FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Strukturelle Parameter von ILP- und MILP-Instanzen aus der Praxis

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Verena Dittmer, B.Sc.
Matrikelnummer 01260244

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Mag. Dr. Stefan Szeider
Mitwirkung: Univ.Ass. Robert Ganian, PhD

Wien, 19. Jänner 2018

Verena Dittmer	Stefan Szeider

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Structural Parameters of Practical ILP and MILP Instances

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Computational Intelligence

by

Verena Dittmer, B.Sc.
Registration Number 01260244

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Mag. Dr. Stefan Szeider
Assistance: Univ.Ass. Robert Ganian, PhD

Vienna, 19th January, 2018

Verena Dittmer

Stefan Szeider

Erklärung zur Verfassung der Arbeit

Verena Dittmer, B.Sc.
Operngasse 32/9, 1040 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 19. Jänner 2018

Verena Dittmer

Kurzfassung

Obwohl die ganzzahlige lineare Programmierung (ILP) und die gemischte ganzzahlige Programmierung (MILP) NP-vollständige Probleme sind, schaffen es moderne Solver diese Probleme mit Millionen von Variablen oder Ungleichungen zu lösen. Trotzdem bleiben bestimmte ILP-Instanzen mit einer relativ geringen Größe immer noch ungelöst. Neueste Fortschritte haben gezeigt, dass manchmal die Struktur von graphischen Modellen der ILP- und MILP-Instanzen (gemessen durch etablierte strukturelle Parameter wie die Treewidth oder Tree-depth) ausgenutzt werden kann um diese effizient zu lösen. In dieser Arbeit analysieren wir die Struktur von graphischen Repräsentationen von ILP- und MILP-Instanzen aus der Praxis, indem wir die Werte von verschiedenen strukturellen Parametern berechnen.

Unser Framework MILP-Struct stellt die Beziehungen zwischen den Variablen und Ungleichungen mittels dem Primal-, Incidence- und Dual-Graphen der ILP- oder MILP-Instanz dar. Auf diesen graphischen Modellen werden dann Unter- und Obergrenzen von den strukturellen Parametern Treewidth, Tree-depth und Torso-width berechnet, für welche in letzter Zeit Fest-Parameter-Algorithmen zum Lösen von ILP oder MILP etabliert worden sind. Die Ergebnisse von MILP-Struct angewendet auf die MIPLIB Bibliothek von praktischen ILP- und MILP-Instanzen zeigen, dass manche der berechneten Parameter tatsächlich viel kleiner als die Anzahl der Variablen sind.

Abstract

Even though Integer Linear Programming (ILP) and Mixed Integer Linear Programming (MILP) are NP-complete problems, state-of-the-art solvers are able to solve instances with millions of variables or constraints. However, certain ILP instances with a relatively small size remain unsolved. Recent advances have shown that in some cases the structure of graphical models of ILP and MILP instances (measured in terms of well-established structural parameters such as treewidth and tree-depth) can be exploited to solve these problems efficiently. In this thesis, we analyze the structure of graphical representations of practical ILP and MILP instances by computing the value of these structural parameters.

We present our framework MILP-Struct that captures the variable-constraint interactions by means of the primal, incidence and dual graph representation of the ILP or MILP instance. On these graphical models, MILP-Struct computes bounds for the structural parameters treewidth, tree-depth and torso-width, which have recently been shown to give rise to fixed-parameter algorithms solving ILP or MILP. Results obtained by applying MILP-Struct on the MIPLIB library of practical MILP and ILP instances show that some of the computed parameters are much smaller than the number of variables.

Contents

CHAPTER 1

Introduction

Besides the Boolean satisfiability problem (SAT) and the constraint satisfaction problem (CSP), integer linear programming (ILP) is one of the most classical NP-complete problems [Kar72]. It is possible to naturally formulate problems as ILP instances; examples include process scheduling [FL05], AI planning [VBS99] and vehicle routing [Lap92].
The ILP problem can be formulated as follows: Given a matrix $A \in \mathbb{Z}^{m \times n}$ and two vectors $b \in \mathbb{Z}^m$, $c \in \mathbb{R}^n$ find a vector $x \in \mathbb{Z}^n$ such that $Ax \leq b$ and cx is maximal. The matrix A is called the constraint matrix and with the vector b the rows can be seen as a set of constraints. When both integer and non-integer variables are considered, one speaks of the Mixed Integer Linear Programming (MILP) problem.
During the last few years, the performance of ILP solvers increased massively. The commercial CPLEX MILP solver, first released in 1991, has undergone a major speedup in each newly released version [Bix12]. The runtime of a test set of about 2000 models were compared on CPLEX Version 1.2 released in 1991, the first version supporting mixed integer programming, with CPLEX 11, that appeared in 2007. Every new version produced a speedup compared to the previous version. On average, the speedup in these years exceeded a factor of two; in other words, each new version was at least two times faster than the one before. In total, this results in a projected, machine-independent improvement of a speedup factor of over 29,000 for CPLEX 11 compared to CPLEX 1.2 [Bix12]. Using the speed comparisons for the mixed integer programming solver Gurobi, where a speedup factor of more than 20 between Gurobi 5.5 (2013) and Gurobi 1.0 (2009) is observed and the fact that Gurobi 1.0 had similar runtimes as CPLEX 11, an impressive combined machine-independent speedup factor of 580,000 between 1991 and 2013 is obtained. The application of different theoretical results from the last 30 years, like the development of branch-and-cut algorithms, are the reason for these software improvements [BKM16].
In addition, in the years from 1993 to 2013 the hardware speedup amounts to approximately a factor of 320,000 measured by the number of floating point operations per

second in supercomputers [BKM16]. Combining the two software improvements of the solvers together with the hardware improvements over this time period, Bertsimas et al. speak of an astonishing 200 billion factor speedup for solving MILP problems [BKM16]. As a result, many ILP and MILP instances that were classified as hard in the beginning can now be solved with relative ease. In spite of these improvements, some ILP or MILP instances are still not solvable by modern solvers, sometimes even when the size of the instance is relatively small in terms of the number of variables and constraints. Koch et al. [KMP13] categorize benchmark instances into three groups: easy instances which may be solved in a few minutes, instances which cannot be solved at all (a specific time limit is always exceeded) and those instances which take relatively long to obtain the solution. The group of in-between instances is hereby relatively small compared to the group of easy and not yet solved instances. A disappointing phenomenon that Koch et al. observe is that even when computers are made faster, the same amount of instances is solved. Some of the in-between instances become easy, but the number of unsolvable instances remains nearly the same. One explanation for this behaviour may be that those instances do not have a certain structure that the solvers can exploit. These instances then need to be solved by "brute force". The NP-completeness of ILP implies that it may take exponential time in general (unless P equals NP).

In order to determine the source of the exponential blow-up in the running time of an ILP instance, one may use the framework of parameterized complexity theory (see [FG06, DF12, DF13, CFK$^+$15]). This allows a more fine-grained investigation of difficult algorithmic problems than classical complexity theory. In the classical setting, time and space complexity is measured only in terms of input size. However, the difficulty of a problem can sometimes be tied to certain structural properties. Parameterized complexity theory may exploit such properties by considering an additional input dimension, a parameter of the problem instance. The parameter is a numerical value depending on the input instance, for example the parameter may be the solution size or some problem-specific structural property. Usually we are interested in instances where the size of the parameter is small compared to the input size. The main notion of interest in parameterized complexity theory is fixed-parameter tractability. This can be seen as a tractable fragment of hard problems, an extension to polynomial-time solvable algorithms by restricting the non-polynomial behaviour only to the parameter [FG06, Preface]. More formally, a parameterized problem is fixed-parameter tractable if there exists an algorithm with runtime $f(k) \cdot poly(n)$ where $f(k)$ is a function depending only on the parameter and $poly(n)$ is a polynomial of the problem input size n. If the parameter is assumed to be fixed instead of being part of the input, a fixed-parameter tractable algorithm runs in polynomial time.

In order to analyze the structure of an ILP or MILP instance, we consider the complexity of variable-constraint interactions. This is achieved by measuring structural properties of graphical representations of instances, in particular the primal, incidence and dual graph representations, which were originally defined for the SAT problem.

1. The primal graph representation contains a vertex for every variable of the instance

and an edge between two vertices, if there is a constraint that contains both variables (with non-zero coefficients in the constraint matrix).

2. The incidence graph has a vertex for every variable and a vertex for every constraint in the instance and an edge between a variable vertex and a constraint vertex if the variable occurs in the constraint.

3. The dual graph contains a vertex for every constraint and has an edge between two constraint vertices if there exists a variable that occurs in both constraints.

