3, we analyzed this path-decomposition algorithm and found that the construction cannot be simplified much; because of its importance as a fundamental building block of tree-decomposition algorithms, we put a large effort into implementing it as efficiently as possible. Despite our quite significant improvements such as the elimination of redundant characteristics, pipelining, and caching, our experiments led us to the conclusion that path decompositions of width greater than 3 cannot be computed using this approach even for graphs of 16 vertices. The tree-decomposition shrinking algorithm makes extensive use of the combination procedures for path-decomposition characteristics; hence this algorithm, too, must be impractical for widths greater than 3. Indeed, the number of potential characteristics grows even faster in the case of reducing a tree decomposition from width k to ℓ than in computing a path decomposition of width ℓ from a tree decomposition of width k: the asymptotic bounds on the number of characteristics are

$$2^{\Theta(k^2 \log k + k^2 \cdot \ell)} \cdot n \qquad \text{and} \qquad 2^{\Theta(k \log k + k \cdot \ell)} \cdot n,$$

respectively. In Chapter 4, we derived a lower bound of $3.58 \cdot 10^{14}$ for the maximum number of different characteristics at a tree node when reducing width-3 tree decompositions to width 2. This huge figure strongly suggests that even a single call to the Bodlaender-Kloks shrinking procedure is not

feasible, much less repeated invocations as in Bodlaender's linear-time algorithm.

On the other hand, Sanders [San96] gives a linear-time algorithm for computing tree decompositions of width 4 and he considers it to be practical; for widths below 4, simple graph-reduction algorithms were derived by Arnborg and Proskurowski [AP86]. Hence we conclude that neither tuning the separator-based tree-decomposition algorithms nor implementing Bodlaender's algorithm would extend the range of tractable problem instances beyond the widths for which special-purpose algorithms exist.

6.2 Further Directions

We set out to investigate the practical value of tree-decomposition algorithms of the most general type, which for any input graph G and any requested width k compute a tree decomposition of width k or state that the graph has treewidth greater than k. The sobering result is that computing optimal-width tree decompositions is—with today's algorithms and computers—intractable for widths greater than 4 and graphs larger than, say, 16 vertices. We already mentioned that for each value of k up to 4, algorithms based on graph reduction have been constructed; Sanders claims that despite the need to differentiate between some one hundred special cases, there are no large hidden constants in the analysis of his algorithm. However, even if it were not practical, the algorithms for treewidth up to 3 certainly are; only six rules for rewriting graphs suffice to define the graphs of treewidth at most 3 as those graphs that can be rewritten to the empty graph.

It was beyond the scope of this work to implement algorithms for particular treewidths, not least because an efficient implementation would probably not be straightforward. Moreover, once we deviate from our original objective of examining the practicality of general and complete tree-decomposition algorithms, there are plenty of alternative ways to proceed. For certain real-world applications, tree decompositions of non-optimal width might be acceptable or further knowledge about the input could be used to improve the calculation of the bounds or to speed up the present algorithms or to devise completely new algorithms. Moreover, in three-dimensional spring-embedder layouts of dense k-trees, their "tree structure" appears to unfold (Figure 38), and this observation might help to develop useful heuristics. All these approaches require a thorough analysis of the concrete application to identify further properties of the problem at hand; the huge constants arising from the general techniques of using tree decompositions lead us to the conclusion that the generality of the treewidth theory makes it—without considerable specialization—unusable in practice.

