Towards Faster Parallel Algorithms for Tree Decompositions

Taekseung Kim

CMU-CS-26-112

May 2026

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:

Guy Blelloch, Chair
William Kuszmaul

Submitted in partial fulfillment of the requirements
for the degree of
Master of Science in Computer Science.

Copyright © 2026 Taekseung Kim

Keywords: treewidth, graph algorithms, tree decomposition, parallel algorithms, dynamic algorithms, query processing, hypertreewidth

Abstract

The tree decomposition problem, also known as the treewidth problem, is a central topic in graph algorithms. Its importance comes from the fact that many graph problems that are NP-hard in general become tractable on graphs of bounded treewidth. This fixed-parameter viewpoint also appears in database theory, conjunctive query evaluation, and constraint satisfaction problems.

In this thesis, we survey major algorithmic developments for computing treewidth. We first define tree decompositions and explain why they support dynamic programming on graphs with tree-like structure. We then review separator-based algorithms, which recursively construct decompositions using small balanced separators. Next, we discuss the compression and improvement paradigm developed in the 1990s, especially in the work of Bodlaender and Kloks, and Bodlaender and Hagerup. This framework reduces the graph, solves the smaller instance, expands the decomposition, and improves it to the desired width, leading to Bodlaender's linear-time exact algorithm for fixed treewidth.

We then briefly discuss modern static and dynamic algorithms. Recent static algorithms often compute constant-factor approximate decompositions with better dependence on the parameter, such as $2^{O(k)}n$ or $2^{O(k)}n \log n$. Dynamic algorithms instead maintain a bounded-width tree decomposition under edge insertions and deletions.

A main perspective of this thesis is parallelism. Some older methods, such as balancing and improvement-based algorithms, have natural parallel components. In contrast, many modern algorithms rely on sequential local modifications, recursive dependencies, or rotation-based updates, making them difficult to parallelize. This gap motivates studying which parts of existing treewidth algorithms can be adapted to parallel static or batch-dynamic settings.

Contents

1 Introduction **8**
1.1 Outline 8

2 Database theory **9**
2.1 Preliminaries 9
2.2 Join Size, AGM Bound, and Structural Parameters 10

3 Width Parameters for Query Hypergraphs **13**
3.1 Treewidth 13
3.2 Hypertreewidth 16
3.3 Generalized Hypertreewidth 18
3.4 Fractional Hypertreewidth 19
3.5 Relationships Between the Parameters 20

4 Tree decomposition and algorithms **21**
4.1 Preliminaries 21
4.1.1 Why Tree Decomposition? 21
4.1.2 Definitions Recalled from Section 3.1 22
4.1.3 Simplicial and I-Simplicial Vertices 22
4.1.4 Parallel Model 22
4.1.5 Summary of Algorithms 23
4.1.6 Useful facts 24
4.2 Static Algorithms for Tree Decomposition 27
4.2.1 Separator-Based Algorithms 27
4.2.2 Compression and Improvement Paradigm 28
4.2.3 Bodlaender–Kloks Improvement Algorithm 31
4.2.4 Bodlaender–Hagerup Parallel Improvement 33
4.2.5 Bodlaender's Linear-Time Exact Algorithm 34
4.3 Modern Static Tree decomposition algorithms 36
4.3.1 Bodlaender et al. Static Approximation Algorithms . 37
4.3.2 Korhonen's Two-Approximation Algorithm 40
4.4 Dynamic tree decomposition 41

5 Future Work **43**

List of Figures

1 A graph together with a width-1 tree decomposition 15
2 A graph together with a width-2 tree decomposition 15
3 A non-example of a tree decomposition 16
4 Recursive separator-based construction of a tree decomposition 29
5 Deleting and reintroducing an I-simplicial vertex 30
6 Compression, expansion, and improvement in a tree-decomposition algorithm 31
7 A characteristic stores a compressed summary of a partial tree decomposition 33
8 Construction of the improved graph G^I 37
9 Separator recursion in the Bodlaender et al. approximation algorithm 38
10 Linear-time approximation idea using early stopping 39

1 Introduction

Database theory, beyond its practical implementation aspects, has a strong connection with graph theory. For a database, each relation can be viewed as a hyperedge, and conjunctive queries can be interpreted as operations over these hyperedges. By its relational definition, there is a deep correspondence between databases and hypergraphs.

In this context, several approaches have been developed to identify and exploit width parameters in order to characterize tractable queries and to expand the class of tractable queries. This perspective can be viewed as a form of fine-grained complexity analysis in the setting of query evaluation. One such parameter is hypertreewidth, which characterizes tractable conjunctive queries in relational systems. Multiple attempts have been made to improve query evaluation algorithms based on these width parameters, and a limited number of algorithms have been proposed to compute these parameters efficiently.

Hypertreewidth originates from the concept of treewidth, a graph parameter that measures how close a graph is to a tree. Several NP-hard problems, such as Maximum Independent Set, become tractable when a tree decomposition of bounded treewidth is given. The problem of computing tree decompositions was extensively studied approximately thirty years ago, notably by Bodlaender, with subsequent but relatively modest improvements in running time. More recent developments by Korhonen have addressed both static and dynamic settings. However, since Bodlaender's 1998 work, there has been no parallel algorithm for computing treewidth.

In this thesis, we present a survey of these graph width parameters in the context of database theory and graph algorithms. We also survey algorithms for computing tree decompositions in both static and dynamic settings. In addition, we will view this in a parallel algorithm perspective, whether each algorithm can be parallelized or not.

1.1 Outline

Section 2 covers the background of database theory, including conjunctive queries, query hypergraphs, and the AGM bound. Section 3 introduces width parameters for query hypergraphs and explains why treewidth appears naturally in this setting. In particular, Section 3 gives the formal definition of tree decomposition and treewidth, while Section 4 later uses these definitions from a graph-theoretic perspective. Section 4 surveys algorithms for computing tree decompositions, organized by their main idea, time complex-

ity, approximation guarantee, structural tools, and limitations. Section 4.2 covers classical static algorithms, Section 4.3 covers modern static approximation algorithms, and Section 4.4 covers dynamic tree-decomposition algorithms. The motivation for treewidth is introduced early through database query structure, and Section 4 explains the theoretical graph-algorithmic viewpoint in more detail. Section 5 discusses future directions toward parallel static and batch-dynamic tree-decomposition algorithms.

2 Database theory

2.1 Preliminaries

In this section, we give an overview of basic database terminology, define conjunctive queries, and explain how conjunctive queries can be represented as hypergraphs.

In database terminology, a relation can be viewed as a table. The columns of the table are called attributes, and the rows are called tuples. For example, a relation

$$R(A, B)$$

has two attributes, A and B. A tuple $(a, b) \in R$ means that the value a appears in column A and the value b appears in column B in one row of the table.

In the logical representation of a query, we usually write variables instead of concrete attribute names. For example, in the atom

$$R(x, y),$$

the variables x and y range over possible values in the database domain. The atom $R(x, y)$ is satisfied by an assignment of values to x and y if the resulting tuple belongs to the relation R. Thus, if $x = a$ and $y = b$, then $R(x, y)$ is true precisely when $(a, b) \in R$.

A conjunctive query is a database query built from a conjunction of relation atoms. It asks for all assignments of variables that simultaneously satisfy all of the listed relations. For example,

$$Q(x, y, z) \ :- \ R(x, y),\ S(y, z),\ T(x, z)$$

asks for all triples (x, y, z) such that $(x, y) \in R$, $(y, z) \in S$, and $(x, z) \in T$. Thus, evaluating a conjunctive query corresponds to computing a join of the relations appearing in the query:

$$R(x, y) \bowtie S(y, z) \bowtie T(x, z).$$

A conjunctive query can be mapped to a hypergraph in a natural way. Each variable is represented as a vertex, and each relation atom is represented as a hyperedge containing the variables that appear in that relation. For the query above, the corresponding hypergraph is

$$V = \{x, y, z\}, \qquad E = \big\{\{x, y\}, \{y, z\}, \{x, z\}\big\}.$$

In this example, every relation atom has two variables, so every hyperedge has size two. Therefore, this particular query hypergraph is an ordinary graph. This is not a contradiction: a graph is a special case of a hypergraph in which every hyperedge has size two. More general conjunctive queries naturally produce hypergraphs with larger hyperedges. For example, an atom

$$U(x, y, z)$$

would correspond to the hyperedge $\{x, y, z\}$, which has size three.

Thus, the query hypergraph records which variables appear together in the same relation atom. From this perspective, a hypergraph is the combinatorial object that captures the structure of a conjunctive query, while ordinary graphs arise as the special case of queries whose atoms all have arity two.

2.2 Join Size, AGM Bound, and Structural Parameters

The connection between conjunctive queries and hypergraphs is useful because the complexity of join processing is strongly influenced by the structure of the query hypergraph. In this section, we focus on full conjunctive queries, meaning that every variable appearing in the body of the query also appears in the output. This assumption lets us identify the output variables with the vertices of the query hypergraph.

Let $H = (V, E)$ be the hypergraph of a full conjunctive query. For each hyperedge $e \in E$, let R_e denote the relation atom whose variables form the edge e. For example, in the triangle query

$$Q(x, y, z) \ :-\ R(x, y),\ S(y, z),\ T(x, z),$$

the hyperedge $\{x, y\}$ corresponds to the relation $R(x, y)$, the hyperedge $\{y, z\}$ corresponds to $S(y, z)$, and the hyperedge $\{x, z\}$ corresponds to $T(x, z)$.

A fractional edge cover of H is a choice of nonnegative weights $(x_e)_{e \in E}$ such that every vertex is covered with total weight at least 1:

$$\sum_{e \ni v} x_e \geq 1 \qquad \text{for every } v \in V.$$

The fractional edge cover number of H is

$$\rho^*(H) = \min \sum_{e \in E} x_e,$$

where the minimum is over all fractional edge covers of H.