The primal and dual graph representations are like opposites to each other, whereas the incidence graph is the only graph that reflects which variable occurs in which constraint. Observe that neither the coefficients of the variables nor the right-hand side value of a constraint is contained in any of the graph representations. However, the structure of the instance is represented in an efficient and compact way. Analyzing structural properties of these graphical representations may thus yield information about the hardness of the ILP instance.
Treewidth is the most prominent structural measure for graphs. Apart from having fundamental connections to graph theory [RS84], it has found an extensive range of applications in many areas of computer science, including Boolean satisfiability [SS09], constraint satisfaction [SS10], and naturally also graph algorithms [Cou90]. At its core, treewidth captures how tree-like a graph (or graph representation of an instance) is. This is very useful from an algorithmic standpoint since graphs of small treewidth often allow the efficient solution of a range of problems; in particular, many problems that are easy on trees can also be solved efficiently (by dynamic programming fixed-parameter algorithms) parameterized by treewidth.
Due to the inherent generality of ILP, treewidth on its own is not sufficient to obtain fixed-parameter tractability for ILP. In fact, there are only a few known cases when an ILP instance is polynomial-time solvable using treewidth. The following two results are thus the more important:

1. ILP FEASIBILITY is fixed-parameter tractable parameterized by the treewidth of the primal graph and the domain size of the variables [JK15]

2. ILP is fixed-parameter tractable parameterized by the treewidth of the incidence graph and the maximum absolute value that can be obtained from a constraint by summing the left-hand side of a constraint over a variable assignment up [GOR17]

Note that for the second result, variable domains need to be bounded. Another well-studied structural parameter is the tree-depth of a graph. This measures how close a graph is to being a star [BDK12]. Unlike the results for treewidth, the following result establishing fixed-parameter tractability of ILP does not depend on the domain size of the variables:

3. ILP is fixed-parameter tractable parameterized by the tree-depth of the extended primal graph of the constraint matrix and the maximum absolute value of a coefficient occurring in the constraint matrix A or in the vector b [GO16].

With these three results for the fixed-parameter tractability of ILP we obtain that if the graph has either bounded treewidth or bounded tree-depth (in case of the primal graph) and some special conditions for the constraint matrix hold, the ILP instance can be solved in polynomial time.
Unfortunately, these results for ILP do not generalize to MILP. For MILP, one natural approach is to separate the polynomial-time solvable non-integer part from the intractable integer part. For this, the primal graph of the MILP instance is considered with the additional information whether a vertex corresponds to an integer or a non-integer variable. With the idea that the integer vertices are the reason for the exponential behaviour of the MILP problem instance, Ganian et al. [GOR17] construct a so called torso of the primal graph, that is obtained by collapsing the non-integer vertices into the integer vertices. The torso-width is then defined to be the treewidth of a torso satisfying certain properties. Similarly as for ILP and the treewidth, MILP is fixed-parameter tractable parameterized by torso-width.
The aim of this thesis is twofold:

1. We provide a framework for the computation of structural parameters of graphical representations of ILP and MILP instances.

2. We then use this framework to compute structural parameters for practical ILP and MILP instances and to analyze the correspondence between those parameters and the hardness of solving the instances.

For the first task, the framework MILP-Struct is provided. It is published under the LGPL license and can be accessed at `https://github.com/kiqo/MILP-Struct`. It is based on the treewidth library LibTW [vDvdHS], that is a Java-based library able to compute tree decompositions and the exact treewidth for small graphs, but also has different implementations of upper and lower bound algorithms for the treewidth. MILP-Struct takes as input a single or multiple ILP or MILP instances and computes one or more graph representations of the programming instance. The graph representation is then translated to the internal graph format of the LibTW library. Different structural parameters may be computed, some of which use the LibTW library. Currently, the following three are supported: The treewidth can be computed for all three graph representations (the primal, incidence and dual graph), the tree-depth and the torso-width can only be computed for primal graphs. In addition to the structural parameters, different statistics about the ILP or MILP instance and its graph representations are computed.
As for the second aim, we apply our framework MILP-Struct on practical instances from the MIPLIB library [mipb, KAA+11] and present the obtained results. The MIPLIB

library is a collection of real-world integer and mixed integer instances from different academic and industrial applications. MILP-Struct is thus the first framework for analyzing structural parameters of graphical representations of ILP or MILP instances.

CHAPTER 2

Preliminaries

This chapter provides definitions for terms that are used throughout the thesis. At first some basic graph notions are defined. Then, integer linear programs and variations of the problem statement are introduced. The primal, incidence and dual graph representations of integer linear programs are defined. After that, three structural parameters of graphs are introduced: the treewidth of a graph [RS84] as the most prominent parameter on which many dynamic programming algorithms are based, the tree-depth [NdM12, Chapter 6] which is also a widely studied parameter, and the torso-width [GOR17] which is a parameter specific for primal graphs of mixed integer linear programs.

2.1 Graph definitions

We refer to the standard textbook by Diestel [Die12] for an in-depth overview of graph terminology. A graph $G = (V, E)$ consists of a set of vertices V and a set of edges E, which is a subset of two-element subsets of V, the set $\binom{V}{2}$. The vertices and edges of a graph G are denoted by $V(G)$ and $E(G)$, respectively. Unless specified otherwise, each graph is assumed to be undirected and simple (no loops and double-edges). An edge $e = \{u, v\}$ is called incident to the vertices u and v. A vertex u is called adjacent to a vertex v if the two vertices are connected by an edge. A clique is a subset of the vertices where there is an edge between any two distinct vertices. A complete graph on n vertices, denoted by K_n, is a graph with n vertices in which there is an edge between any two distinct vertices. If any two vertices of a graph are connected by a path, the graph is connected. A connected component of a graph is a maximal subgraph where every two vertices are connected by a path. A connected graph that does not contain any cycle is a tree. A forest is a disjoint union of trees. The set of vertices that are adjacent to a vertex v are called the neighbours of v. The degree of a vertex v is the number of neighbours of v, denoted $d(v)$. The neighbours of a set of vertices W, also denoted as $N(W)$, is the set $\{v \in V(G) \backslash W \mid \exists w \in W$ such that v is adjacent to $w\}$.

The subgraph of $G = (V, E)$ induced by a vertex set W, denoted $G[W]$, is defined by $G[W] = (W, \{\{v, w\} \in E(G) \mid v \in W \wedge w \in W\})$. A bipartite graph $G = (V \cup W, E)$ is a graph that can be divided into two disjoint vertex sets V and W and every edge in G is incident to a vertex in V and a vertex in W. A vertex set $S \subseteq V$ is a separator of G if $G[V \backslash S]$ has more than one connected component. The contraction of an edge $\{u, v\} \in E$ is the operation of removing the edge $\{u, v\}$ and the two vertices u and v and replacing it by a new vertex x which is adjacent to all neighbours of u and of v. A graph H is a minor of a graph G if H is the result of applying one or more contractions to some subgraph of G. This means that H can be obtained from G by vertex deletions, edge deletions or edge contractions. A vertex set X in a graph G is collapsed, denoted $G \circ X$, if X is deleted from the graph and an edge is added between each pair of neighbours of X. In the resulting graph $G \circ X$, the neighbours of X form a clique. If the set X consists of only one vertex v, then we also speak about the process of eliminating v from the graph G. Note that eliminating every vertex v of a set of vertices X results in the same graph as $G \circ X$ and that the order of the elimination does not influence the result graph.

2.2 Integer linear programs

An integer linear program (ILP), also called an integer program, is the following problem:

Given An integer matrix $A \in \mathbb{Z}^{m \times n}$ and two vectors $b \in \mathbb{Z}^m$, $c \in \mathbb{R}^n$

Task Find a vector $x \in \mathbb{Z}^n$ such that $Ax \leq b$ and cx is maximal

Closely related to this problem is the ILP FEASIBILITY problem: Given an integer matrix $A \in \mathbb{Z}^{m \times n}$ and a vector $b \in \mathbb{Z}^m$ it asks whether there exists a vector $x \in \mathbb{Z}^n$ such that $Ax \leq b$. Both ILP and ILP FEASIBILITY are well-known to be NP-complete [Kar72]. The well-studied linear programming problem (LP) has the same formulation, but without the restriction of the vector x to take integer values. The matrix A is called the constraint matrix, cx is called the objective function.
A matrix is totally unimodular if every square submatrix has determinant $1, -1$, or 0. This implies that a matrix can only be totally unimodular when every entry is in $\{1, -1, 0\}$ as every entry corresponds to a one by one square submatrix. If the constraint matrix A of an ILP instance is totally unimodular, the ILP instance can be solved in polynomial time [Sch86, Chapter 19]. Moreover, it can be tested in polynomial time whether A is totally unimodular [Sch86, Chapter 19, 20]. This represents the most classical example of a tractable ILP fragment.

For our purposes we will also use an equivalent, constraint-based representation of an ILP instance adapted from [GOR17]. An ILP instance I is a tuple (F, η), where F is a set of linear constraints over variables $X = \{x_1, ..., x_n\}$. Each constraint $A_i \in F$ has the form $a_{i1}x_1 + a_{i2}x_2 + ... + a_{in}x_n \leq b_i$. The objective function η is a linear function over X of the form $\eta(X) = c_1x_1 + c_2x_2 + ... + c_nx_n$. As a convention, we omit variables from

constraints or the objective function whose coefficient is equal to zero. The variables with non-zero coefficients of $A_i \in F$ are denoted by $var(A_i)$. The variables of I are also written as $var(I)$.

A mixed integer linear program (MILP) contains both real and integer variables and is thus a "mix" of a LP and ILP. Formally, a MILP instance I is a tuple (F, η), where F is a set of linear constraints over the disjoint variable set $X \cup Y$, where X is the set of integer variables, denoted by $var_{\mathbb{Z}}(I)$, and Y is the set of real variables, denoted by $var_{\mathbb{R}}(I)$. η is a linear function over the variables $X \cup Y$.