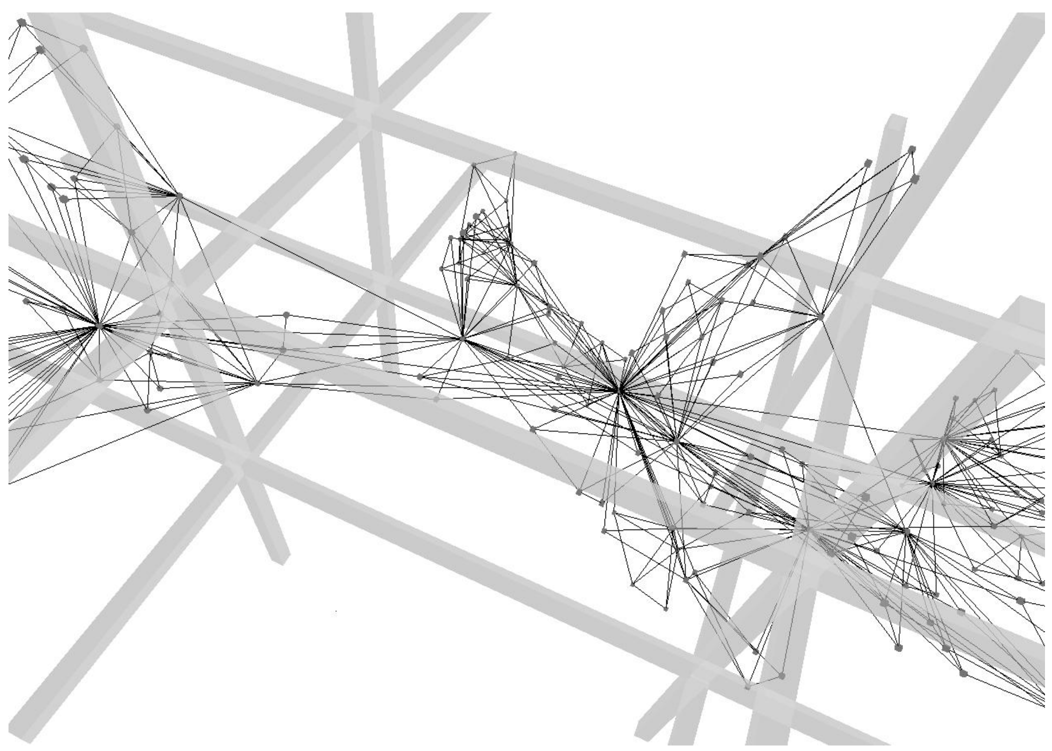


Figure 38: Snapshot of a three-dimensional “Virtual-Reality” rendering of a 3-tree. We have added a large grid for aiding orientation when navigating through the graph using standard viewing software. Our implementation of a spring-embedder layout algorithm and a program for translating graphs with a layout into Virtual-Reality scenes are part of the “graph utilities” package described in the appendix.

6.3 Comments on the Development Tools

The majority of the software developed as part of this work was written in the C++ programming language [SE90] using LEDA, a library of efficient data types and algorithms [MNSU98]. In this section, we address drawbacks of this approach in the hope that our comments will be useful for future experimental algorithm implementations. All things considered, we consider C++ and LEDA to be among the best tools currently available, yet with much potential for improvement. Joyner [Joy96] gives a comprehensive list of the shortcomings of the C++ language in general; therefore we discuss only the issues that arose in our programming with LEDA.

C++ Standardization

At the time of writing, the C++ standard (ISO/IEC 14882:1998) has been officially approved for two months, though not yet published. Until the standardization effort, the C++ language evolved through extensions that the inventors of C++ at AT&T Research Labs made to their `cfront` compiler, extensions that were approximately copied by several compiler vendors. During the process of standardization, which started in 1989, significant changes were made to the language, and subsequent drafts of the standard were followed to a varying extent by the different compilers. Consequently, it appears that the standardization of C++ led for many years to a less stable specification of the language, and this will change only slowly as vendors catch up with the final standard. Our programs written with LEDA were affected in three ways by the evolution of the C++ language:

- Each compiler release with incompatible changes to the language necessitated the adaption of all source code. For example, the scope of variable declarations in the `for` loop was changed from GNU C++ version 2.5 to 2.6, making the code in Figure 39 illegal, whereas previously, a redeclaration of `i` was considered to be an error. More obscure changes, such as the abolishment of "guiding declarations," the introduction of the `typename` keyword, and modifications to the resolution of overloaded functions (i.e., functions with the same name but different argument types) caused compilation errors that were hard to diagnose.
- Releases of LEDA always supported the compiler versions that were current at that time. As a consequence, there is usually only a small range of compiler versions with which any given LEDA release works; hence, the "C++ dialect" of our programs is largely determined by the choice of the version of LEDA.

```
for (int i=0; i<10; ++i) {
    // do something
}
for (i=0; i<10; ++i) {
    // do something
}
```

Figure 39: Originally, variables declared in the head of a `for` loop belonged to the surrounding scope, so the code snippet above was correct. In ISO C++, however, `i` belongs to the scope of the body, so that it is undeclared in the head of the second `for` loop.

- The ISO C++ standard defines classes for basic data structures such as arrays, lists, and sets. The access to these structures using "STL iterators" differs significantly from the "LEDA style" of using macros such as `forall`. In the early stages of our implementation, LEDA did not support the new style of accessing data structures, so that the interoperability with the new standard library was limited. Moreover, the LEDA style of enumerating elements was awkward to implement for our own classes. It appears that in the current release, STL iterators are, for the most part, supported.