The quantity $\rho^*(H)$ is a covering parameter, not a matching parameter. A matching chooses disjoint edges, while an edge cover chooses edges that cover all vertices. A fractional edge cover relaxes this covering problem by allowing each edge to be chosen with a fractional weight. Thus, $\rho^*(H)$ measures the minimum total fractional amount of relation atoms needed to cover all variables of the query.

The AGM bound, named after Atserias, Grohe, and Marx, is a worst-case upper bound on the output size of a conjunctive query, expressed in terms of a fractional edge cover of its query hypergraph [1]. It states that for every fractional edge cover $(x_e)_{e \in E}$,

$$|Q(D)| \leq \prod_{e \in E} |R_e|^{x_e}.$$

Here $Q(D)$ is the output of the query on the database instance D, and R_e is the input relation corresponding to the hyperedge e.

In particular, if every input relation has size at most N, then

$$|Q(D)| \leq N^{\sum_{e \in E} x_e}.$$

Minimizing the exponent over all fractional edge covers gives

$$|Q(D)| \leq N^{\rho^*(H)}.$$

We now return to the triangle query

$$Q(x, y, z) \ :- \ R(x, y), \ S(y, z), \ T(x, z).$$

A naive approach may enumerate all triples (x, y, z) from the database domain, which leads to cubic time in the domain size. Another common way to evaluate joins is by a binary join plan. A binary join plan joins two inputs at a time, where each input is either an original relation or an intermediate join result. For example, one possible binary join plan for the triangle query is

$$(R(x, y) \bowtie S(y, z)) \bowtie T(x, z).$$

Even if each input relation has size at most N, the intermediate result $R(x, y) \bowtie S(y, z)$ can have size $O(N^2)$. Thus, a binary join plan may produce

intermediate results that are larger than the worst-case size of the final output. For more details on join plans and the AGM bound, see Atserias, Grohe, and Marx [1].

The AGM bound shows that the worst-case output size of the triangle query is only

$$O(N^{3/2}).$$

To see where this exponent comes from, consider the triangle hypergraph with edges

$$\{x, y\}, \qquad \{y, z\}, \qquad \{x, z\}.$$

Assigning weight 1/2 to each of the three edges gives

$$x_{\{x,y\}} = x_{\{y,z\}} = x_{\{x,z\}} = \frac{1}{2}.$$

This is a valid fractional edge cover because every variable appears in two edges. For example, the variable x appears in $\{x, y\}$ and $\{x, z\}$, so the total weight covering x is

$$\frac{1}{2} + \frac{1}{2} = 1.$$

The same holds for y and z. Therefore every vertex is covered with total weight at least 1, and the total weight of the cover is

$$\frac{1}{2} + \frac{1}{2} + \frac{1}{2} = \frac{3}{2}.$$

Moreover, this is optimal: the three cover constraints are

$$x_{\{x,y\}} + x_{\{x,z\}} \geq 1,$$

$$x_{\{x,y\}} + x_{\{y,z\}} \geq 1,$$

and

$$x_{\{x,z\}} + x_{\{y,z\}} \geq 1.$$

Adding these inequalities gives

$$2\left(x_{\{x,y\}} + x_{\{y,z\}} + x_{\{x,z\}}\right) \geq 3,$$

so every fractional edge cover has total weight at least 3/2. Hence

$$\rho^*(H) = \frac{3}{2}.$$

If each relation has size at most N, the AGM bound gives

$$|Q(D)| \leq O(N^{3/2}).$$

The reason we introduce fractional edge covers is that they give a query-structure-based bound on the worst-case output size. Instead of analyzing one particular join plan, we assign weights to relation atoms in a way that covers all variables. These weights become the exponents in the AGM bound. For the triangle query, this explains why the final output is bounded by $O(N^{3/2})$, even though a binary join plan may create an intermediate result of size $O(N^2)$.

More generally, fractional edge covers give a way to translate the combinatorial structure of a query hypergraph into a worst-case join-size bound. This motivates the study of structural parameters of query hypergraphs. The fractional edge cover number gives a bound on the size of one global join, but more complicated queries are often evaluated by decomposing the query hypergraph into simpler pieces. The efficiency of such decompositions is measured by width parameters. Treewidth, hypertreewidth, generalized hypertreewidth, and fractional hypertreewidth measure how tree-like the query hypergraph is, using either vertices, hyperedges, or fractional hyperedge covers. These parameters identify classes of conjunctive queries that can be evaluated efficiently because the query can be broken into locally manageable parts.

3 Width Parameters for Query Hypergraphs

In this section, we discuss several width parameters for query hypergraphs. The goal is to explain how treewidth, hypertreewidth, generalized hypertreewidth, and fractional hypertreewidth measure different forms of tree-like structure in conjunctive queries.

3.1 Treewidth

Treewidth is a width parameter for graphs. It measures how close a graph is to being a tree. The central object in its definition is a tree decomposition [18].. Intuitively, a tree decomposition represents a graph by arranging overlapping subsets of vertices, called bags, in a tree-like structure. The bags are required to preserve the vertices and edges of the original graph, while also ensuring that each vertex appears continuously along the decomposition tree.

Definition 3.1 (Tree decomposition)**.** *Let $G = (V, E)$ be a graph. A tree decomposition of G is a pair*

$$(T, \{B_t\}_{t \in V(T)}),$$

where T is a tree and each bag B_t is a subset of V, satisfying the following three conditions:

1. ***Vertex coverage:*** *every vertex of G appears in at least one bag. That is,*

$$\bigcup_{t \in V(T)} B_t = V.$$

2. ***Edge coverage:*** *for every edge $\{u, v\} \in E$, there exists a node $t \in V(T)$ such that both endpoints appear in the same bag:*

$$\{u, v\} \subseteq B_t.$$

3. ***Connectedness condition:*** *for every vertex $v \in V$, the set of nodes whose bags contain v,*

$$\{t \in V(T) : v \in B_t\},$$

induces a connected subtree of T.

Definition 3.2 (Width and treewidth)**.** *The width of a tree decomposition $(T, \{B_t\}_{t \in V(T)})$ is*

$$\max_{t \in V(T)} |B_t| - 1.$$

The treewidth of G, denoted $\mathrm{tw}(G)$, is the minimum width over all tree decompositions of G:

$$\mathrm{tw}(G) = \min_{(T, \{B_t\})} \left(\max_{t \in V(T)} |B_t| - 1 \right).$$

The subtraction by one is a convention chosen so that trees have treewidth 1, while edgeless graphs have treewidth 0. Thus, small treewidth means that the graph can be decomposed into small bags arranged in a tree.

In the following figures, we present examples of a graph G, a tree decomposition T, and the resulting treewidth. Figure 1 shows a width-1 tree decomposition of a tree. Figure 2 shows a width-2 tree decomposition. Figure 3 is not a valid tree decomposition, because the bags containing the vertex x do not form a connected subtree of T.

A width-1 tree decomposition

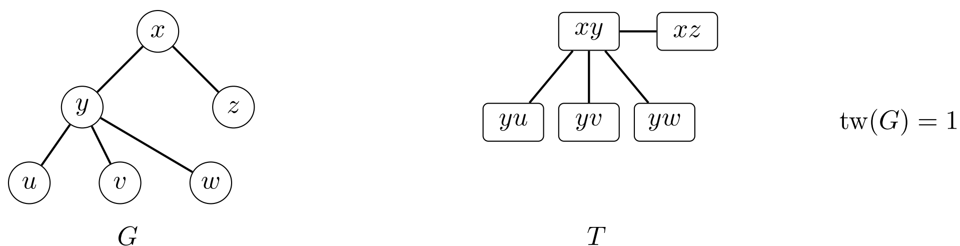


Figure 1: A graph together with a width-1 tree decomposition. Each bag has size two, so the width is $2-1=1$. The bags are arranged so that every edge of G appears in some bag and, for every vertex, the bags containing that vertex form a connected subtree of T.

A width-2 tree decomposition

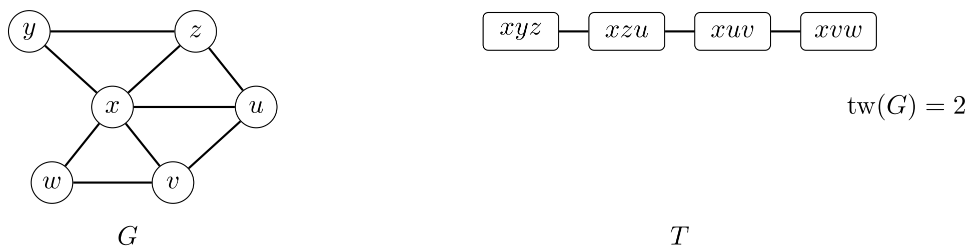


Figure 2: A graph together with a width-2 tree decomposition. Each bag has size three, so the width is $3-1=2$. The path structure of the decomposition keeps the bags containing each vertex connected; for example, the bags containing x form the whole path in T.

A non-example

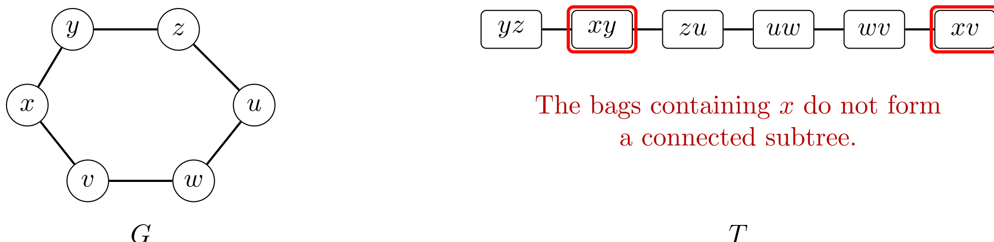


Figure 3: A non-example of a tree decomposition. Although each edge of G appears in some bag, the connectedness condition fails for the vertex x. The bags containing x are xy and xv, but the path between these two bags in T contains bags that do not contain x.