The following three graph representations can be defined on the constraint matrix of any ILP or MILP instance. For simplicity it is stated only for ILP instances but can also be defined on MILP instances.

Definition 2.1 (Primal graph)**.** *The primal graph $G_I = (V, E)$ of an ILP instance $I = (F, \eta)$ is the graph that has the variables of I as its vertices and contains an edge for every two variables that occur with non-zero coefficients together in a constraint, i.e. $V = var(I)$ and $E = \{\{x_i, x_j\} \mid x_i, x_j \in var(I)$ and $\exists A_k \in F$ with $a_{ki} \neq 0$ and $a_{kj} \neq 0\}$.*
The extended primal graph of $I = (F, \eta)$ has the variables of I as its vertices and contains an edge for every two variables that occur with non-zero coefficients together in a constraint or in the objective function η.

The primal graph of the ILP instance is sometimes also called the Gaifman-graph of the constraint matrix in literature. However, similar to the approach in Boolean satisfiability, we will further refer to it as the primal graph.

Definition 2.2 (Incidence graph)**.** *The incidence graph $H_I = (V, E)$ of an ILP instance $I = (F, \eta)$ is the graph that has the variables of I and the constraints F as its vertices and contains an edge between a variable vertex and a constraint vertex if the variable has a non-zero coefficient in the constraint, i.e. $V = var(I) \cup F$ and $E = \{\{x_i, A_j\} \mid x_i \in var(I)$ and $A_j \in F$ and $a_{ji} \neq 0\}$.*
The extended incidence graph of I is the graph that has the variables of I, the constraints F and the objective function η as its vertices and contains an edge between a variable vertex and a constraint vertex, or the objective function vertex if the variable has a non-zero coefficient in the constraint or in the objective function.

The incidence graph gives a more refined view on the structure of the underlying problem instance. The incidence graph of the constraint matrix contains information to determine which variable occurs in which constraint. In contrast, the primal graph contains only information about whether two variables appear together in a constraint, but not in which specific constraint.

Definition 2.3 (Dual graph)**.** *The dual graph $J_I = (V, E)$ of an ILP instance $I = (F, \eta)$ is the graph that has the constraints F as its vertices and contains an edge between two*

constraint vertices if there exists a variable that occurs in both constraints with a non-zero coefficient, i.e. $V = F$ *and* $E = \{\{A_i, A_j\} \mid A_i \in F \text{ and } A_j \in F \text{ and } \exists x_k \in var(I) \text{ with } a_{ik} \neq 0 \text{ and } a_{jk} \neq 0\}$.
The extended dual graph of I *is the graph that has the constraints* F *and the objective function* η *as its vertices and contains an edge between a constraint vertex and the objective function vertex or two constraint vertices if there exists a variable with a non-zero coefficient that occurs in both constraints, in case both vertices are constraint vertices, or in the constraint and the objective function otherwise.*

When the objective function is not considered in the graph representation, we also speak of the simplified primal, incidence or dual graph representation in order to distinguish them of the extended graph representations (see also [GOR17]).

2.3 Parameterized complexity

Parameterized complexity theory (see [FG06, DF12, DF13, CFK+15]) gives a more detailed view of hard algorithmic problems. As a rather new branch of complexity theory, it was brought forward in a series of articles by Downey and Fellows [DF92, DF95a, DF95b] in the mid-1990s. They defined the complexity class fixed-parameter tractable, introduced reductions for parameterized problems and proved fundamental completeness results for various problems [FG06, Preface]. In normal complexity theory, the runtime of a problem instance is usually studied by its input size n. When an additional parameter k is considered, one obtains parameterized complexity classes. This parameter is a numerical value which is in a certain way dependent on the input. The parameter may either correspond to the value of the problem objective function, or it may measure structural properties of the input instance [FLM+08]. The parameter is usually small compared to the input size of the instance. For example, consider the problem of evaluating a query over a database. Whereas the size of the query is small, the size of the database is usually much larger. A natural parameter for this problem is then the size of the input query [FG06, Preface]. Formally, decision problems are described as languages over finite nonempty alphabets Σ.

Definition 2.4 ([FG06, Chapter 1])**.** *Let* Σ *be a finite alphabet.*

1. *A parameterization of* Σ^* *is a mapping* $\kappa : \Sigma^* \to \mathbb{N}$ *that can be computed in polynomial time.*
2. *A parameterized problem over* Σ *is a pair* (Q, Σ^*) *where* $Q \subseteq \Sigma^*$ *is a set of strings over* Σ *and* κ *is a parameterization of* Σ^*.

For a given parameterized problem (Q, κ), the strings $x \in \Sigma^*$ are called instances of Q or (Q, κ), and the numbers $\kappa(x)$ are called parameters. Such problems are represented in the following form [FG06, Chapter 1]:

Instance $x \in \Sigma^*$

Parameter $\kappa(x)$

Problem Decide whether $x \in Q$

Compared to classical complexity theory, an additional dimension is considered. This leads to a more refined complexity hierarchy than in the classical complexity theory [FG06, Preface].
For problems that are known to be NP-complete or harder, one hopes to obtain algorithms that run in time $f(\kappa(x)) \cdot poly(|x|)$, where f is a computable function dependent only on the parameter $\kappa(x)$, and $poly(|x|)$ is a polynomial in the length of the string (or size of the input), i.e. $poly(|x|) \in O(|x|^c)$ for some constant c [FG06, Chapter 1].

Definition 2.5 (Fixed-parameter tractability [FG06, Chapter 1])**.** *Let Σ be a finite alphabet and $\kappa : \Sigma^* \to \mathbb{N}$ be a parameterization.*

1. *An algorithm A with an input alphabet Σ is a fixed-paramater tractable algorithm with respect to the parameterization κ if the running time of the algorithm A on every input $x \in \Sigma^*$ is bounded by*

$$f(\kappa(x)) \cdot poly(|x|)$$

   where $f : \mathbb{N} \to \mathbb{N}$ is a computable function.

2. *A parameterized problem (Q, κ) is fixed-paramater tractable if there exists a fixed-parameter tractable algorithm with respect to κ that decides Q. The complexity class FPT is the class of fixed-paramater tractable problems.*

For graph problems we will denote the size of the input graph as n and the parameter as k. Consider the two runtimes n^k and $2^k \cdot n$. Even though both are exponential in k, the first "blows up" the whole problem, whereas the second is only exponential in k but in a way which is independent of the input size. The complexity class FPT keeps the non-polynomial behaviour of a problem restricted by the parameter; it can thus also be seen as a relaxation of classical tractability, the class of polynomial time solvable algorithms [FG06, Chapter 1].

2.4 Treewidth

The treewidth is the most prominent structural parameter of a graph. Many graph problems can be solved efficiently by dynamic programming algorithms on graphs of bounded treewidth. Often, they even turn out to be fixed-parameter tractable parameterized by the treewidth. The concept of a tree decomposition and treewidth was introduced by Robertson and Seymour [RS84].

Definition 2.6. *A tree decomposition of a graph G is a pair (T, X), where T is a tree and $X = \{X(t) \mid t \in V(T)\}$ is a family of subsets of $V(G)$ with the following properties:*

W1 $\bigcup_{t \in V(T)} X(t) = V(G)$

W2 *For all edges $\{u, v\} \in E(G)$ there exists $t \in V(T)$ such that $u \in X(t)$ and $v \in X(t)$*

W3 *For all $i, j, k \in V(T)$: if j is on the path from i to k in $V(T)$, then $X(i) \cap X(k) \subseteq X(j)$*

When replacing the third property by the following, one obtains an equivalent definition [Bod98]:

W3' For all $v \in V(G)$ the set of vertices $\{t \in V(T) \mid v \in X(t)\}$ forms a connected subgraph (i.e. a subtree) of T

The elements of X, the sets $X(t)$, are referred to as bags of the tree decomposition in order to distinguish them of the set of vertices $V(G)$ of the graph. The width of a tree decomposition (T, X) is $max_{t \in V(T)} |X(t)| - 1$. The treewidth of a graph G, also denoted as $tw(G)$, is the minimum width w such that there is a tree decomposition of G of width w.

An example of a tree decomposition can be found in Figure 2.1. On the left-hand side a graph and on the right-hand side a tree decomposition (T, X) of it is displayed. T clearly forms a tree. The set X corresponds to the contents of the bag, for example $\{a, b, c\} \in X$. Note that the three conditions of a tree decomposition are fulfilled: Every vertex of the graph occurs in one of the bags. For every edge there exists some bag in the tree decomposition such that both endpoints are contained in the bag. For every vertex x in the graph it holds that when looking at the subgraph of bags that contain x, this subgraph is a tree. The tree decomposition contains three elements in every bag. Therefore, the width of this tree decomposition is two.