C++ Compilation Speed

The great complexity of the C++ language is reflected by large compilation times. As an example, recompiling after making a change to a certain source file in the "tdecomp" project took well over half a minute on a SUN Ultra 1 workstation; this was with all compiler optimizations disabled. Even for medium size test cases, compiler optimizations were highly desirable, but enabling them increased the compilation time by a factor of three. When changes involved header files, the delay was even greater because a header file is usually included by several source files, each of which needs to be recompiled.

Specifically, our criticism is that in C++, small changes often entail compilation times that grow with the size of the project. The modification of an inline function causes all clients of a class to be recompiled, even when optimizations are disabled. The overhead of parsing library declarations is reduced by some compilers using "pre-compiled headers", yet one would expect the compiler to find out whether a change affects a class interface and thus dependent classes, and only in this case to recompile the dependent classes. However, C++ is designed towards only examining one source file at

a time, all but precluding project-global analysis.

Tracking and Copying Objects

The memory management of C++ turned out to be a substantial impediment to implementing large algorithms efficiently. Consider the data structure for characteristics of partial solutions in the generic tree-automaton algorithm (Section 2.4). Combination procedures construct C++ objects representing such characteristics; an object may get inserted into the cache of the current tree node, or get passed to the parent of the tree node, or be stored with a partial solution that is being generated. All in all, references to these objects are kept in many places and not all objects are treated the same way; however, since space is a scarce resource for the path-decomposition algorithm, we need to release the memory occupied by a characteristic soon after it is not referenced anymore. The lifetime of the objects representing characteristics is not determined by a static scope, so they need to be dynamically allocated and dynamically freed. For dynamic memory management, C++ offers the new and delete operators, which normally allocate and release memory using the C functions malloc and free. In other words, the programmer has to find out when an object is no longer used, and then call delete. For objects with such "diverse" lifetimes as characteristics, this is a difficult task; we were forced to count the references to each object and dispose of it as soon as this count reached zero. Similar reference counters are manually implemented in many places in current C++ libraries, such as the GNU implementation of the ISO C++ string class and the LEDA classes integer and rational. Nonetheless, this approach has two distinctive drawbacks: circular references cannot be detected, and there is no uniform way to implement reference counting. One way to furnish reference counting to arbitrary classes is to design a reference template ref<*class*>, which behaves like a pointer to an object of class *class*, but calls delete on the object when the last ref<*class*> reference to it is discarded. This approach fails due to clashes with the type system of C++; for instance, there is no way to make ref<*parent*> a superclass of ref<*child*>. Analogous obstacles rule out solving the problem by bequeathing classes with a reference counter from an ancestor class refcountable and, in any case, the programmer cannot be forced to handle "raw" pointers correctly.

When using LEDA—or, for that matter, any other library of data structures—the lack of a garbage collector leads to redundant copying of objects: Inserting a large object, such as the representation of a characteristic, into a LEDA list causes the object to be copied into the set; LEDA cannot store a pointer to the object because the original object might be delete-d just

after the insertion. Removing the first element of a list and storing it in a variable again involves a copy operation; worse, any object returned by a function needs to be copied, as we explain using the example in Figure 40. `A` is some class with a copy constructor and a constructor taking no argument; the function `main` calls `func`, reserving space for the return value on the

```
A func()
{
    A a1, a2;
    // some computation
    if (condition)
        return a1;
    else
        return a2
}

main()
{
    A result;
    result = func().do_something();
}
```

Figure 40: Example of redundant copy operations.

stack. After `func` is entered, the constructor taking no argument is called for the objects `a1` and `a2`, with memory allocated on the stack frame of `func`. At each of the return statements, the copy constructor is called with `a1` or `a2` as parameter to create an object in the area reserved for the return value. This copy operation could be avoided if it were clear at the entry of `func` which object would be returned; in Pascal, for example, the implicit return variable gets the name of the function and so the problem is avoided. A sophisticated C++ optimizer might be able to save most copy-constructor calls, yet the casual C++ programmer is probably not aware of this problem and the compilers we checked (GNU and Sun) did not optimize it away.