Relating this definition back to database joins, a conjunctive query gives a hypergraph rather than just a graph. To apply graph treewidth to a hypergraph $H = (V, E)$, we first convert H into its primal graph. The primal graph of H is the graph with the same vertex set V, where two distinct vertices $u, v \in V$ are adjacent if there exists a hyperedge $e \in E$ such that $u, v \in e$. Equivalently, each hyperedge of H is replaced by a clique on the vertices contained in that hyperedge. Thus, treewidth measures the graph-like structure induced by co-occurrence of variables in relation atoms [9].

Bounded treewidth of the primal graph implies that a conjunctive query can be evaluated by dynamic programming over a tree decomposition. However, primal treewidth can be too restrictive for hypergraphs. The reason is that a single large relation atom becomes a clique in the primal graph. Therefore, even if the original query hypergraph has a simple structure, its primal graph may have large treewidth. This motivates width parameters that work directly with hyperedges rather than first replacing each hyperedge by a clique. Hypertreewidth is one such parameter.

3.2 Hypertreewidth

Hypertreewidth is a width parameter designed directly for hypergraphs [10]. It measures whether the variables in each bag can be covered by a small number of hyperedges. In database terms, this means that the complexity of a bag is measured by how many relation atoms are needed to cover its variables, rather than by the number of variables alone.

A hypertree decomposition of a hypergraph $H = (V, E)$ consists of a tree T, a family of bags $(B_t)_{t \in V(T)}$ with $B_t \subseteq V$, and a family of edge covers $(\lambda_t)_{t \in V(T)}$ with $\lambda_t \subseteq E$. Intuitively, B_t is the set of variables handled at node t, and λ_t is the set of hyperedges, or relation atoms, used to cover those variables.

The decomposition must satisfy the following requirements:

1. **Vertex connectedness.** For every vertex $v \in V$, the set of nodes

$$\{t \in V(T) : v \in B_t\}$$

   forms a connected subtree of T.

2. **Hyperedge coverage.** For every hyperedge $e \in E$, there exists a node $t \in V(T)$ such that

$$e \subseteq B_t.$$

3. **Bag coverage by hyperedges.** For every node $t \in V(T)$, the bag B_t is covered by the hyperedges assigned to t:

$$B_t \subseteq \bigcup_{e \in \lambda_t} e.$$

In the original definition of hypertree decompositions, there is also an additional technical condition, often called the special condition [10, 11]. After rooting T, this condition requires that for every node t, the vertices from the subtree below t that are covered by the hyperedges in λ_t must already appear in the bag B_t. This condition is one of the restrictions that will be relaxed when we define generalized hypertreewidth below.

The width of the hypertree decomposition is

$$\max_{t \in V(T)} |\lambda_t|.$$

Thus, the width is not the maximum bag size. Instead, it is the maximum number of hyperedges needed to cover a bag. The hypertreewidth of H, denoted $\mathrm{hw}(H)$, is the minimum width over all hypertree decompositions of H.

This distinction is important for database queries. A bag may contain many variables, but if those variables are all contained in one or two relation atoms, then the bag can still be manageable from the perspective of join evaluation. For example, consider a query with one large relation atom

$$R(x_1, x_2, \ldots, x_r).$$

The corresponding hypergraph has one hyperedge

$$e = \{x_1, x_2, \dots, x_r\}.$$

Its primal graph is the clique K_r, whose treewidth is $r - 1$. However, as a hypergraph, this query has a hypertree decomposition with one bag

$$B_t = \{x_1, x_2, \dots, x_r\}$$

and one covering hyperedge

$$\lambda_t = \{e\}.$$

Therefore its hypertreewidth is 1. This example shows why hypertreewidth can be much smaller than the treewidth of the primal graph.

A usual tree decomposition does not automatically give a hypertree decomposition, because hypertree decompositions require each bag to be covered by a small number of original hyperedges. If we view an ordinary graph as a hypergraph whose hyperedges all have size two, then a bag with many vertices may require several graph edges to cover it. Thus, treewidth and hypertreewidth measure different quantities: treewidth counts vertices in a bag, while hypertreewidth counts hyperedges needed to cover a bag.

To summarize, hypertreewidth keeps the tree-decomposition requirements of connectedness and hyperedge coverage, but changes how the width is measured: the width is determined by the number of relation atoms needed to cover each bag.

3.3 Generalized Hypertreewidth

Generalized hypertreewidth is a relaxation of hypertreewidth [11]. It uses the same basic idea of covering each bag by hyperedges, but it drops the additional special condition required in the original definition of hypertreewidth. The connectedness and hyperedge coverage conditions remain.

A generalized hypertree decomposition of a hypergraph $H = (V, E)$ consists of a tree T, a family of bags $(B_t)_{t \in V(T)}$ with $B_t \subseteq V$, and a family of edge covers $(\lambda_t)_{t \in V(T)}$ with $\lambda_t \subseteq E$, satisfying the following conditions:

1. For every vertex $v \in V$, the set

   $$\{t \in V(T) : v \in B_t\}$$

   forms a connected subtree of T.

2. For every hyperedge $e \in E$, there exists a node $t \in V(T)$ such that

$$e \subseteq B_t.$$

3. For every node $t \in V(T)$, the bag B_t is covered by the hyperedges assigned to t:

$$B_t \subseteq \bigcup_{e \in \lambda_t} e.$$

The width is again

$$\max_{t \in V(T)} |\lambda_t|.$$

The generalized hypertreewidth of H, denoted $\mathrm{ghw}(H)$, is the minimum width over all generalized hypertree decompositions of H.

The difference from hypertreewidth is therefore not that the connectedness condition disappears. Rather, generalized hypertreewidth keeps the basic tree-decomposition conditions and the bag-covering condition, but removes the extra special condition from hypertree decompositions. Since generalized hypertreewidth relaxes hypertreewidth, it is always at most hypertreewidth:

$$\mathrm{ghw}(H) \leq \mathrm{hw}(H).$$

For query processing, bounded generalized hypertreewidth means that the query can be decomposed into bags whose variables are covered by a bounded number of relation atoms. This supports efficient dynamic programming over the decomposition. hypertreewidth, and fractional hypertreewidth.

3.4 Fractional Hypertreewidth

Fractional hypertreewidth further relaxes generalized hypertreewidth by allowing each bag to be covered fractionally rather than integrally. Instead of assigning a set of hyperedges to each bag, we assign nonnegative weights to hyperedges.

For a set of vertices $B \subseteq V$, a fractional edge cover of B is a function

$$\gamma : E \to \mathbb{R}_{\geq 0}$$

such that

$$\sum_{e \ni v} \gamma(e) \geq 1 \qquad \text{for every } v \in B.$$

The weight of this cover is

$$\sum_{e \in E} \gamma(e).$$

A fractional hypertree decomposition of a hypergraph $H = (V, E)$ is a tree decomposition $(T, \{B_t\}_{t \in V(T)})$ of the hypergraph together with a fractional edge cover γ_t of each bag B_t. The width of the decomposition is

$$\max_{t \in V(T)} \sum_{e \in E} \gamma_t(e).$$

The fractional hypertreewidth of H, denoted $\mathrm{fhw}(H)$, is the minimum width over all fractional hypertree decompositions.

Fractional hypertreewidth is closely related to the AGM bound. At each bag, the cost of processing the local join can be bounded using the fractional edge cover number of that bag. Thus, fractional hypertreewidth can be viewed as applying the AGM bound locally inside a tree-like decomposition. This makes it a powerful parameter for analyzing worst-case optimal join processing.

3.5 Relationships Between the Parameters

The parameters above form a hierarchy. Since fractional covers are more general than integral covers, and generalized hypertree decompositions relax hypertree decompositions, we have

$$\mathrm{fhw}(H) \leq \mathrm{ghw}(H) \leq \mathrm{hw}(H).$$

Treewidth is related through the primal graph of the hypergraph. If the primal graph has small treewidth, then the query is tractable by standard dynamic programming. Hypergraph width parameters can be much smaller than the treewidth of the primal graph, because they exploit the fact that many variables may come from a single relation atom. The large-relation example above shows this: a single hyperedge on r variables has hypertreewidth 1, while its primal graph is the clique K_r and has treewidth $r - 1$.

More precisely, one should not say that hypertreewidth is always simply "smaller than treewidth," because the parameters measure different objects and use different notions of width. Treewidth counts vertices in bags of the primal graph, while hypertreewidth counts hyperedges needed to cover bags of the hypergraph. A safe way to state the relationship is that small primal

treewidth implies tractability, but small hypertreewidth or fractional hypertreewidth can identify additional tractable queries whose primal treewidth is large.

In database-theoretic terms, treewidth measures whether the variable interaction graph is tree-like, hypertreewidth measures whether each bag can be covered by few relation atoms, and fractional hypertreewidth measures whether each bag has a small fractional edge cover. Therefore, these parameters provide increasingly flexible ways to identify tractable classes of conjunctive queries.

The algorithmic problem of computing these width parameters is important because the parameters are useful only when the corresponding decompositions can actually be found. In database theory, bounded-width results usually have the following form: if a query hypergraph admits a decomposition of small width, then the query can be evaluated efficiently by dynamic programming over that decomposition. Thus, the decomposition is not only a certificate of tractability, but also the object used by the evaluation algorithm itself. Consequently, efficient algorithms for computing width parameters directly affect whether these structural ideas can be used in practice.