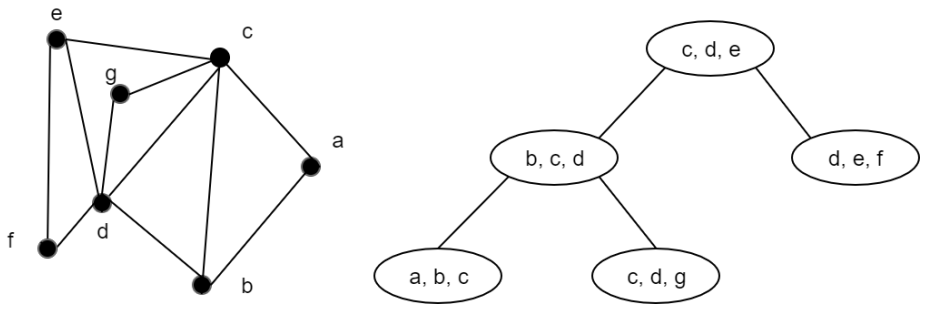


Figure 2.1: An example graph on the left side and a possible tree decomposition on the right side

Some simple observations of the treewidth include the following: The treewidth is at least one if the graph $G = (V, E)$ contains an edge. A well known lower bound for the treewidth is the degree of the lowest-degree vertex in G [BK11]. The trivial tree decomposition, where there is a single bag that contains all the vertices, has as width the number of vertices minus one. Thus, any graph has treewidth at most $n - 1$ if n is the number of vertices in the graph. The treewidth of a forest, and thus also of a tree, is one. The following result is widely known.

Lemma 2.1 ([Sze04])**.** *Let (T, X) be a tree decomposition of a graph G and let $K \subseteq V(G)$ be a set of vertices that induces a complete subgraph in G. Then $K \subseteq X(t)$ holds for some $t \in V(T)$.*

The treewidth of a graph that contains a clique of size n is thus at least $n - 1$. Note that the degree of the lowest-degree vertex is at least $n - 1$ which also implies that the treewidth is at least $n - 1$. For a complete bipartite graph $G = (V \cup W, E)$ the treewidth is $min\{|V|, |W|\}$. The lowest-degree vertex of G has degree $min\{|V|, |W|\}$ and thus the treewidth of a complete bipartite graph must be at least $min\{|V|, |W|\}$. By constructing a tree decomposition that contains in every bag the vertices from the smaller side together with one vertex from the other side, as depicted in Figure 2.2 (for a bipartite graph), one obtains this optimal width. For graphs that are not connected, the following lemma

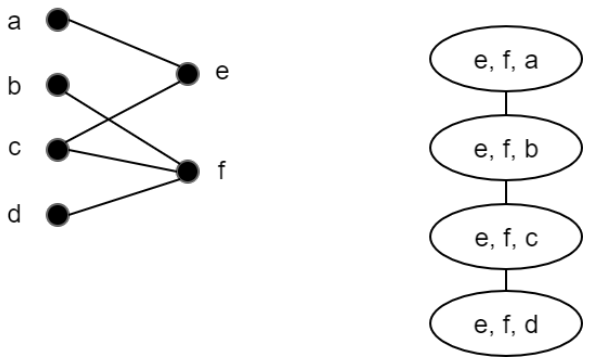


Figure 2.2: A bipartite graph on the left side and a tree decomposition of the graph with the optimal width two on the right side

connects the treewidth of the connected components with the treewidth of the total graph:

Lemma 2.2 ([Bod98])**.** *Let G be a graph. Then the treewidth of G is equal to the maximum treewidth of its connected components.*

In general, the treewidth of a graph is not easy to compute. In particular, given a graph G and an integer k, the problem of determining whether the treewidth of G is at most k is NP-complete [ACP87]. We refer to the following parameterized problem as the Treewidth problem.

Input G, k

Parameter k

Task Obtain a tree decomposition for G of width at most k or correctly identify that no such decomposition exists.

The following well-known treewidth result from Boolean satisfiability [Sze04] also applies to ILP and MILP instances.

Theorem 2.1 (Following [Sze04, Lemma 4])**.** *Let $I = (F, \eta)$ be an ILP or MILP instance. Let $tw(G_P)$, $tw(G_I)$ and $tw(G_D)$ be the treewidth of the primal, incidence and dual graph of I. Then*

1. $tw(G_I) \leq tw(G_P) + 1$
2. $tw(G_I) \leq tw(G_D) + 1$

Proof. 1. Let (T, X) be a tree decomposition of G_P of width k. By Lemma 2.1 for every constraint $A_i \in F$ there exists a vertex $t_{A_i} \in V(T)$ such that $var(A_i) \subseteq X(t_{A_i})$. The tree T' is obtained from T by adding for every constraint $A_i \in F$ a new vertex t'_{A_i} and an edge $\{t_{A_i}, t'_{A_i}\}$ to T'. X' is obtained by extending X to include the new vertex $t'_{A_i} \in V(T')$ with $X'(t'_{A_i}) = var(A_i) \cup \{A_i\}$. (T', X') is then a tree decomposition of G_I as the conditions W1–W3 are fulfilled. Let $w(F)$ be the maximum number $|var(A_i)|$ over all $A_i \in F$. The width of the (T', X') is then at most the maximum of $tw(G_P)$ and $w(F)$ and $w(F) \leq tw(G_P) + 1$ by Lemma 2.1. Thus, $tw(G_I) \leq tw(G_P) + 1$.

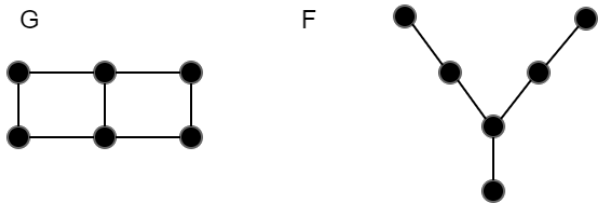
2. The proof proceeds analogous as in 1. except that a tree decomposition of the dual graph G_D is given. □

Note that Theorem 2.1 can also be applied to extended graph representations.

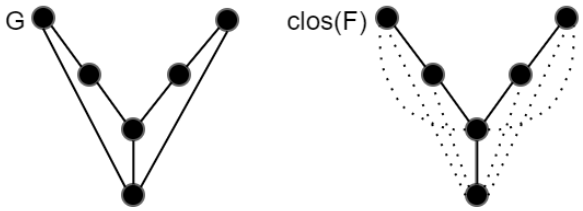
2.5 Tree-depth

Tree-depth is, like the treewidth, a structural parameter of graphs which appears under different names in literature. The following definition from [NdM06] is based on the height of rooted forests. A rooted forest is a disjoint union of rooted trees where a rooted tree is a tree that has a distinguished vertex, the root vertex. For a vertex x in a rooted forest F the height of x is the number of vertices of the path from x to the root. The maximum height over all vertices of F is the height of F. Given two vertices x and y of a rooted forest F, the vertex x is an ancestor of y in F if x appears on the (unique) path from y to the root of the tree of F to which y belongs. The closure $clos(F)$ of a rooted forest F is defined to be the graph that has the same vertex set $V(F)$ and has as

edge set the same edges as in F and additional edges between two vertices if they are ancestors, i.e. the edge set is the set $\{\{x, y\} \mid x \text{ is an ancestor of } y \text{ in } F \text{ and } x \neq y\}$. The tree-depth $td(G)$ of a graph G is the minimum height over all rooted forests F such that it holds that $G \subseteq clos(F)$ [NdM06].



(a) A 2×3 grid graph G and a rooted forest F for G



(b) The graph G is a subgraph of the closure of F

Figure 2.3: Example for the tree-depth of a 2×3 grid graph

A rooted forest F for a 2×3 grid graph G can be seen in Figure 2.3. When rearranging the graph G in 2.3b, one can see that it is a subgraph of the closure of F. The height of the rooted forest F is four. Thus, the tree-depth of G is at most four.

Another definition of the tree-depth is based on elimination trees [NdM06]. The elimination tree Y of a connected graph G is defined recursively in the following way. If G consists of a single vertex x then the rooted tree Y is $\{x\}$. Otherwise choose a vertex $r \in V(G)$ to be the root of Y. Let $G_1, ..., G_p$ bet the connected components of $G - r$. Construct recursively for every component G_i the elimination tree Y_i of G_i. Y is then constructed by adding an edge $\{r, r_i\}$ between the root r_i of every Y_i and r [NdM06]. By the following lemma we obtain that the two definitions of tree-depth are equivalent:

Lemma 2.3 ([NdM06])**.** *Let G be a connected graph. A rooted tree Y is an elimination tree for G if and only if $G \subseteq clos(Y)$. Hence $td(G)$ is the minimum height of an elimination tree for G.*

From this lemma we can obtain the following inductive definition:

Lemma 2.4 ([NdM06])**.** *Let G be a graph with connected components $G_1, ..., G_p$. Then*

$$td(G) = \begin{cases} 1, & if\ |V(G)| = 1; \\ 1 + min_{v \in V(G)}\ td(G - v), & if\ p = 1\ and\ |V(G)| > 1; \\ max_{i=1,..,p}\ td(G_i), & otherwise \end{cases}$$

We remark that the concept of minimum elimination trees is well-studied in literature and appears under different notions like the rank function, vertex ranking number or cycle rank (see [NdM12, Chapter 6]). Every depth-first search tree of a connected graph G is also an elimination tree for G [NdM12, Chapter 6].