Pipelining

Pipelining is a programming technique to avoid storing intermediate results, to compute results "just in time," and to parallelize producers and consumers of data. To that end, subroutines that normally return list data structures are converted to compute the elements of the list one by one. The caller does

```
list<characteristic> combine(...)
{
    loop_state i;
    for (init(i); valid(i); next(i)) {
        // construction of a candidate of a characteristic
        if (found_characteristic) {
            full_set.insert(new_characteristic);
        }
    }
    return full_set;
}
```

Figure 41: A C++ function with a loop computing a set of full characteristics.

```
characteristic next_combination(loop_state &i, ...)
{
    if (!initialized) {
        init(i);
    }
    goto inside;
    for (; valid(i); next(i)) {
        // construction of a candidate of a characteristic
        if (found_characteristic) {
            return new_characteristic;
        }
    inside:
        ;
    }
    // all characteristics have been returned
    return no_more_characteristics;
}
```

Figure 42: A construct for pipelining the loop in Figure 41. Each invocation of this function returns one new characteristic or states that there are no further characteristics.

not enumerate the elements of the list, but requests further elements from the subroutine, so that the list is never completely instantiated. Moreover, only the elements of the list that are needed by the caller are actually computed; in parallelized setting, the caller and the subroutine can work concurrently, i.e., while the caller processes one element of the list, the subroutine can already produce the next. Pipelining is used on many levels of computing, such as in arithmetic circuits of microprocessors [PH90] and in database systems [SKS97].

Converting source code to take advantage of pipelining requires some unclean workarounds in C++. Namely, converting a loop as in Figure 41 with (possibly large) procedures `init`, `valid` and `next` necessitates a construct like that of Figure 42 where the variables `initialized` and `i` must be conserved across subsequent requests for the next characteristic. The `goto` can be avoided by turning the `for` loop into a `do {...} while`-loop, however, the problem of maintaining the state of the loop counter `i` remains and gets much worse for nested loops. It is possible to use multithreading in C++, but the language lacks coroutines [Mar80], which would allow an implementation of pipelining that is both clean and efficient.

An Alternative

After investigating several programming languages, we found that the Eiffel programming language [Mey92] remedied all the problems we encountered in programming C++. It has a powerful object system with multiple and repeated inheritance, exception handling, generic classes (corresponding to templates in C++), and garbage collection. Among its unique features is the support for "Programming by Contract," where preconditions and postconditions of functions and class invariants are specified within the language, allowing them to be inherited by functions in derived classes and extracted by automatic documentation tools. The term "Programming by Contract" stems from the interpretation that when object `A` invokes method `m` of object `B`, `A` guarantees that the parameters satisfy the preconditions of `m` and `B` is committed to ensure that the postcondition of `m` will be met and the class invariant of `B` is preserved. Moreover, Eiffel requires global program analysis to ensure correctness, which has the useful byproduct that all current Eiffel compilers support incremental compilation; hence recompilation times remain in relation to the changes made. Finally, Eiffel does not have coroutines, but a language extension for an object-oriented equivalent of coroutines has been proposed by Meyer [Mey97].

When we came to the conclusion that in C++, we could not improve memory-management while maintaining the readability and extendibility of

the source code, we made an effort to port the path-decomposition algorithm to Eiffel, but due to time constraints, this project was eventually suspended—the lack of a data structure library like LEDA could not be compensated for by a one-man effort.

Appendix A

Notes on the Software

We pursued several lines of development. In Section A.1 we describe our tools for graph generation, which draw on the techniques presented in Section 5.1. Using the C++ language and the LEDA library [MNSU98], we implemented the [ACP87] tree-decomposition algorithm, the generic tree-automaton technique for solving problems on graphs of bounded treewidth, and as instances of the latter, algorithms for COLORING and computing path decompositions. Implementation notes on these programs are given in Section A.2. The source code, some 12,000 lines of code, is included on electronic media with all official copies; it is available as well on the Internet at

`http://www.mpi-sb.mpg.de/~roehrig/dipl`

The unfinished port to the Eiffel programming language (3,500 lines of code) is available on request.

A.1 Graph Utilities

We needed utilities to generate and manipulate a large number of graphs in a scriptable environment. For this purpose, we created the "grut" package of command line graph utilities. To ensure interoperability with the "graphlet" interactive graph editor [Him96] and LEDA, we chose the GML file format [Him97] to store graphs. All programs in the grut package fulfill a narrow purpose, such as outputting a random tree or annotating a graph with a layout; they take their parameters from the command line, read input from the standard input and write output to the standard output. As an extension to GML, they all maintain a log of changes made to a graph, so that the genesis of test cases can always be determined. We reproduce here the `README` file from the source code.