Among these parameters, treewidth has played a particularly central role. Although treewidth is defined for graphs, it provides the foundation for many decomposition-based algorithms and also serves as a starting point for understanding more general hypergraph parameters. Computing treewidth is therefore a fundamental algorithmic problem: it determines whether a given graph has a tree-like structure and, if so, produces a tree decomposition that can be used for dynamic programming. In the following sections, we review several algorithms for computing treewidth, focusing on how they detect, construct, and improve tree decompositions.

4 Tree decomposition and algorithms

4.1 Preliminaries

4.1.1 Why Tree Decomposition?

Treewidth is important because many graph problems that are NP-hard on general graphs become tractable on graphs of bounded treewidth. Examples include Independent Set, Maximum Independent Set, Vertex Cover, Clique, q-Coloring for fixed q, Hamiltonian Path, Hamiltonian Cycle, and Connected Vertex Cover [7, 4]. The common reason is that a tree decomposition

breaks the graph into small overlapping bags arranged in a tree structure. Once the bag size is bounded by $k + 1$, dynamic programming over the decomposition tree often gives algorithms with running time of the form

$$2^{O(k)} \cdot n,$$

or more generally $f(k) \cdot n^{O(1)}$. Thus, treewidth provides a way to turn structural information about the graph into efficient algorithms for problems that are otherwise computationally intractable.

4.1.2 Definitions Recalled from Section 3.1

The formal definitions of tree decomposition, width, and treewidth were given in Section 3.1. We recall them here only briefly: a tree decomposition represents a graph by arranging overlapping vertex sets, called bags, in a tree so that vertices are covered, edges are covered, and the bags containing each fixed vertex form a connected subtree. The width is the maximum bag size minus one, and $\mathrm{tw}(G)$ is the minimum possible width over all tree decompositions of G.

4.1.3 Simplicial and I-Simplicial Vertices

Some compression algorithms use local graph reductions based on vertices whose neighborhoods already have a clique-like structure.

A vertex v is *simplicial* if its neighborhood $N(v)$ forms a clique. Equivalently, every two distinct neighbors of v are adjacent. If v is simplicial and $|N(v)| \leq k$, then a width-k decomposition of $G - v$ can be extended to a width-k decomposition of G by attaching a leaf bag $\{v\} \cup N(v)$ to a bag containing $N(v)$.

In Bodlaender's compression framework, the analogous notion is defined using the improved graph G^I. A vertex v is *I-simplicial* if it has degree at most k and its neighborhood forms a clique in the improved graph G^I. This is useful because G^I adds edges that are safe for width-k treewidth testing, so clique structure in G^I can be used to justify deleting and later reintroducing vertices during compression. We will further introduce the formal definition of improved graph and

4.1.4 Parallel Model

When discussing parallel algorithms, we use the PRAM model. PRAM stands for parallel random-access machine. In this model, many processors

operate in parallel while accessing a shared memory. The EREW PRAM variant is the exclusive-read exclusive-write model: no two processors may read from or write to the same memory location at the same time. Thus, a bound such as $O(\log n)$ time using $O(n)$ operations on an EREW PRAM means logarithmic parallel time and linear total work. This is the model used in the Bodlaender–Hagerup parallel treewidth algorithm [5]; for general background on PRAM algorithms, see [12].

4.1.5 Summary of Algorithms

The following table summarizes the main algorithms discussed in this section. The final column records the parallel perspective rather than a formal theorem for every row.

Algorithm / line of work	**Time bound / guarantee**	**Parallel perspective**
Reed separator-based algorithm	$O(f(k)n \log n)$; width bounded in terms of k	Recursive pieces are parallelizable in principle, but separator levels create an $O(\log n)$ dependency depth.
Bodlaender–Kloks improvement	$f(k, \ell) \cdot n$ given a width-ℓ decomposition	Can be parallelizable, mentioned in [5]
Bodlaender–Hagerup balancing/improvement	$O(\log n)$ time and $O(n)$ operations on EREW PRAM for fixed k	Explicitly parallel.
Bodlaender exact algorithm	$2^{O(k^3)}n$ exact algorithm for fixed k	Compression has parallel potential; improvement is the main bottleneck.
Bodlaender et al. approximation	$2^{O(k)}n \log n$ and later $2^{O(k)}n$ constant-factor approximation	Separator recursion exposes parallel subproblems, but the known algorithms are sequential.
Korhonen static approximation	$2^{O(k)}n$ and width at most $2k + 1$	Local improvements are adaptive and appear hard to parallelize directly.
Korhonen dynamic algorithm	$2^{O(k)}n \log n$ amortized update time; width at most $9k + 8$, considering we insert from empty graph to current graph.	Sequential dynamic update method; batch-dynamic parallelization would need conflict control, which is hard to do under current method

Table 1: Summary of tree-decomposition algorithms discussed in this thesis.

4.1.6 Useful facts

Lemma 4.1 (Path property)**.** *Let $(T, \{B_t\}_{t \in V(T)})$ be a tree decomposition of G. If a vertex $v \in V(G)$ appears in two bags B_s and B_t, then v appears in every bag on the unique path from s to t in T.*

Proof. By the connectedness condition, the set

$$T_v = \{r \in V(T) : v \in B_r\}$$

induces a connected subtree of T. Since $s, t \in T_v$, the unique path between s and t in the tree T must lie entirely inside T_v. Therefore, every bag on this path contains v. □

Lemma 4.2 (Bags separate subtrees)**.** *Let* $(T, \{B_t\}_{t \in V(T)})$ *be a tree decomposition of* G*, and let* $t \in V(T)$*. Removing* t *from* T *splits* T *into connected components* $C_1, \ldots, C_r$*. For each component* C_i*, define*

$$V_i = \left(\bigcup_{s \in C_i} B_s \right) \setminus B_t.$$

Then there is no edge in G *between a vertex of* V_i *and a vertex of* V_j *for* $i \neq j$*.*

Proof. Suppose, for contradiction, that there is an edge $\{u, v\} \in E(G)$ with $u \in V_i$ and $v \in V_j$ for $i \neq j$. By the edge coverage condition, there is a bag B_r containing both u and v.

Since $u \in V_i$, there is some bag in the component C_i containing u. Similarly, since $v \in V_j$, there is some bag in the component C_j containing v. The bag B_r cannot lie in both components. If B_r lies outside C_i, then the connectedness condition for u forces u to appear in B_t, because every path from C_i to outside C_i passes through t. Similarly, if B_r lies outside C_j, then v must appear in B_t.

In all cases, the existence of a bag containing both endpoints forces the edge to be represented through the separator bag B_t. Therefore, after removing the vertices of B_t, vertices belonging to different components of $T - t$ cannot be adjacent in G. □

This lemma explains why bags behave like separators. A bag B_t separates the parts of the graph represented by the different connected components of $T - t$. This separator viewpoint is one of the main reasons tree decompositions are useful for algorithms: dynamic programming can process different subtrees independently once the shared bag is fixed.

Lemma 4.3 (Trees have treewidth one)**.** *If* G *is a tree with at least one edge, then*

$$\mathrm{tw}(G) = 1.$$

Proof. For every edge $\{u, v\} \in E(G)$, create a bag $B_{uv} = \{u, v\}$. Since G is a tree, these edge-bags can be arranged in a tree structure following the adjacency structure of G. Every vertex appears in the bags corresponding

to its incident edges, and these bags form a connected subtree. Every edge is contained in its corresponding bag. Hence this is a tree decomposition of width

$$\max_{uv \in E(G)} |B_{uv}| - 1 = 2 - 1 = 1.$$

Therefore, $\operatorname{tw}(G) \leq 1$.

Since G has at least one edge, it cannot have treewidth 0, because a width-0 decomposition has bags of size at most one and therefore cannot cover any edge. Thus, $\operatorname{tw}(G) = 1$. Figure 1 shows this phenomenon visually. □

Lemma 4.4 (Cliques force large bags)**.** *If K_r is a clique on r vertices, then every tree decomposition of K_r has a bag containing all r vertices. In particular,*

$$\operatorname{tw}(K_r) = r - 1.$$

Proof. For each vertex v of the clique, let

$$T_v = \{t \in V(T) : v \in B_t\}.$$

By the connectedness condition, each T_v is a connected subtree of T. Since every pair of vertices u, v is adjacent in K_r, the edge coverage condition implies that there is a bag containing both u and v. Hence the subtrees T_u and T_v intersect for every pair u, v.

Subtrees of a tree satisfy the Helly property for trees: if a collection of subtrees of a tree pairwise intersect, then they all have a common intersection. This is a standard property of trees and is often used in basic arguments about tree decompositions. Therefore,

$$\bigcap_{v \in V(K_r)} T_v \neq \emptyset.$$

Thus, there exists a bag containing every vertex of K_r. This bag has size at least r, so every tree decomposition has width at least $r - 1$.

On the other hand, the single-bag decomposition with bag $V(K_r)$ has width $r - 1$. Therefore,

$$\operatorname{tw}(K_r) = r - 1.$$

□

Lemma 4.5 (Treewidth of grids)**.** *Let G be the $n \times n$ grid graph. Then*

$$\operatorname{tw}(G) = n.$$

We omit the proof of this lemma, but the $n \times n$ grid graph is a useful example of a graph that is much less dense than a clique but still has large treewidth. It serves as a canonical example showing that treewidth captures more than edge density; it also captures global grid-like structure. See, for example, standard graph-theory treatments of treewidth and grid minors [8].