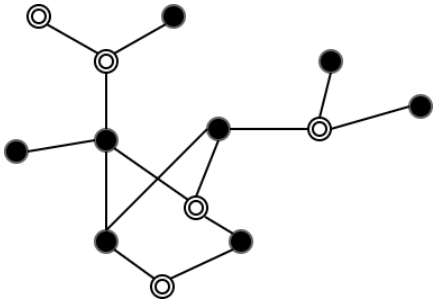
2.6 Torso-width

Torso-width is a structural parameter specifically designed for MILP instances introduced by Ganian et al. [GOR17]. Given a MILP instance $I = (F, \eta)$. Let q be an arbitrary constant. An integer variable x_i has bounded domain if x_i has a lower bound c and an upper bound d, i.e. $x \leq c$ and $x \geq d$ are constraints in F. Moreover, x_i has q-bounded domain if $c - d \leq q$. Let $B_q(I)$ be the set of all q-bounded domain variables and let $U_q(I) = var(I) \setminus B_q(I)$ be the set of q-unbounded domain variables. Note that the set of non-integer variables are a subset of $U_q(I)$.
Let G_I be the primal graph representation of I. A graph G is a q-torso of I iff there exists a set $S \supseteq U_q(I)$, such that $G = G_I \circ S$. A q-torso corresponds to the graph that is obtained after collapsing at least all non-integer variables and those integer variables that are not q-bounded, and possibly some q-bounded integer variables.
Let $G = G_I \circ S$ be a q-torso of I obtained by collapsing the vertex set S and H a connected component of $G_I[S]$. The fitness of H is the number of integer variables in H. The fitness of G, denoted $\tau(G)$, is the maximum fitness over all connected components of $G_I[S]$. The torso-width of the q-torso G is defined as $torw(G) = max\{tw(G), \tau(G)\}$ where $tw(G)$ is the treewidth of G [GOR17].

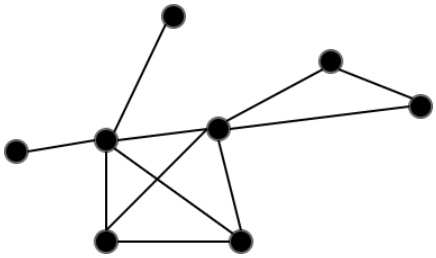
Definition 2.7 (q-torso-width)**.** *The q-torso-width of a MILP instance I is the minimum $torw(G)$ over all q-torsos G of I.*

A concrete example for the torso-width of a MILP instance can be found in Figure 2.4. Here the filled vertices correspond to q-bounded variables, while the non-filled vertices correspond to variables which are not q-bounded. Figure 2.4b is a q-torso obtained from the primal graph of a MILP instance in Figure 2.4a. Observe that the degree of some vertices increases. In this case also the treewidth increases from two to three as the torso in Figure 2.4b contains a clique of size four as a subgraph. The q-torso in Figure 2.4c is obtained by collapsing all the vertices that correspond to the q-unbounded variables plus the bottom-left q-bounded variable vertex. The treewidth of this torso is two, therefore the q-torso-width of the MILP instance is at most two.

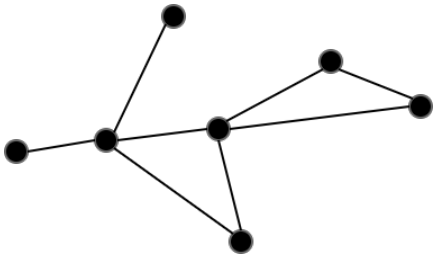
For the purposes of this thesis, we will also define and consider the notion of ∞-torso and ∞-torso-width, which naturally correspond to the case where we consider q to be unbounded. The ∞-torso of a MILP instance I is a graph that is obtained after collapsing at least all vertices that correspond to the non-integer variables in the instance I. For simplicity, we call the vertices in the primal graph of a MILP instance integer and non-integer vertices. Observe that when a ∞-torso G is obtained by only collapsing the non-integer vertices, the ∞-torso-width of G is precisely $tw(G)$. Furthermore, the



(a) The primal graph of a MILP instance. The filled vertices correspond to q-bounded variables, the non-filled vertices to q-unbounded variables.



(b) A possible q-torso obtained by collapsing the set of vertices that correspond to q-unbounded variables



(c) A possible q-torso obtained by collapsing the set of vertices that correspond to q-unbounded variables plus one of the q-bounded variables

Figure 2.4: Example for possible q-torsos

∞-torso-width of a MILP instance I is clearly a lower bound for its q-torso-width, for any finite q, as the ∞-torso-width may consider the width of more torsos than in the case of the q-torso-width.
Finally, from the proof of [GOR17, Lemma 4] it follows that any q-torso of I is a 2-approximation of an optimal q-torso. For the ∞-torso-width of I this implies that for the torso G obtained by collapsing the non-integer vertices, it holds that the ∞-torso-width is at least $0.5 \cdot tw(G)$.

CHAPTER 3

Related Work

This chapter gives an overview over the state-of-the-art results for structural parameters, computing them, and the associated complexity results. At first, results for the computation time of the Treewidth problem and its membership in the complexity class FPT are presented. We also provide an overview of implementations for computing tree decompositions or treewidth heuristics. These often rely on practically more time-efficient lower and upper bound methods for treewidth, which are presented and compared in the following sections. After that, lower and upper bound methods for tree-depth are elaborated. State-of-the-art results from the application of structural parameters for the ILP and MILP problems are presented in the last section.

3.1 Fixed-parameter tractability of treewidth

Most applications of treewidth assume that a tree decomposition of small width is provided as part of the input. For this reason, the efficient computation of a tree decomposition with small width is important. Bodlaender [Bod93] presented a fixed-parameter tractable algorithm for Treewidth; in particular, his algorithm computes a tree decomposition of width at most k or outputs that the treewidth is larger than k with a linear dependency on the size of the input graph. This algorithm can be used for theoretical results, but in practical applications the dependency on the parameter is very large, such that even for small values of k the algorithm is not applicable. Fomin et al. [FKT04] give an algorithm that uses minimal separators and potential maximal cliques and computes the treewidth in time $O(1.9601^n \text{ poly}(n))$. With combinatorial proofs Fomin et al. [FKTV08] show that this algorithm even runs in time $O(1.8899^n \text{ poly}(n))$. Bodlaender et al. [BFK$^+$12] present a theoretical algorithm based on balanced separators that has runtime $O(2.9512^n \text{ poly}(n))$ and uses polynomial space.

Feige et al. [FHL05] present an approximation algorithm, which runs in polynomial time and finds a tree decomposition of width $O(k\sqrt{log\ k})$ if the treewidth of the graph is k.

Name	Year	Downloadable	Open-Source	Description
QuickTree	1997	No	-	Algorithm for computing a minimal triangulation of the graph
QuickBB	2004	Yes	No	Anytime branch and bound algorithm for finding perfect elimination orderings
Hypertree	2005[1]	Yes	No	Methods for computing hypertree decompositions, some of which include the computation of tree decompositions
LibTW	2006	Yes	Yes	Library that provides several implementations for existing algorithms to compute treewidth lower and upper bounds, as well as exact methods
dlib	2009	Yes	Yes	Machine-learning toolkit that contains graph tools including a method for computing a tree decomposition
Toto	2017	No	-	Open database for tree decompositions accessible via a website; Services for computing bounds or the exact treewidth including the corresponding tree decomposition

Table 3.1: Methods for computing exact or bounds on treewidth

More recently, a 5-approximation algorithm for treewidth presented by Bodlaender et al. [BDD+13] runs in time single-exponential in k and linear in n. It either tells that the treewidth is larger than k, or returns a tree decomposition of width at most $5k + 4$.

Many different implementations of algorithms for treewidth or treewidth bounds exist. An overview of some implementations can be found in Table 3.1, sorted chronologically. The column 'Year' notes hereby the year of the appearance of the scientific paper about the treewidth method. 'Downloadable' notes whether we were able to find a downloadable version of it; 'Open-Source' means whether the implementation that is available for download also provides the program's source code. Some of these libraries include the computation of tree decompositions whereas others are aiming for the computation of lower and upper bounds for treewidth.
QuickTree [SG97] is an algorithm for the triangulation of a graph. It outputs a triangulated graph, where the size of the largest clique is the minimum, and a perfect elimination