From: Hein Roehrig <hein@acm.org>
Time-stamp: "1998-09-23 11:20:29 roehrig"

COPYRIGHT

grut - GRaph UTilities
Copyright (C) 1998 Hein Roehrig

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

INTRODUCTION

This is a snapshot of the C++ and Perl command line graph utilities developed for my master's thesis. The underlying simple graph format is GML, as described in

http://www.fmi.uni-passau.de/
    archive/archive.theory/ftp/graphlet/GML.ps.gz

As an extension to GML, the utilities maintain a history of the changes made to graph.

INSTALLATION

- Prequisites: GNU make, gcc 2.8 or egcs, perl 5. Optionally autoconf, automake and libtool. LEDA is not used.

- in the following, the directory of this file will be referred to as $srcdir.

- make a separate compilation directory, now referred to as $compdir

- configure the package

```
  cd $compdir
  $srcdir/configure --disable-shared

  If you want to use a different C/C++ compiler, do the following:

  CC=/opt/egcs-1.0.1/bin/gcc CXX=/opt/egcs-1.0.1/bin/c++
      $srcdir/configure --disable-shared

- cd $compdir; make

  for debugging :
    make CXXFLAGS="-pipe -g -Wall"

  for profiling:
    make CXXFLAGS="-pipe -O -fno-inline -DNDEBUG -pg -Wall"

  for production:
    make CXXFLAGS="-pipe -O3 -DNDEBUG -Wall"


RUNNING

All programs take their input from stdin and write the output to
stdout. Errors and other messages are sent to stderr. The programs
take a "-v" switch to increase verboseness, and those using random
numbers take a "-S integer" switch to define the seed. Other options
depend on the program and are given by running the program with the
"--help" flag (you are invited to have a look at the source code as
well).

Generation:
makepath    generate a path of given length
makecactus  generate a cactus of given size
rtree       generate random trees

Modification:
tree2ktree  generate a k-tree from a tree at random
thinout     randomly delete edges
permute     randomly permute nodes
id2label    set the node labels to the node ids
label2id    set the node ids from the node labels

Layout:
layout3d    3D spring embedder
gml2vrml    convert a GML graph with 3D layout to VRML

Other:
graphstat   give statistics of a graph
```