Lemma 4.6 (Size of nice tree decompositions)**.** *Let G be an n-vertex graph, and suppose that G has a tree decomposition of width at most ℓ. Then, after converting the decomposition into a nice tree decomposition, we may assume that the decomposition tree has $O(\ell n)$ nodes. In particular, for fixed ℓ, the number of nodes in the nice tree decomposition is $O(n)$ [6].*

Nice tree decomposition is a normal form, which is discussed in [6]. This fact is useful because dynamic programs over nice tree decompositions usually process one node of the decomposition tree at a time. Therefore, when the width ℓ is fixed, a running time that is linear in the number of nodes of the nice tree decomposition is also linear in the number of vertices of the original graph.

4.2 Static Algorithms for Tree Decomposition

There have been several important lines of work on static algorithms for computing tree decompositions. The main algorithmic question is the following: given a graph G and an integer k, either determine that $\mathrm{tw}(G) > k$, or construct a tree decomposition whose width is bounded in terms of k. Early algorithms used balanced separators, while later linear-time algorithms introduced the compression and improvement paradigm. More recent algorithms improve the dependence on k or give better approximation guarantees.

Throughout this section, we summarize each line of work according to its main idea, time complexity, approximation guarantee, structural tools, and limitations.

4.2.1 Separator-Based Algorithms

A natural way to construct a tree decomposition is to recursively find small balanced separators. A separator is a set of vertices $S \subseteq V(G)$ whose removal breaks the graph into two or more disconnected components. A separator is called balanced if every connected component of $G \setminus S$ has size at most a fixed constant fraction of $|V(G)|$. It is called small, in this context, if its size is bounded by a function of the target treewidth k, typically $O(k)$.

The key structural fact is that if $\text{tw}(G) \leq k$, then G has balanced separators of size $O(k)$; in particular, such separators can be obtained from bags of a width-k tree decomposition. Reed's separator-based algorithm [17] uses this principle algorithmically.

The algorithm is given a graph G and an integer k. It either concludes that $\text{tw}(G) > k$, or finds a balanced separator S whose size is bounded in terms of k. After finding such a separator, the algorithm recursively decomposes each connected component of $G \setminus S$ and then combines the recursive decompositions by adding a bag containing S.

This merging step should be interpreted carefully. If C is a component of $G \setminus S$, then the recursive call is applied to the subgraph induced by $C \cup S$, so that vertices of S remain available as an interface between the component and the rest of the graph. The resulting decomposition for $G[C \cup S]$ contains the separator vertices in appropriate bags, and these local decompositions can then be attached to a central bag containing S. Thus, the separator is not merely drawn on top of the recursive decompositions; it is included as the interface through which the recursive pieces are connected.

The key idea is therefore simple: a bag of the tree decomposition acts as a separator, and once the separator is fixed, the remaining components can be processed independently. If each recursive level processes $O(n)$ total vertices and the separator is balanced, then the recursion has $O(\log n)$ levels. This gives an $O(f(k) n \log n)$-type time bound, where the dependence on k comes from the separator-finding subroutine. Figure 4 shows the high-level recursive construction.

The strength of this approach is that it gives a clean structural explanation of why tree decompositions exist: small treewidth implies the existence of small balanced separators, and recursive separators can be assembled into a tree decomposition. However, the main limitation is the extra logarithmic factor in n. Because the recursion has logarithmic depth, vertices may be processed again in several recursive levels along a root-to-leaf path. Thus, even if the total work per level is linear, the total work becomes $O(n \log n)$, up to the dependence on k. This motivates later algorithms that try to reduce the graph size by a constant fraction in each round while spending only linear work per round.

4.2.2 Compression and Improvement Paradigm

Many later tree-decomposition algorithms follow a compression/improvement paradigm. This paradigm separates the algorithm into two conceptual steps: *compression*, which reduces the graph size, and *improvement*, which turns a

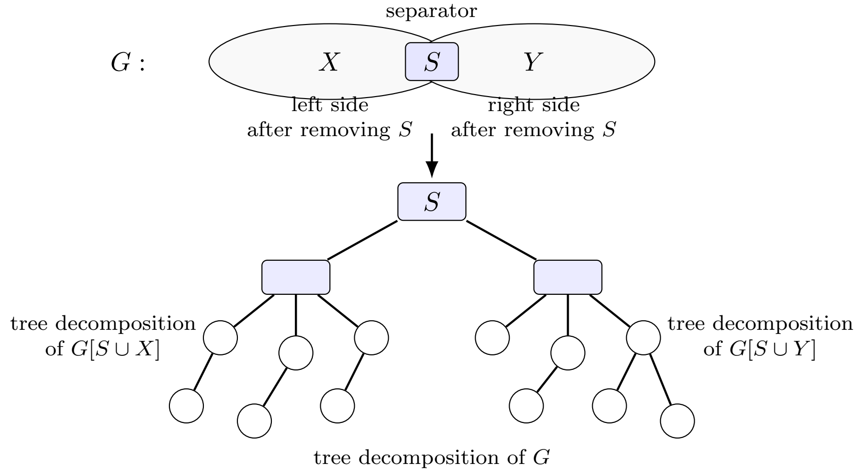


Figure 4: Recursive separator-based construction of a tree decomposition. The set S is a balanced separator of G, and X and Y denote the pieces left after removing S. The algorithm recursively constructs tree decompositions of subgraphs that include the separator as an interface, such as $G[S \cup X]$ and $G[S \cup Y]$, and then attaches these decompositions to a central bag containing S.

rough decomposition into a better one. Figure 6 shows the high-level flow.

The *compression* step reduces the size of the graph while preserving enough information to recover a decomposition of the original graph. More precisely, given a graph G, the algorithm constructs a smaller graph G' such that a tree decomposition of G' can be expanded back into a tree decomposition of G, although the expanded decomposition may have larger width.

Typical compression operations include contracting a large matching, deleting vertices whose neighborhoods are already clique-like in an auxiliary graph, or applying a reduction that removes a constant fraction of vertices while preserving enough information to reconstruct a decomposition of the original graph. The goal is not necessarily to produce an optimal decomposition immediately, but to make the instance smaller while keeping the treewidth under control. Figure 5 gives one example of a deletion-based compression step.

The *expansion* step reverses the compression. After recursively computing a decomposition of the smaller graph G', the algorithm lifts it back to a decomposition of the original graph G. Usually this is done by adding the

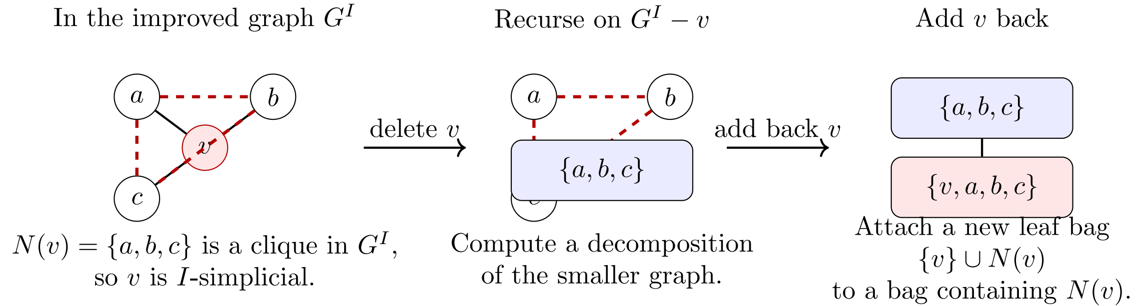


Figure 5: Deleting and reintroducing an I-simplicial vertex. An I-simplicial vertex is a vertex v of degree at most k whose neighborhood $N(v)$ forms a clique in the improved graph G^I. The dashed red edges are edges present in G^I, not necessarily in the original graph G. Since $N(v)$ is a clique in G^I, any width-k decomposition of $G^I - v$ contains a bag with all vertices of $N(v)$. Since $|N(v)| \leq k$, we can add v back by attaching a leaf bag $\{v\} \cup N(v)$ of size at most $k + 1$. Progress is made because the recursive instance has one fewer vertex.

compressed vertices back into the decomposition. Let the lifted decomposition have width ℓ, where typically $\ell = O(k)$.

The *improvement* step then takes this width-ℓ decomposition and either improves it to width at most k or determines that $\mathrm{tw}(G) > k$. Thus, compression gives a smaller instance and a rough decomposition, while improvement turns the rough decomposition into an exact width-k decomposition.

This paradigm is powerful because each compression round can be designed to remove a constant fraction of vertices. In bounded-treewidth graphs, the number of relevant edges is also controlled by a function of k times the number of vertices, so linear work in the current graph can be interpreted as linear in the current instance size. If both compression and improvement can be implemented with work proportional to the current number of vertices and edges, then the total work over all rounds becomes linear in n, up to the parameter dependence on k.

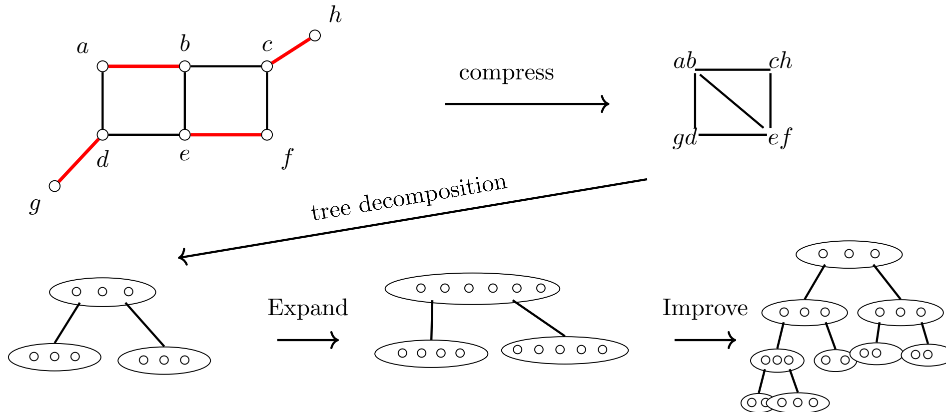


Figure 6: Compression, expansion, and improvement in a tree-decomposition algorithm. The graph on the top left is compressed by contracting the highlighted red edges, producing the smaller graph on the top right. A tree decomposition of the compressed graph is then expanded back to the original graph, which may increase the bag sizes. Finally, the improvement step reduces the expanded rough decomposition to the desired width, or determines that no width-k decomposition exists.