[1] Denotes the year where the basis for the implementation-framework was completed [hyp]

sequence. The size of the largest clique in the result graph minus one is the treewidth of the graph, thus the algorithm computes the exact treewidth. It was tested against graphs of up to 100 vertices that have a treewidth up to 10.
QuickBB [GD04] is an algorithm for finding perfect elimination orderings of a graph and implemented as a branch and bound algorithm. In terms of cpu time, it performs up to 50 times better than Quicktree tested on randomly generated graphs with 100 vertices and treewidth up to 10. Instances with a high treewidth are preferred over instances with low treewidth as its runtime is $O(n^{n-k})$. Its advantage is that it is an anytime algorithm: after the algorithm exceeds a specified time limit, it still returns valid results of lower and upper bounds for treewidth. If the time limit is increased, QuickBB will give better results.
The focus of Hypertree [hyp] are hypertree decompositions, which are generalizations of tree decompositions. Hypertree offers many possibilities to compute hypertree decompositions, some of which include the computation of tree decompositions. For an overview of hypertree decomposition computation methods and how tree decompositions are used for it, see [DGG$^+$08].
Compared to the other methods, LibTW [vDvdHS06] is a library with many implementations of lower and upper bound heuristics as well as exact computation algorithms. It is an open-source library implemented in Java [vDvdHS] licensed with the LGPL license, that allows for modification of the source code. There are five upper bound heuristics and five lower bound heuristics implemented including several variants of the lower bound heuristics. In addition, two exact algorithms, a dynamic programming algorithm and the branch and bound algorithm QuickBB (with some adaptations) are implemented and tested on graphs with up to 50 vertices [vDvdHS06]. The large pool of implemented algorithms allows to decide whether only lower and upper bounds, the exact treewidth, or a tree decomposition of optimal width are computed. If GraphViz [GN00] is installed, LibTW may also show a graphical representation of the input graph or computed tree decomposition [vDvdHS].
Dlib [Kin09, dli] is an open-source library written in C++ for providing machine-learning algorithms. In addition, it contains many graph tools for handling both undirected and directed graphs. The `create_join_tree` method in `<dlib/graph_utils.h>` finds a tree decomposition of a given graph. However, dlib does not state any properties about the width of the tree decomposition that is returned by this method.
Relatively new is the open database Toto [vWK17] which offers several services related to tree decompositions: it provides a graph database with known lower and upper bounds for treewidth and the best existing tree decomposition, allows for the upload of better tree decompositions, and offers a service to compute a tree decomposition for an arbitrary graph. At the moment, the database only stores graphs with up to 150 vertices, but the computation can also be applied to graphs that exceed this size. For the computation of lower bounds, it uses the MMD+(least-c) method. The Greedy-Fill-In heuristic is used as upper bound algorithm. For the exact computation of treewidth QuickBB is used. Whereas the lower and upper bound heuristics are executed at server-side, the QuickBB algorithm runs in the web browser of the user. The services are accessible by a website or

may be accessed via an API for programming languages such as Java or PHP [vWK17].

3.2 Lower bound algorithms for treewidth

In practical applications the graphs may have a much larger number of vertices and edges and thus computing the exact treewidth may not be possible. For such large graphs the treewidth may also be very large, in the worst case the number of vertices minus one. As a consequence, algorithms which have a runtime exponential in the treewidth usually do not terminate. In some cases the tree decomposition of a graph itself is not necessary, but one may want to have an approximate value of the treewidth. Hence, one may consider lower and upper bounds for the treewidth, which are faster to compute than exact methods or approximations and thus may be applicable for larger graph instances. Good lower bounds play an important role for branch and bound algorithms like QuickBB [BK11]. With high lower bounds more branches can be cut off and the search space decreases. In addition, when given a good lower bound on the treewidth of a graph it may be possible to determine that using a dynamic programming algorithm will take much time. Also some preprocessing steps are only applicable for graphs, when the treewidth is large enough [BK11].
Bodlaender and Koster [BK11] give a summary of approaches for obtaining lower bound algorithms, some of which are explained here. A lower bound on the treewidth can be obtained by the degrees of the vertices in the graph. Let $\delta(G)$ denote the degree of the lowest-degree vertex in G and $\delta_2(G)$ denote the degree of the second-lowest-degree vertex in G.

Lemma 3.1 ([BK11]). *Let $G = (V, E)$ be a graph of treewidth at most k. Then*

i. $k \geq \delta(G)$

ii. If G contains at least two vertices, then $k \geq \delta_2(G)$.

iii. If G is not a clique, then G contains at least two vertices v, w of degree at most k such that v and w are not adjacent.

The first property is a well known lower bound on the treewidth. It is often referred to as Min-Degree-Lemma. Ramachandramurthi [Ram97] proposed a new lower bound, based on the third property: Let $\gamma(G)$ be defined as $min_{\{v,w\} \notin E} max\{d(v), d(w)\}$ if G is not a clique and else $|V| - 1$. Then the following holds:

Corollary 3.1 ([BK11, Ram97]). *For each graph G, $tw(G) \geq \gamma(G) \geq \delta_2(G) \geq \delta(G)$.*

$\gamma(G), \delta_2(G)$ and $\delta(G)$ can be computed in linear time [BK11, KWB05]. However, the obtained lower bounds are usually bad since one or two low-degree vertices make the lower bound very low even though the graph may contain a large clique. Using the following two corollaries one can obtain better lower bounds.

Corollary 3.2 ([BK11]). *Let H be a subgraph of G. Then $tw(H) \leq tw(G)$.*

Corollary 3.3 ([BK11]). *Let H be a minor of G. Then $tw(H) \leq tw(G)$.*

So one can compute the lowest vertex degree, second-lowest vertex degree and $\gamma(G)$ over all subraphs or minors of the graph and obtain lower bounds on the treewidth of the original graph. For example, if a graph contains a clique, all three lower bound values are then at least the size of the clique minus one.
The degeneracy of a graph is the maximum of $\delta(H)$ over all subgraphs H of G. It can be computed with the Maximum Minimum Degree (MMD) algorithm by repeatedly deleting the vertex with the current lowest degree. The degeneracy is set to the maximum of all vertex degrees at the time of their deletion (see [BK11] for the exact algorithm). The contraction degeneracy of G is the maximum of $\delta(H)$ over all minors H of G. Computing the contraction degeneracy is NP-complete [BKW04]. Instead, one may use a heuristic for computing a lower bound on it as any lower bound on the contraction degeneracy is also a lower bound for the treewidth. The algorithm MMD+ [BK11, GD04, BKW04] computes a lower bound on the contraction degeneracy by repeatedly contracting the vertex that has the current lowest degree with some neighbour vertex. The contraction degeneracy is set to the maximum of all vertex degrees at the time of their contraction with a neighbour vertex. There are different strategies to choose the neighbour vertex: either one chooses the neighbour of lowest degree, called min-d strategy, the neighbour of largest degree, the max-d strategy, or the neighbour that has a minimum of common neighbours, the least-c strategy [BK11]. The last heuristic, MMD+(least-c), performs better than the other two in an experimental evaluation [BKW04].

The Maximum Cardinality Search (MCS) algorithm introduced by Tarjan and Yannakakis [TY84] is a different approach for obtaining a lower bound on the treewidth. First, it was used for obtaining an upper bound on the treewidth, Lucena [Luc03] then showed how it can be used as a lower bound [BK11]. MCS is a certain way to visit the vertices in a graph. In the beginning no vertex is visited. In each step an unvisited vertex with the largest number of already visited neighbours is chosen to be visited. This order is clearly not unique: choosing the starting vertex and choosing one of the possible vertices, if there are more vertices with the same number of already visited neighbours, may produce different orderings. Such an ordering can be used for obtaining a lower bound: If a vertex v is adjacent to k already visited neighbours according to the MCS ordering, then the treewidth of the graph is at least k. Consequently, the best lower bound that can be obtained by a given MCS ordering is by considering the vertex that has the largest number of already visited neighbours at the time of its visit. The lower bound obtained is then called the MCS lower bound [Luc03, BK11].

Another lower bound method can be obtained from improved graphs, first used as a lower bound by Clautiaux et al. [CCMN03, BK11]. These graphs are obtained by repeatedly adding an edge between two non-adjacent vertices v and w if there are $k+1$ vertex disjoint paths going from v to w. The graph obtained G_p is then called the $(k+1)$-path improved graph of G, with the property that the treewidth of G_p equals the treewidth of

G [Bod00, BK11].
The LBP algorithm that computes a lower bound on the treewidth of a graph [CCMN03, BK11] uses a different lower bound algorithm (e.g. MMD or MMD+). It starts by using this lower bound algorithm and computes a lower bound l for the current graph. It then computes the $(l+1)$-improved graph and recomputes the lower bound. If the lower bound obtained is larger, then l is increased by one. This is done until the lower bound l does not change any more. Instead of considering $(k+1)$-path improved graphs, one may also consider $(k+1)$-neighbour improved graphs, where an edge is added between two non-adjacent vertices v and w if they have at least $k+1$ common neighbours. The LBN algorithm is analogous to the LBP algorithm with the difference that the $(k+1)$-neighbour improved graph is constructed [BK11]. In general, the better the lower bound algorithm used as a subroutine, the better the obtained lower bounds, but more time is spent for obtaining a result. The same holds for the LBP and LBN algorithm: The LBP algorithm gives better results, but it is also slower [BK11].

If the graph is very dense, then all of these algorithms give good results. One class of graphs where this is not the case is the class of planar graphs. Bodlaender et al. [BGK05] presented a lower bound algorithm which is based on the concept of brambles that works well on planar graphs.

Definition 3.1 ([BGK05])**.** *Let $G=(V,E)$ be a graph and $W_1, W_2 \subseteq V$. W_1 and W_2 are touching if they have a vertex in common or if there is an edge which connects them. A set $\mathcal{B}$ is called a bramble if it contains mutually touching connected vertex sets. A hitting set $H \subseteq V$ for $\mathcal{B}$ is a set of vertices such that for all $W \in \mathcal{B}$ it holds that $W \cap H \neq \emptyset$. The order of a bramble $\mathcal{B}$ is the minimum size of a hitting set for $\mathcal{B}$. The bramble number of G is the maximum order of all brambles of G.*

The following theorem shows that the bramble number corresponds to the treewidth (plus one).