A.2 Tree Decomposition and Path Decomposition

All C++ software for computing and verifying tree decompositions and path decompositions is contained in the "tdecomp" package. An overview of the distribution and installation instructions are contained in the `README` file, which follows.

```
From: Hein Roehrig <hein@acm.org>
Time-stamp: "1998-09-23 11:21:37 roehrig"



COPYRIGHT

tdecomp - Programs for Tree and Path Decomposition
Copyright (C) 1998  Hein Roehrig

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or (at
your option) any later version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307
USA



INTRODUCTION

This is a snapshot of the C++ tree decomposition and path
decomposition programs developed for my master's thesis. At some point
during development, I came to the conclusion that using C++ was not a
good idea since it has serious deficiencies such as missing garbage
collection. Unfortunately, I did not get around to re-implement
everything in Eiffel; however, the present code proves rather well
that the Bodlaender-Kloks algorithm is impractical.

If you would like to look at the source, the interesting parts are in
the following files:

tautomat.h contains the generic algorithm for solving problems using a
        tree decomposition.

iseq.h, iseq.cc contain the code dealing with T-sequences.
```

```
pdc.h contains the combination algorithms for path decomposition

coloring.cc contains the combination algorithms for k-COLORING.

tdecomp1.cc contains the the n^(k+2) tree decomposition algorithm by
        Arnborg, Corneil and Proskurowski.

The primary benchmarking tools are

pdcpathG: scheduler for series of benchmarks

cfman: configuration manager for executing the same test suite on
        multiple variations of the same algorithm

ppac: measures the resource usage (elapsed wall clock time, CPU
        cycles, memory consumption)



INSTALLATION

- Prequisites: GNU make, LEDA 3.7, gcc 2.8 or egcs. Optionally: perl
  5, autoconf, automake and my grut graph utilities. The sources can
  be back-ported to LEDA 3.5 and 3.6 without much work; however,
  earlier versions of gcc and most other C++ compilers don't do
  because they do not support features like member templates.  Note
  also that Quantify up to version 4.2 does not work with gcc 2.8 (I
  had to learn it the hard way...).

- in the following, the directory of this file will be referred to as
  $srcdir. The location of LEDA will be referred to as $ledadir.

- make a separate compilation directory, now referred as $compdir

- configure the package

  cd $compdir
  $srcdir/configure --with-leda=$ledadir

  if LEDA is installed in $ledadir/include and $ledadir/lib, or

  $srcdir/configure --with-leda-include=/LEDA/INSTALL/incl
      --with-leda-lib=/LEDA/INSTALL/solaris/g++/lib

  If you want to use a different C/C++ compiler, do the following:

  CC=/opt/egcs-1.0.1/bin/gcc CXX=/opt/egcs-1.0.1/bin/c++
      $srcdir/configure --with-leda=$ledadir

- cd $compdir; make

  for debugging :
    make CXXFLAGS="-pipe -g -Wall -Wno-reorder"
```

```
  for profiling:
    make CXXFLAGS="-pipe -O -fno-inline -DNDEBUG
                   -DLEDA_CHECKING_OFF -pg -Wall -Wno-reorder"

  for production:
    make CXXFLAGS="-pipe -O3 -DNDEBUG
                   -DLEDA_CHECKING_OFF -Wall -Wno-reorder"

- optionally (may need huge amounts of memory/time):

  make check



RUNNING

- Note: numbers referring to vertices in the output are the values
  from the GML "id" field.

- At the beginning of the individual source files, debugging and other
  options can be set via preprocessor directives

- For running the programs, the LD_LIBRARY_PATH variable probably
  needs to point to the location of the LEDA DLLs. For the test and
  benchmark scripts, you should also set TIMECMD and srcdir. Of
  course, all shell variables need to be exported to the environment.

- All programs dump core on errors and on ^C. Therefore you should
  consider to set ulimit -c0 to switch off core dumps.

- All programs write their output to stdout and diagnostic messages to
  stderr. All programs take the "-v" switch to increase verboseness.

- The tdecomp and the pdecomp programs either compute or verify tree
  decompositions/path decompositions. Verify mode is specified with
  the "-V" switch; without this switch, computation mode is selected.
  For computation and optionally for verification, a "-k integer"
  switch can be given to indicate the required width of the
  decomposition.

        tdecomp -vk2 graph1.gml > graph1-tdc.gml

  computes a tree decomposition of width 2 of graph 1, with lots of
  information during the computation, and with the output tree
  decomposition written to graph1-tdc.gml.

        pdecomp -vk3 graph1.gml graph1-tdc.gml > graph1-pdc.gml

  computes a width 3 path decomposition of graph1.gml using the tree
  decomposition graph1-tdc.gml, and write the output to
  graph1-pdc.gml.
```

```
      tdecomp -vVk2 graph1.gml graph1-tdc.gml

  verbosely verifies the tree decomposition, and

      pdecomp -vVk3 graph1.gml graph1-pdc.gml

  verifies the path decomposition.

- The coloring program works similarly, except that the output
  consists of a coloring and the -k parameter indicates the number of
  permitted colors.

- The file format for tree decompositions is as follows: The tree is
  written out as a GML graph, and nodes of the tree have a GML keyword
  "bag" of type "list", in which the graph vertices in the bag of that
  tree node are given. E.g.

  graph [
    directed 0
    node [ id 0 bag [ node 7 node 100 ] ]
    node [ id 1 bag [ node 4 node 7 ] ]
    node [ id 2 bag [ node 4 ] ]
    edge [ source 0 target 1 ]
    edge [ source 1 target 2 ]
  ]

  would be a (width 1) tree decomposition of graph

  graph [
    directed 0
    node [ id 4 ]
    node [ id 7 ]
    node [ id 100 ]
    edge [ source 4 target 7 ]
    edge [ source 7 target 100 ]
  ]

- For benchmarking, perl and grut are needed. The test cases are
  generated using make* scripts and executed using the corresponding
  pdc* scripts. If you are reading this not much later than summer
  1998, beware that the dimensions of the test cases are chosen to go
  to the limit of the largest machine I had access to.



GRAPHS

test cases for tree decomposition verification
----------------------------------------------
g000.gml t000.gml
g001.gml t001.gml
g002.gml t002.gml
```

```
test cases for tree decomposition computation
---------------------------------------------
g003.gml t003.gml
g004.gml t004.gml
g005.gml t005.gml
g006.gml t006.gml
g007.gml t007.gml
g008.gml t008.gml
g009.gml t009.gml
g010.gml t010.gml
g019.gml t019.gml

test cases for coloring (tree automaton)
----------------------------------------
g003.gml t003.gml
g011.gml t011.gml
g011.gml t012.gml

test cases for path decomposition
---------------------------------
g013.gml t013.gml handcrafted, 6 nodes, pathwidth 2
g016.gml t016.gml from path20, width 2, not thinned out
g020.gml t020.gml pathwidth 3, treewidth 3, 11 nodes
g022.gml t022.gml tree of 5 nodes in Y form
g023.gml t023.gml cycle of 5 nodes with two "dangling" nodes
g025.gml t025.gml based on g016.gml, with 1/4 of the edges removed
g026.gml t026.gml graph consisting of 4 stacked triangles, tw 2, pw 3
g032.gml t032.gml test case for single split error in computing results
g038.gml t038.gml for profiling, path 128, width 2, 25% deleted

misc graphs
-----------
g015.gml t015.gml generated, treewidth 4
```