4.2.3 Bodlaender–Kloks Improvement Algorithm

The Bodlaender–Kloks algorithm [6] provides one of the central improvement subroutines for treewidth. Its input is a graph G, an integer k, and a nice tree decomposition of G of width ℓ. A nice tree decomposition is a special rooted tree decomposition whose nodes have a restricted set of simple types. This normal form is useful because it lets a dynamic program process the decomposition one local operation at a time. For more details, see [6]

The algorithm decides whether $\mathrm{tw}(G) \le k$ and, if so, constructs a tree decomposition of width at most k. For fixed k and ℓ, the running time is linear in the number of nodes of the given nice tree decomposition, with a parameter dependence on k and ℓ.

Lemma 4.7 (Bodlaender–Kloks improvement)**.** *Given a graph $G = (V, E)$, an integer k, and a nice tree decomposition of G of width ℓ, there is an algorithm that decides whether* $\mathrm{tw}(G) \le k$*. If* $\mathrm{tw}(G) \le k$*, the algorithm constructs a tree decomposition of G of width at most k. For fixed k and ℓ, the running time is*

$$f(k, \ell) \cdot |V(T)|,$$

where T is the input nice decomposition tree and $f(k, \ell)$ is a function depending only on k and ℓ.

We now describe how Bodlaender and Kloks achieve this improvement. The algorithm is a dynamic program over the given nice tree decomposition. The difficulty is that a partial solution is itself a partial tree decomposition of some part of the graph, so storing it explicitly would create too many states. Different partial decompositions may be large and complicated, even when their interaction with the rest of the graph is essentially the same.

Bodlaender and Kloks avoid storing the entire partial decomposition. Instead, for each node of the input decomposition, they store a finite summary of how a partial width-k decomposition interacts with the boundary vertices of that node. This summary is called a characteristic.

At a high level, a characteristic records the essential shape of the partial decomposition after restricting it to the boundary. In the presentation terminology, this is done by restricting the partial decomposition to the boundary set, removing inessential leaves, suppressing degree-two paths, and storing the resulting trunk together with additional interval/list data. Thus, the dynamic program stores only compressed information about the interaction between the partial solution and the boundary, rather than the entire partial tree decomposition.

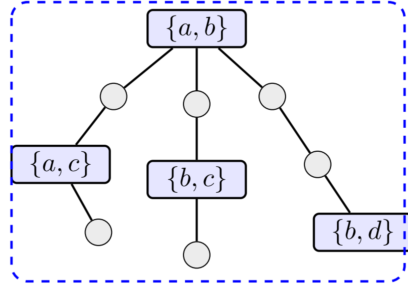
The key idea is therefore a finite-state dynamic program over a given tree decomposition. At each node, the algorithm combines possible characteristics from the children and keeps exactly those summaries that can arise from a valid partial decomposition of width at most k. If a compatible characteristic survives at the root, the algorithm reconstructs an exact width-k tree decomposition. Figure 7 illustrates how the characteristic compresses the information stored by the dynamic program.

In Bodlaender's linear-time exact algorithm, described below, this improvement step is used after a compression and expansion step. Thus, the section first describes the improvement subroutine, and then explains how Bodlaender's algorithm supplies a rough decomposition to which this subroutine can be applied. The improvement step is the source of the large parameter dependence, commonly written as

$$2^{O(k^3)} \cdot n.$$

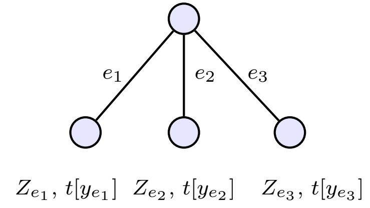
Its limitation is therefore not the dependence on n, which is linear, but the large dependence on k and the complexity of the dynamic-programming state space.

Restricted partial decomposition at node i



Restrict to the boundary X_i, remove inessential leaves, and suppress degree-two paths

Characteristic trunk



For each trunk edge e, store an interval model Z_e and a typical list $t[y_e]$

Figure 7: A characteristic stores a compressed summary of a partial tree decomposition at a node i of the input nice decomposition. The set X_i is the boundary through which the already-processed part of the graph interacts with the rest of the graph. The left side shows a restricted partial decomposition. The characteristic keeps only the essential trunk: inessential leaves are removed and degree-two paths are suppressed. For each trunk edge e, additional data such as the interval model Z_e and the typical list $t[y_e]$ records how hidden portions of the partial decomposition may attach along that edge.

4.2.4 Bodlaender–Hagerup Parallel Improvement

Bodlaender and Hagerup [5] study tree decomposition from a parallel-algorithmic perspective. Their contribution is important because many dynamic programs on tree decompositions depend not only on the width of the decomposition, but also on the depth of the decomposition tree. A tree decomposition of small width may still have large depth, which creates long dependencies in a parallel computation.

We use the EREW PRAM model defined in the preliminaries of this section.

The key idea is to balance the decomposition tree. First, Bodlaender and Hagerup show that a binary tree can be transformed into a rooted binary tree decomposition of logarithmic depth and constant width. Intuitively, this is done by recursively choosing balanced splitting points in the tree so that the resulting decomposition tree has logarithmic height. They then lift this balancing idea from trees to tree decompositions: given a tree decomposition of a graph G of width k, one can compute a rooted binary tree decomposition

of G of depth $O(\log n)$ and width at most $3k + 2$.

Lemma 4.8 (Balancing a tree)**.** *Given an n-node rooted binary tree T, one can compute a rooted binary tree decomposition of T of depth $O(\log n)$ and width at most 2 in $O(\log n)$ time, using $O(n)$ operations and $O(n)$ space on an EREW PRAM.*

Lemma 4.9 (Balancing a tree decomposition)**.** *For every fixed constant $k \geq 1$, given a tree decomposition with n nodes and width k of a graph G, one can compute a rooted binary tree decomposition of G of depth $O(\log n)$ and width at most $3k + 2$ in $O(\log n)$ time, using $O(n)$ operations and $O(n)$ space on an EREW PRAM.*

Combining this balancing step with an improvement algorithm yields a parallel width-minimization theorem. In this subsection, the input is already a tree decomposition of bounded width; the focus is therefore on balancing and parallel improvement rather than on the compression step used to obtain the initial decomposition in Bodlaender's exact algorithm. Compression is still important for a full parallel exact algorithm. In principle, some compression operations have parallel potential: a large matching can be contracted in parallel, and many deletion rules can be applied simultaneously when their affected neighborhoods are disjoint or otherwise nonconflicting. The challenge is to coordinate these local reductions with the later improvement step, whose state space is much more global.

Theorem 4.1 (Width minimization)**.** *For every fixed constant $k \geq 1$, given a tree decomposition with n nodes and width k of a graph G, one can construct a minimum-width tree decomposition of G in $O(\log n)$ time, using $O(n)$ operations and $O(n)$ space on an EREW PRAM.*

The takeaway is that tree decompositions can be made shallow without increasing the width by more than a constant factor. This is crucial for parallel dynamic programming: after balancing, the dependency depth becomes $O(\log n)$.

4.2.5 Bodlaender's Linear-Time Exact Algorithm

Bodlaender's linear-time algorithm [3] combines the compression paradigm with the Bodlaender–Kloks improvement step to compute exact treewidth in linear time for every fixed k. Although this algorithm historically predates the Bodlaender–Hagerup parallel result, we discuss it after the improvement subroutine because its structure is easiest to explain once compression, expansion, and improvement have been introduced.

The high-level structure is recursive: reduce the graph to a smaller graph, solve the smaller instance, expand the decomposition back to the original graph, and then improve the resulting decomposition to width k.

The main compression tool is based on the improved graph. Given a graph G and an integer k, the improved graph G^I adds an edge between two non-adjacent vertices if they have at least $k+1$ common neighbors of degree at most k in G. This operation is safe for treewidth testing because, in any width-k tree decomposition, such vertices must appear together in some bag, and therefore can be treated as adjacent. The improved graph is then used to identify reducible structure.

In Figure 5, the red edges are not meant to be produced by the specific pairwise common-neighbor rule shown in Figure 8. Rather, that figure illustrates the situation after the improved graph has already been constructed: the vertex v is I-simplicial because its neighborhood is a clique in G^I. Figure 8 separately shows the rule for constructing G^I. Note that G^I is the improved graph of G.

There are two main types of reductions. First, if the graph contains a large matching among suitable low-degree vertices, the algorithm contracts this matching to obtain a smaller graph. After recursively decomposing the smaller graph, the contracted edges can be expanded. Second, if many vertices are simplicial in the improved graph, the algorithm deletes many of them. After recursively decomposing the smaller graph, each deleted vertex can be added back by attaching a leaf bag containing the vertex and its neighborhood.

The central compression statement can be written as follows.

Theorem 4.2 (Bodlaender's compression technique [3])**.** *There is an algorithm that, given an n-vertex graph G and an integer k, runs in time $k^{O(1)}n$ and returns one of the following outcomes:*

1. *determines that the treewidth of G is larger than k;*
2. *returns a matching in G with at least $n/O(k^6)$ edges;*
3. *returns a graph G' with at most $n - n/O(k^6)$ vertices such that, if the treewidth of G is at most k, then the treewidth of G' is at most the treewidth of G. Furthermore, any tree decomposition of G' of width at most k can be transformed into a tree decomposition of G of width at most k in time $k^{O(1)}n$.*

The full algorithm, meaning compression plus recursive solution, expan-

sion, and Bodlaender–Kloks improvement, has running time

$$2^{O(k^3)} \cdot n.$$

It is exact: it either determines that $\mathrm{tw}(G) > k$, or constructs a tree decomposition of width at most k. The linear dependence on n comes from the fact that each compression round removes a constant fraction of the vertices, so the total size of all recursive instances is linear. The $2^{O(k^3)}$ factor comes from the improvement algorithm.