Theorem 3.1 ([BGK05, ST93] for a proof)**.** *Let k be a non-negative integer. A graph has treewidth k if and only if it has bramble number $k+1$.*

This implies that if a bramble of order k is found, then the treewidth of the graph is at least $k-1$. Bodlaender et al. present [BGK05] two lower bound algorithms that are based on this: the first can be applied to general graphs, whereas the second is for planar graphs. Both algorithms perform well for planar and nearly-planar graphs, compared to the contraction degeneracy of a graph [BGK05].

Bodlaender and Koster [BK11] compare the algorithms MMD, MCS lower bound and LBN together with MMD+ as a subroutine. Each of the algorithms is considered once without contraction and once with it. In general, the algorithms with contraction outperform these without contraction. The LBN+(MMD+) algorithm, i.e. the LBN algorithm with contraction and MMD+ used as a subroutine, gives the best lower bounds but may take considerably more cpu time. MMD and MMD+ are the fastest algorithms. MMD+ gives

like the other contraction algorithms good results. Bodlaender and Koster propose to use MMD+ for very large graphs if the other contraction algorithms are too slow. However, when the graph is planar, one should use one of the two brambles algorithms because the degree-based methods give worse results [BK11].

3.3 Upper bound algorithms for treewidth

In this section treewidth upper bound algorithms are presented. Note that any approximation algorithm also gives an upper bound on the treewidth. Most of the results presented here can be found in further detail also in Bodlaender and Koster [BK10]. At first, the concepts of chordal graphs and elimination orderings and their connection to tree decompositions are introduced. Those concepts form the foundation for the upper bound algorithms based on elimination orderings. Then, the two kinds of elimination ordering construction methods are explained: chordal graph recognition heuristics and greedy degree heuristics. At the end of this section, the results of an experimental comparison of the upper bound algorithms by Bodlaender and Koster [BK10] are presented.
One approach for computing tree decompositions which is not discussed here in further detail relies on finding small separators in the graph. Recall that a separator is a set of vertices $S \subseteq V$ which separates the graph G, i.e. $G[V \backslash S]$ has more than one connected components. This approach works as follows: it recursively decomposes the graph by finding small separators and then uses these to construct a tree decomposition. However, this approach is slow and may result in bounds which are higher than those obtained by simpler methods [BK10]. Thus, it will not be further considered here.

3.3.1 Chordal graphs and elimination orderings

We begin with a few necessary definitions.

Definition 3.2 ([BK10])**.** *A graph $G = (V, E)$ is chordal if every cycle in G with length at least four has a chord, that is an edge which connects two non-successive vertices in the cycle. An elimination ordering of a graph $G = (V, E)$ is a bijection $f : V \to \{1, 2, ..., n\}$. An elimination ordering f is perfect if for all $v \in V$ the set of its higher numbered neighbours $\{w \mid \{v, w\} \in E \wedge f(w) > f(v)\}$ forms a clique. A triangulation of a graph G is a chordal graph that is obtained by adding zero or more edges to G. A triangulation H is a minimal triangulation of G if no proper subgraph of H is also a triangulation.*

The following theorem shows the equivalence between a graph that is chordal, a graph that has a perfect elimination ordering and a tree decomposition where every bag forms a clique in the graph.

Theorem 3.2 ([BK10])**.** *Let $G = (V, E)$ be a graph. The following statements are equivalent:*

i. G is chordal.

ii. G has a perfect elimination ordering.

iii. There is a tree decomposition (T, X) of G such that every bag X_t of $t \in T$ forms a clique in G

Note that this theorem may not be used to obtain an upper bound on the treewidth of a graph. Algorithm 3.1 Fill [BK10, BBHP04], presented below, takes as input a graph G and an elimation ordering π and adds edges to G such that the returned graph H is chordal. The returned graph H is also called the filled graph of G and π, denoted also as G_π^+. Since H has a perfect elimination ordering (specifically, the ordering π used for its construction), by Theorem 3.2 H is also chordal and thus a triangulation of G [BK10].

Algorithm 3.1: Fill [BK10]

```
   Input: A graph G, elimination ordering π
   Output: A chordal graph H
1  begin
2      H = G
3      for i = 1 to n do
4          Let v = π^{-1}(i), i.e. the ith vertex in the ordering π
5          for each pair of neighbours x,y of v s.t. x ≠ y and π(x) > π(v) and
            π(y) > π(v) do
6              if x and y are not adjacent then
7                  Add edge {x, y} to H
8  return H
```

The following theorem by Bodlaender and Koster [BK10] is a consequence of Theorem 3.2 and forms the bases for the upper bound algorithms. It gives an alternative characterisation of the treewidth.

Theorem 3.3 ([BK10])**.** *Let $G = (V, E)$ be a graph and let $k \leq n$ be a non-negative integer. The following statements are equivalent.*

i. G has treewidth at most k.

ii. G has a triangulation H such that the maximum size of a clique in H is at most $k + 1$.

iii. There is an elimination ordering π such that the maximum size of a clique of the filled graph G_π^+ is at most $k + 1$.

iv. There is an elimination ordering π such that no vertex $v \in V$ has more than k neighbours with a higher number in π in G_π^+.

Statement *iv.* of Theorem 3.3 can be used for obtaining upper bounds on the treewidth. By constructing some elimination order π of a graph G, one can check what is the maximal number of neighbours with a higher number in G_π^+. This number then provides an upper bound for the treewidth, without the need to construct a tree decomposition [BK10]. Even though Bodlaender and Koster [BK10] show that given G and π, the tree decomposition of G_π^+ can be constructed without building the graph G_π^+ first, here we will focus our attention merely on obtaining upper bounds for treewidth (without considering the construction of tree decompositions). The upper bound algorithms that are based on elimination orderings can be classified into two kinds of algorithms: chordal graph recognition algorithms and greedy triangulation algorithms.

3.3.2 Chordal graph recognition algorithms

The MCS algorithm by Tarjan and Yannakakis [TY84] was first introduced as a chordal graph recognition algorithm. The constructed vertex ordering (see Section 3.2) can be used for obtaining an upper bound. If the underlying graph is chordal, MCS guarantees to yield a perfect elimination ordering. When the goal is to recognize chordal graphs, the selection of the starting vertex of the MCS does not influence the result. However, in order to obtain tree decompositions of low width, choosing the first vertex may have a high impact. Therefore, one may run the algorithm with all possible vertices as the starting vertex with an added complexity of $O(n)$. An adaptation to the MCS algorithm is the MCS-Min algorithm that produces an ordering such that the filled graph is a minimal triangulation [BK10].
Another algorithm by Rose et al. [RTL76] is based on breadth-first search using a lexicographic ordering scheme. It is often called Lexicographic Breadth First Search (Lex-BFS) and can also recognize triangulated graphs [BK10].

3.3.3 Greedy triangulation algorithms

Greedy triangulation algorithms produce a vertex elimination ordering by considering a certain heuristic to choose the next vertex. If in the i-th step some vertex v is chosen by a heuristic, it is then eliminated from the graph. In the elimination ordering π the vertex is then assigned the number $\pi(v) = i$. This process is repeated until all vertices are eliminated from the graph [BK10]. An upper bound for the treewidth is then the maximum of the vertex degrees at the time of their elimination [CCMN03].
The most simple heuristic is the Greedy-Degree heuristic [BK10]. It chooses the vertex with lowest degree in the current graph. Another heuristic is the Greedy-Fill-In heuristic that chooses the vertex that has the smallest number of non-adjacent neighbour pairs, where the idea is to generate as few edges as possible [BK10]. Those heuristics are compared by Bodlaender and Koster [BK10], see also Section 3.3.4. A different heuristic was presented by Clautiaux et al. [CCMN03]: Before choosing the next vertex in the elimination ordering, for each vertex v a lower bound on the treewidth of the graph obtained by eliminating v is computed. Then, the vertex is chosen that has the lowest sum of two times the lower bound plus the degree of the vertex in the current graph [BK10].