Bibliography

[ACP87] S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic and Discrete Methods*, 8:277–284, 1987.

[AP86] S. Arnborg and A. Proskurowski. Characterization and recognition of partial 3-trees. *SIAM Journal on Algebraic and Discrete Methods*, 7:305–314, 1986.

[BH98] H. L. Bodlaender and T. Hagerup. Parallel algorithms with optimal speedup for bounded treewidth. *SIAM Journal on Computing*, 27(6):1725–1746, 1998.

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

[Bod93] H. L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11:1–21, 1993.

[Bod96a] H. L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, 25(6):1305–1317, 1996.

[Bod96b] H. L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. Technical Report UU-CS-1996-02, Department of Computer Science, Utrecht University, 1996.

[Bod97] H. L. Bodlaender. Treewidth: Algorithmic techniques and results. In I. Prívara and P. Ružička, editors, *Proceedings of the 22nd International Symposium on the Mathematical Foundations of Computer Science (MFCS'97).* Springer Lecture Notes in Computer Science 1295, pages 29–36, 1997.

[Cha98] A. Charlesworth. Starfire: Extending the SMP envelope. *IEEE Micro*, pages 39–49, January/February 1998.

[Cou90] B. Courcelle. The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. volume 85 of *Information & Computation*. 1990.

[DF95] R. G. Downey and M. R. Fellows. Fixed-parameter tractability and completeness I: Basic results. *SIAM Journal on Computing*, 24(4):873–921, 1995.

[FL89] M. R. Fellows and M. A. Langston. On search, decision and the efficiency of polynomial-time algorithms (extended abstract). In *Proceedings of the 21st ACM Symposium on the Theory of Computation (STOC'89)*, pages 501–512, 1989.

[Hag98a] T. Hagerup. Bodlaender's algorithm explained. private communication, 1998.

[Hag98b] T. Hagerup. Comparing integer sequences. private communication, 1998.

[Him96] M. Himsolt. The graphlet system. In S. North, editor, *Graph Drawing 96*, volume 1190 of *Lecture Notes in Computer Science*, pages 233–240. Springer-Verlag, Heidelberg, 1996.

[Him97] M. Himsolt. GML: A portable graph file format. Technical report, Universität Passau, 1997. http://www.fmi.uni-passau.de/Graphlet/GML/gml-tr.html.

[Joy96] I. Joyner. A critique of C++. http://www.elj.com/cppcv3/, 1996.

[Kar72] R.M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, 1972.

[Klo94] T. Kloks. *Treewidth. Computations and Approximations*, volume 842 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, 1994.

[Lag90] J. Lagergren. Efficient parallel algorithms for tree-decomposition and related problems. In *Proceedings of the 31st Symposium on the Foundations of Computer Science (FOCS'90)*, pages 173–182, St. Louis, MS, 1990. IEEE Computer Society Press.

[Mar80] C. D. Marlin. *Coroutines. A Programming Methodology, a Language Design and an Implementation*, volume 95 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, 1980.

[Mey92] B. Meyer. *Eiffel: the language*. Prentice Hall object-oriented series. Prentice Hall, Englewood Cliffs, NJ, 1992.

[Mey97] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, NJ, second edition, 1997.

[Möh90] R.H. Möhring. Graph problems related to gate matrix layout and PLA folding. In G. Tinhofer, E. Mayr, H. Noltemeier, and M. M. Syslo, editors, *Computational Graph Theory*, volume 7 of *Computing Supplement*, pages 17–51. Springer-Verlag, Wien, 1990.