From a parallel-algorithmic perspective, the compression part of this framework is the more promising component. Matching contractions, deletion of many reducible vertices, and other local reductions can potentially be applied to many disjoint or nonconflicting parts of the graph at the same time. Thus, one possible route toward a parallel static treewidth algorithm is to parallelize the compression rounds first, while using balancing ideas to keep the resulting decomposition shallow. The harder remaining issue is the improvement step: the Bodlaender–Kloks dynamic program has a global state-space structure over the input decomposition, and it is less clear how to expose enough independent subproblems for efficient parallelism.

The main limitation is practical rather than purely asymptotic. The algorithm has a large hidden constant and a large dependence on k. Both of the compression step and improvement step has large constant factors along the algorithm, which makes these algorithms impractical.

4.3 Modern Static Tree decomposition algorithms

After Bodlaender's linear-time exact algorithm, there was a long period in which the exact fixed-parameter benchmark

$$2^{O(k^3)} \cdot n$$

remained the standard result for computing exact treewidth with linear dependence on the input size [3]. This does not mean that there were no approximation algorithms during this period. Rather, the main point is that the older exact framework still had a large dependence on the parameter k, and later work revisited the problem from the perspective of constant-factor approximation.

Modern static algorithms use constant-factor approximation algorithms for treewidth. Instead of insisting on a width-k decomposition whenever $\mathrm{tw}(G) \le k$, these algorithms return a decomposition whose width is $O(k)$, or else determine that the treewidth is larger than k. This relaxation allows

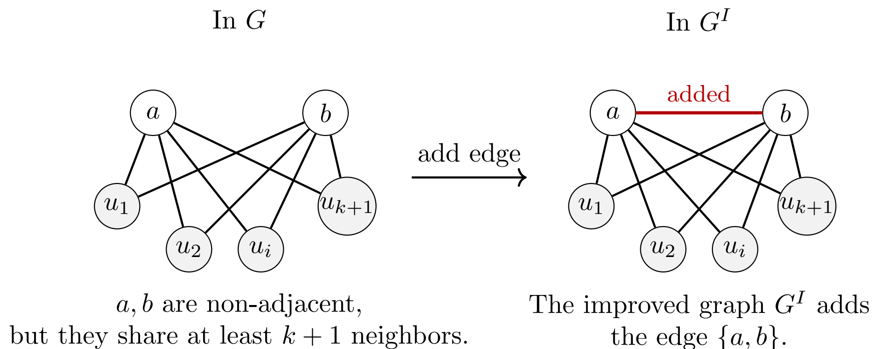


Figure 8: Construction of the improved graph G^I. If two non-adjacent vertices a and b have at least $k+1$ common neighbors of degree at most k in G, then the edge $\{a, b\}$ is added in G^I. This is safe for testing whether $\mathrm{tw}(G) \leq k$: if G has treewidth at most k, then the added edge is forced in every width-k tree-decomposition sense, so $\mathrm{tw}(G) \leq k$ if and only if $\mathrm{tw}(G^I) \leq k$.

significantly better dependence on k. In particular, the work of Bodlaender et al. gives single-exponential approximation algorithms such as $2^{O(k)} n \log n$ and later $2^{O(k)} n$ [2]. Korhonen later improved the approximation guarantee to a factor of 2, also with single-exponential dependence on k [13].

4.3.1 Bodlaender et al. Static Approximation Algorithms

Bodlaender et al. improve the approximation side of the treewidth problem [2]. Instead of computing an exact width-k decomposition with the large $2^{O(k^3)}$ dependence, their algorithms compute constant-factor approximate decompositions with better dependence on k.

Their first main algorithm gives a constant-factor approximation in time

$$2^{O(k)} n \log n.$$

At a high level, the algorithm again relies on balanced separators, but it uses them with a more careful recursive invariant than the earlier separator-based algorithms. Given a current boundary set S, the algorithm considers the components of $G \setminus S$. For an active component C, it works inside the subgraph $G[S \cup C]$, finds a balanced separator X, and then recurses on the pieces with new boundary sets.

This description looks similar to Reed's separator recursion, but the algorithmic improvement is in how the recursion is controlled. The algo-

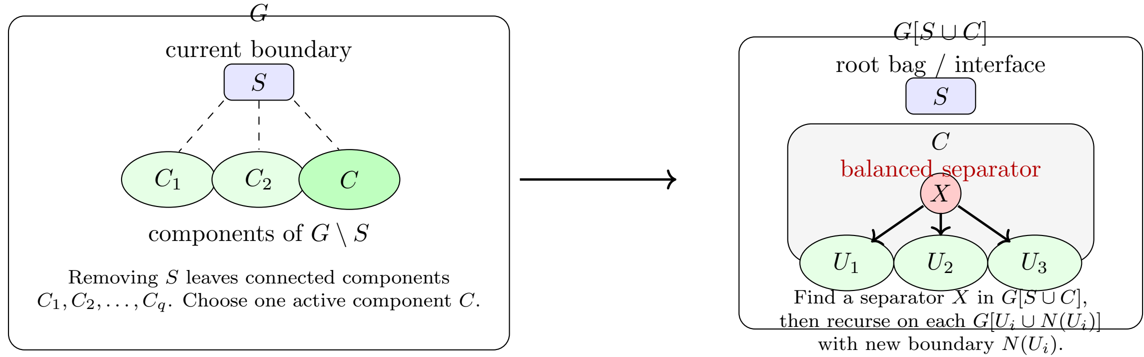


Figure 9: Separator recursion in the Bodlaender et al. approximation algorithm. The set S is the current boundary, and $C_1, C_2, \ldots, C_q$ are the connected components of $G \setminus S$. The algorithm selects an active component C and works inside the subgraph $G[S \cup C]$. It finds a balanced separator X inside this subproblem, which splits C into pieces U_1, U_2, U_3. The recursive calls are made on subgraphs of the form $G[U_i \cup N(U_i)]$, where $N(U_i)$ becomes the new boundary. The purpose of the boundary is to keep track of the interface through which the recursive piece attaches to the rest of the decomposition.

rithm keeps the boundary sets small, chooses separators in a way compatible with these boundaries, and uses separator-finding subroutines with single-exponential dependence on k. Thus, the high-level recursion still looks like separator recursion, but the parameter dependence is improved: the approximation setting allows the algorithm to search for separators and assemble the decomposition without paying the larger dependence associated with exact improvement. Figure 9 illustrates this recursive step.

Their second algorithm improves the dependence on n by obtaining a linear-time constant-factor approximation. Instead of continuing the separator recursion all the way down to individual vertices, the algorithm stops once all remaining components are small, for example of logarithmic size, and then handles these small components using an older exact or approximation routine.

The reason this early stopping helps is that separator recursion can charge a vertex on many levels of the recursion tree. By stopping once the remaining pieces are small, the algorithm avoids repeatedly running the

Stopping on small components

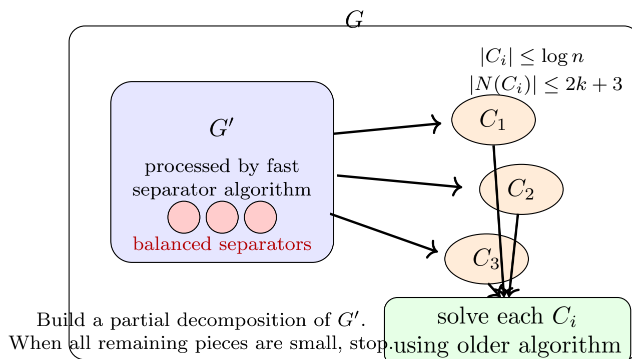


Figure 10: Linear-time approximation idea using early stopping. The separator recursion is used to process a large part G' of the graph. Once every remaining component C_i is small, the algorithm stops recursing on those components. Each small component has a controlled boundary $N(C_i)$, so it can be solved separately using an older algorithm. This avoids charging the separator routine on many lower-level subproblems and supports the final linear-time bound.

separator routine on many tiny subproblems. The large part of the graph is processed by the fast separator recursion, and the leftover pieces are small enough that the older routine can be used locally without increasing the total running time beyond linear, up to the single-exponential dependence on k. The result is a linear-time approximation algorithm, with running time of the form

$$2^{O(k)}n.$$

The main algorithmic contribution is not merely the observation that applications can tolerate approximate decompositions. Rather, the contribution is that constant-factor approximate tree decompositions can be computed with single-exponential dependence on k, and in the second algorithm with linear dependence on n. The limitation is that these algorithms are not exact: they do not necessarily return a width-k decomposition even when one exists. They trade exactness for better parameter dependence and a more efficient approximation framework.

4.3.2 Korhonen's Two-Approximation Algorithm

Korhonen's static algorithm [13] further improves the approximation guarantee. It achieves a 2-approximation in time

$$2^{O(k)} \cdot n.$$

More precisely, given a graph G and an integer k, the algorithm either determines that $\mathrm{tw}(G) > k$, or outputs a tree decomposition of width at most $2k + 1$.

At a high level, the algorithm maintains a tree decomposition and repeatedly looks for a large bag that can be locally improved. Here "improved" means that the bag can be replaced by a small tree-decomposition structure whose bags are smaller, while preserving the tree-decomposition conditions. This use of the word improvement is different from the Bodlaender–Kloks improvement subroutine and also different from the improved graph used in Bodlaender's compression step.

The main operation is to split a large bag using a suitable separation and then edit the current decomposition around that bag. Concretely, the algorithm replaces the large bag by several smaller bags arranged in a local tree structure, and reattaches the neighboring parts of the old decomposition to this new local structure. The edit must preserve both edge coverage and the connectedness condition for every vertex. If such local improvements can be performed, the width of the decomposition is reduced. If no such improvement is possible, the current decomposition certifies the desired approximation guarantee.

This gives a strong static approximation result for treewidth: it combines a 2-approximation ratio with a single-exponential dependence on k. Compared with exact algorithms, it avoids the larger $2^{O(k^3)}$ dependence.

The main limitation, especially from the perspective of parallel algorithms, is that the improvement process is highly sequential. The algorithm searches for a local change to the current decomposition, performs the edit, and then continues from the modified decomposition. Each edit changes the structure on which the next search is performed. These dependencies make it difficult to split the computation into many independent subproblems. This is one reason why older compression/improvement or separator-based ideas remain interesting when looking for parallel static tree-decomposition algorithms.

4.4 Dynamic tree decomposition

Static tree-decomposition algorithms compute a decomposition for a fixed graph. In the dynamic setting, the graph changes over time through edge insertions and deletions, and the goal is to maintain a bounded-width tree decomposition after each update. This is a *fully dynamic* setting, because both insertions and deletions are allowed. The update sequence is viewed as a sequence of operations applied to the graph, and the data structure must update its maintained tree decomposition after each operation. For the purposes of this survey, the most important formal condition is the promise that the treewidth of the graph remains at most k throughout the update sequence. More detailed adversarial assumptions, such as whether the update sequence is chosen by an oblivious adversary, are mainly relevant for randomized dynamic algorithms; they are not central to the high-level description here.

Recomputing a decomposition from scratch after every update would be too expensive. For example, even a linear-time static algorithm would require time proportional to the graph size after each update. The dynamic problem therefore asks whether the decomposition can be repaired locally after each edge insertion or deletion.

Korhonen's dynamic treewidth algorithm [14] gives a logarithmic-time amortized update bound under the promise that the treewidth of the graph remains at most k throughout the update sequence. More precisely, the algorithm maintains a tree decomposition of width at most $9k + 8$ under edge insertions and deletions, with amortized update time

$$2^{O(k)} \log n.$$

Thus, for fixed k, the update time is logarithmic in the number of vertices.

Theorem 4.3 (Korhonen, dynamic treewidth in logarithmic time [14])**.** *Let G be a dynamic graph updated by edge insertions and deletions, and suppose that $\mathrm{tw}(G) \leq k$ throughout the update sequence. There is a data structure that maintains a tree decomposition of G of width at most $9k + 8$ with amortized update time*

$$2^{O(k)} \log n.$$

The key idea is to maintain a tree decomposition with an additional *downwards well-linked* invariant. Informally, a well-linked condition says that a region of the graph has enough internal connectivity that it cannot be easily separated by a small cut. The word "downwards" refers to how

this connectivity condition is imposed relative to a rooted tree decomposition: the invariant controls how vertices in lower parts of the decomposition connect through the bags above them. This extra structure makes the decomposition more robust under updates, because it gives the algorithm a controlled way to locate and repair the part of the decomposition affected by an inserted or deleted edge.

The repair operation uses local rotations of the decomposition tree. These rotations are similar in spirit to rotations in splay trees. After an edge insertion or deletion, the affected part of the decomposition is moved toward a controlled location, repaired, and then the downwards well-linked invariant is restored. The amortized analysis shows that even though a single update may trigger several rotations, the average cost over the update sequence is only

$$2^{O(k)} \log n.$$

This algorithm is approximate rather than exact. Even though the graph is promised to have treewidth at most k, the maintained decomposition has width at most $9k + 8$. For fixed-parameter applications, this is still useful because the width remains bounded by a function of k.

The paper also shows that certain dynamic programming schemes over the maintained tree decomposition can be updated dynamically. This gives dynamic versions of applications usually obtained from bounded-treewidth dynamic programming. One can view these as Courcelle-type applications: in the static setting, Courcelle's theorem says that many graph properties expressible in monadic second-order logic can be solved efficiently on graphs of bounded treewidth. Here, the point is not to use the full theorem in detail, but to emphasize that bounded-width decompositions support many dynamic programming applications beyond computing the decomposition itself.

The main contribution of this work is that it reduces the dependence on n in dynamic treewidth maintenance to logarithmic amortized update time. This should be compared with recomputing a tree decomposition from scratch after each update, which would require at least linear dependence on the current graph size even if the static algorithm is linear for fixed k.

The main limitation, especially from a parallel-algorithmic perspective, is that the rotation-based repair process is adaptive. Each rotation changes the shape of the decomposition tree and affects which rotation should be applied next. This creates sequential dependencies between the repair steps. In a batch-dynamic setting, where many edge updates arrive at once, the difficulty is even more pronounced: different updates may try to rotate or

repair overlapping parts of the same decomposition tree, leading to conflicts. Thus, although the algorithm is very efficient dynamically, it does not immediately suggest a parallel batch-dynamic update procedure.

5 Future Work

This thesis began from the goal of understanding whether tree-decomposition algorithms can be made parallel in dynamic settings, especially in a batch-dynamic model where many edge insertions or deletions are processed together. In studying this question, we found that the static parallel version of the problem is itself not yet fully developed. In particular, after the parallel work of Bodlaender and Hagerup, there has been relatively little progress on parallel algorithms for computing tree decompositions, while most modern developments have focused on sequential static or dynamic algorithms.

The goal of this thesis is therefore best viewed as a first step toward a larger research direction: developing parallel algorithms for tree decomposition, beginning with the static setting and eventually extending the ideas to parallel batch-dynamic updates.

A natural direction is to revisit the main components of classical treewidth algorithms from a parallel perspective. Compression steps often have an inherently parallel flavor: many local reductions, contractions, or deletions may be applicable in different parts of the graph at the same time. This suggests that compression-based approaches could be a useful starting point for parallel static algorithms.

The more delicate part is the improvement step. Classical improvement algorithms, such as the Bodlaender–Kloks dynamic program, use complicated state spaces over a given tree decomposition. Understanding whether these states can be combined in parallel, or whether a different improvement framework is needed, is an important open direction. A possible strategy is to combine modern approximation techniques with older parallel balancing ideas, so that the decomposition has both bounded width and small depth.

There is also a practical direction. The PACE challenges in 2016 and 2017 included treewidth tracks and encouraged implementations of exact, heuristic, sequential, and parallel treewidth solvers [15, 16]. These competitions suggest that practical progress on treewidth computation is possible even when the worst-case theoretical algorithms are complicated. A useful future direction would be to connect the theoretical compression and improvement framework with the engineering perspective of practical solvers: for example, by testing which compression reductions are effective in practice

and which parts of the computation can be parallelized robustly.

Another direction is to study parallel batch-dynamic tree decomposition after a better parallel static theory is developed. In a parallel batch-dynamic model, many updates arrive together, and the algorithm should repair the decomposition using parallel work rather than processing updates one at a time. This requires new ways to identify independent regions of the decomposition tree, resolve conflicts between overlapping repairs, and rebalance the decomposition after a batch of changes.

Overall, the next step is to develop a modern parallel static algorithm for tree decomposition, using ideas from compression, approximation, tree-decomposition balancing, and practical solver design. Once such a static parallel framework is better understood, it may provide the right foundation for parallel batch-dynamic treewidth algorithms.

References

[1] Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. In *Proceedings of the 49th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '08, pages 739–748, 2008.

[2] Hans Bodlaender, Pl Drange, Markus Dregi, Fedor Fomin, Daniel Lokshtanov, and Micha Pilipczuk. A $c^k n$ 5-approximation algorithm for treewidth. *SIAM Journal on Computing*, 45:317–378, 03 2016.

[3] Hans L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '93, page 226–234, New York, NY, USA, 1993. Association for Computing Machinery.

[4] Hans L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11(1–2):1–21, 1993.

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

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

[7] Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms.* Springer, 2015.

[8] Reinhard Diestel. *Graph Theory.* Springer, 5 edition, 2017.

[9] Jörg Flum, Markus Frick, and Martin Grohe. Query evaluation via tree-decompositions. *Journal of the ACM*, 49(6):716–752, 2002.

[10] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. *Journal of Computer and System Sciences*, 64(3):579–627, 2002.

[11] Georg Gottlob, Zoltán Miklós, and Thomas Schwentick. Generalized hypertree decompositions: NP-hardness and tractable variants. In *Proceedings of the Twenty-Fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '05, pages 13–22, New York, NY, USA, 2005. Association for Computing Machinery.

[12] Joseph JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

[13] Tuukka Korhonen. A single-exponential time 2-approximation algorithm for treewidth. *SIAM Journal on Computing*, 0(0):FOCS21–174–FOCS21–194, 0.

[14] Tuukka Korhonen. Dynamic treewidth in logarithmic time. pages 787–796, 12 2025.

[15] PACE Challenge. PACE 2016 – Track A: Tree Width. `https://pacechallenge.org/2016/treewidth/`, 2016. Accessed 2026-05-05.

[16] PACE Challenge. PACE 2017 – Track A: Tree Width. `https://pacechallenge.org/2017/treewidth/`, 2017. Accessed 2026-05-05.

[17] Bruce A. Reed. Finding approximate separators and computing tree width quickly. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing*, STOC '92, page 221–228, New York, NY, USA, 1992. Association for Computing Machinery.

[18] Neil Robertson and P. D. Seymour. Graph minors. II. algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3):309–322, 1986.