[MNSU98] K. Mehlhorn, S. Näher, M. Seel, and C. Uhrig. *The LEDA user manual: Version 3.6*. Max-Planck-Institut für Informatik, Saarbrücken, 1998.

[MT91] J. Matoušek and R. Thomas. Algorithms finding tree-decompositions of graphs. *Journal of Algorithms*, 12:1–22, 1991.

[NW78] A. Nijenhuis and H. S. Wilf. *Combinatorial Algorithms for Computers and Calculators*. Computer Science and Applied Mathematics, a Series of Monographs and Textbooks. Academic Press, New York, second edition, 1978.

[PH90] D. A. Patterson and J. L. Hennessy. *Computer architecture: a quantitative approach*. Morgan Kaufmann, San Mateo, 1990.

[Ree92] B. A. Reed. Finding approximate separators and computing tree width quickly. In N. Alon, editor, *Proceedings of the 24th ACM Symposium on the Theory of Computation (STOC'92)*, pages 221–228. ACM Press, 1992.

[Ros74] D. J. Rose. Triangulated graphs and the elimination process. *Discrete Mathematics*, 7:317–322, 1974.

[RS83] N. Robertson and P. D. Seymour. Graph minors. I. Excluding a forest. *Journal of Combinatorial Theory Series B*, 35:39–61, 1983.

[RS86] N. Robertson and P. D. Seymour. Graph minors. II. Algorithmic aspects of tree-width. *Journal of Algorithms*, 7:309–322, 1986.

[San96] D. P. Sanders. On linear recognition of tree-width at most four. *SIAM Journal on Discrete Mathematics*, 9(1):101–117, 1996.

[SE90] B. Stroustrup and M. A. Ellis. *The annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, 1990.

[Sei90] H. Seidl. Deciding equivalance of finite tree automata. *SIAM Journal of Computation*, 19(3):424–437, 1990.

[SKS97] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw Hill Series in Computer Science. McGraw Hill, New York, third edition, 1997.

[Tho97] M. Thorup. Structured programs have small tree-width and good register allocation. In R. H. Möhring, editor, *Graph-theoretic concepts in computer science (WG-97): 23rd international workshop, Berlin, Germany, 1997*, volume 1335 of *Lecture Notes in Computer Science*, pages 318–332. Springer-Verlag, Heidelberg, 1997.

[Tin90] G. Tinhofer. Generating graphs uniformly at random. In G. Tinhofer, E. Mayr, H. Noltemeier, and M. M. Syslo, editors, *Computational Graph Theory*, volume 7 of *Computing Supplement*, pages 235–255. Springer-Verlag, Wien, 1990.

[WE85] N. H. E. Weste and K. Eshraghian. *Principles of CMOS VLSI design: a systems perspective.* Addison-Wesley, Reading, MA, 1985.

[Wil81] H. S. Wilf. The uniform selection of free trees. *Journal of Algorithms*, 2:204–207, 1981.

Index

BANDWIDTH, 31
basis (in a k-tree), 17
bridge vertex, 87

characteristic, 21–22, 24
  final, 46
  preliminary, 36–37
characteristic, 101
COLORING, 26
completeness, 22, 25, 47
correctness, 22, 25

fixed-parameter tractability, 12, 72
full set of characteristics, 22, 23, 25, 47, 48, 79

graph, 14
  labeled, 91
  unlabeled, 91
GATEMATRIXLAYOUT, 30

HAMILTONIANCIRCUIT, 20

INDEPENDENTSET, 6–10, 21–22, 26
induction on a tree, 23
internal vertex, 87
isomorphic, 91

k-tree, 17

leaf vertex, 86

node, 15
  forget, 22
  introduce, 22
  join, 22
  start, 22

partial k-tree, 17
partial solution, 20
path decomposition, 29, 32
PATHWIDTH, 26, 29
pathwidth, 29
pipelining, 27, 57, 58, 102

reduced bag sequence, 34
rewriting tree decompositions, 18, 22, 26

separators, 80

(total) k-tree, 17
tree automaton, 23–28, 57, 73, 101, 106
tree decomposition, 15–16
  rooted, 16
tree node, 15
TREEWIDTH, 18, 88
treewidth, 16
trunk, 74
  degenerate, 75
$\mathcal{T}$-sequences, 42–43, 58

$\mathcal{U}$-sequences, 42
utilization sequence, 36, 42

vertex, 15
VLSI design, 29

W-separator, 83