3.3.4 Evaluation of upper bound heuristics

The following performance results of upper bound heuristics were obtained in an experimental evaluation by Bodlaender and Koster [BK10]. There, the authors compared two greedy based heuristics, Greedy-Degree and Greedy-Fill-In, and the two chordal graph recognition heuristics discussed above, specifically MCS and Lexicographic Breadth First Search. We summarize their results below.
Usually graphs from applications are used as inputs, but for these graphs it is hard to obtain the exact treewidth. As it is important for the analysis of upper bound heuristics to know the exact treewidth of a graph, the algorithms were run on randomly generated partial-k-trees instead. A k-tree is a triangulation of a graph G such that all maximal cliques are of size $k+1$. Note that by Theorem 3.3 this means that G has treewidth at most k. If a graph is a subgraph of a k-tree, it is a partial-k-tree. When creating a partial-k-tree, one first creates a k-tree with the specified number of vertices n and the size of the maximal clique k and then deletes p-percentage of the edges. 50 instances of partial-k-trees for each of the combinations of the parameters of $n \in \{100, 200, 500, 1000\}, k \in \{10, 20\}$ and $p \in \{30, 40, 50\}$ were constructed. The contraction degeneracy lower bound is computed with the MMD+ algorithm (see Section 3.2) for ensuring that the treewidth of the partial-k-tree is at least k [BK10]. This graph creation method was taken from the evaluation of the Quicktree algorithm [SG97].
Both the Greedy-Degree and the Greedy-Fill-In method obtain results that are close to the optimum. For $k = 20$ the upper bound averages of 50 instances obtained by the Greedy-Fill-In heuristic lie between 20.50 and 23.50, by the Greedy-Degree heuristic in between 20.00 and 22.50, so the Greedy-Degree heuristic obtains better upper bounds than the Greedy-Fill-In heuristic. If an additional post-processing step is added to both heuristics, then Greedy-Fill-In obtains better results. A combination of both heuristics, taking the sum of both heuristic values, performs slightly better than the Greedy-Degree heuristic for $k = 20$. Regarding computation time, Greedy-Degree is much faster than the other methods [BK10].
When comparing the chordal graph recognition heuristics on partial-k-trees (here for $k = 10$ only), MCS outperforms the Lexicographic Breadth First Search. Surprisingly, the greedy heuristics obtain much better results: the best widths obtained by the MCS is 15. An explanation for the bad performance of MCS is that even when only a small amount of edges is removed, none of the start vertices produce an elimination ordering with the optimal treewidth [BK10].

3.4 Tree-depth computation

The computation of the tree-depth of a graph is, similarly as for treewidth, NP-complete [Pot88]. On the other hand, given a constant t and a graph G, it is possible to answer in linear time whether the tree-depth of G is at most t [NdM12, Chapter 6]. However, since we focus on large graphs with possibly large tree-depth values, we will concentrate on lower and upper bounds instead of using exact methods (like in the case of treewidth).

Recall that every depth-first search tree of a connected graph G is an elimination tree for G [NdM12, Chapter 6]. This implies that any depth-first search tree provides a simple and efficiently computable upper bound on the tree-depth of the graph.
Regarding lower bounds, let us first consider the tree-depth of a path graph P_n of length n [NdM12, Chapter 6]:

$$td(P_n) = \lceil log_2(n+1) \rceil$$

This implies that if a path of length n exists in a graph G, the tree-depth of G is at least $\lceil log_2(n+1) \rceil$. This provides a fairly crude but easy to compute lower bound on the tree-depth. A better lower bound can be obtained from the following result by Bodlaender et al. [BGHK95] which connects treewidth and tree-depth:

$$tw(G) \leq td(G) - 1$$

This result allows us to translate treewidth lower-bounds to tree-depth lower bounds.

3.5 Structural parameters for ILP and MILP

Even though LP instances are solvable in polynomial time (see for example the polynomial time algorithm by Karmarkar [Kar84]), the integrality restriction results in NP-completeness for ILP (a proof for the NP-membership of ILP FEASIBILITY by Papadimitriou can be found in [Pap81], for the NP-hardness by Karp [Kar72]). A polynomial time algorithm for the problem of ILP FEASIBILITY is therefore unlikely to exist. However, if the number of variables is fixed, ILP FEASIBILITY can be solved in polynomial time [Len83].
Considering structural parameters has yielded many positive results for the standard NP-complete problem Boolean satisfiability (SAT). With the goal to construct efficient fixed-parameter tractable algorithms for SAT, one considers a graphical representation like the primal or incidence graph of the problem instance. As SAT is fixed-parameter tractable by different structural parameters like the treewidth [Sze04], applying structural parametrizations to ILP may yield similar results [GO16]. For the time being, relatively few results are known for ILP and ILP FEASIBILITY. Jansen and Kratsch [JK15] study how subsystems of the constraint matrix, that either are totally unimodular or have bounded treewidth, may be used to obtain kernels for the ILP FEASIBILITY problem. They also show that ILP FEASIBILITY is fixed-parameter tractable when paramaterized by the treewidth of the primal graph and the domain size of the variables, more exactly with runtime $O(d^{k+1}k\ poly(n))$, where d is the domain size of the variables and k the width of a tree decomposition of the primal graph.
Ganian and Ordyniak [GO16] show that parameterizing by the treewidth of the extended primal graph and the maximum absolute value of a coefficient does not result in fixed-parameter tractability (assuming that FPT $\neq$ paraNP): ILP FEASIBILITY is NP-hard when the treewidth of the extended primal graph is at most three and the maximum absolute value of a coefficient does not exceed two. Those hardness results also carry over to the more general parameter clique-width [CMR00]. On the other hand, ILP is

fixed-parameter tractable parameterized by the tree-depth of the extended primal graph of the constraint matrix A and the maximum absolute value of a coefficient occurring in the constraint matrix A or the vector b of the ILP instance. When parameterized only by the tree-depth, they show that already ILP FEASIBILITY is $W[1]$-hard [GO16].
In their follow-up work, Ganian et al. [GOR17] consider the incidence graph of the constraint matrix of an ILP instance. In comparison to the primal graph, the incidence graph provides information in which constraints a variable appears in. This information is used to construct an efficient bottom-up dynamic programming algorithm on the tree decomposition. More exactly, they show that ILP is solvable in time $O(\gamma^{2tw(H_I)+2}tw(H_I)(n+m))$, provided that a tree decomposition of minimal width $tw(H_I)$ of the incidence graph H_I is given. n and m are the number of variables and constraints, γ is the maximum absolute value that can be obtained from a constraint by summing the left-hand side of a constraint over a variable assignment up. γ may thus only be defined when the domains of the variables are bounded. A consequence of this result is that on non-negative ILP instances, i.e. instances where neither the coefficients in the constraint matrix A nor the domain values of the variables are negative, ILP can be solved in time $O((B_{max})^{tw(H_I)+1}(n+m))$ when a tree decomposition of minimal width $tw(H_I)$ of the incidence graph H_I is given [GOR17]. B_{max} is here the maximum absolute value in the vector b. In addition, they obtain hardness results for ILP when parameterized by the treewidth and either the maximum absolute value of the coefficients of the constraint matrix or the maximum of the domain values of all variables. For more details see [GOR17].

Unfortunately, the fixed-parameter tractable algorithms for integer linear programming mentioned above do not generalize to MILP. Ganian and Ordyniak [GOR17] introduce the structural parameter torso-width which is specifically constructed for MILP instances. The idea underlying torso-width is to focus on obtaining a decomposition for the parts of the graph representation that will be handled by dynamic programming, namely the integer variables which have bounded domain size. Outside of the torso, no assumption about the structure may be made. The following result is obtained by applying a dynamic programming algorithm on the tree decomposition of a q-torso:

Theorem 3.4 ([GOR17])**.** *Let q be a fixed integer and I be an input MILP instance. Then I is fixed-parameter tractable parameterized by q-torso-width.*

These results show that structural parameters may help solve ILP and MILP. Furthermore, it would be interesting to know how the difficulty of ILP or MILP instances reflects in structural parameters like the treewidth or the tree-depth. For industrial SAT instances, the treewidth parameter is not a recommendable metric for practical hardness as given by their solving time [Mat11], though for a more specific problem, the analysis of feature models for validity based on SAT, it is a good indicator for the hardness of the analysis run [PSP13]. A more positive result is obtained for CSP: Béjar [BFM05] predicts the runtime of CSP solving algorithms by analyzing parameters, with the result that the lower and upper bounds on the treewidth of the constraint graph gave some of the better

results in this respect. To the best of our knowledge, there has been no analysis of the correlation of structural parameters and the solving time of state-of-the-art ILP solvers until now.

CHAPTER 4

Methodology

In this chapter we describe the methods that were used to construct the framework MILP-Struct for computing structural parameters of graphical representations of ILP and MILP instances. MILP-Struct builds on the two libraries MIPLIB [KAA+11] and LibTW [vDvdHS06], which are described in the first two sections. The MIPLIB library is a collection of practical MILP instances. In Chapter 5, the results of applying MILP-Struct on the MIPLIB benchmark and challenge instances are presented. LibTW provides the algorithms for computing lower and upper bounds for treewidth and builds the foundation for MILP-Struct. The main contribution of the thesis, the MILP-Struct framework, is described extensively in Section 4.3.

4.1 MIPLIB

Before the implementation of the framework MILP-Struct, it is feasible to first search for a suitable benchmark set of ILP and MILP instances. We are interested in instances that are used in practical applications and that are not constructed in an artificial setting. In practical applications, one might first measure structural parameters to determine the difficulty of solving the ILP or MILP instance and then solve the instance if it has small structural parameters. Therefore, it is important to use instances from practice.
Even though there exist many different benchmark sets for SAT (for example the benchmark set of the yearly SAT competition [Bal16]) and CSP [csp99], in the case of ILP it is hard to find publicly available benchmark instances of practical nature. For this reason, instead of using a benchmark set of ILP instances, the MIPLIB library [mipb, KAA+11] is used. It is a collection of both mixed and normal integer programming instances. First created in 1992, it tried to meet the needs of researchers for a collection of mixed and integer programming instances. Over the years, many researchers contributed to its growth and topicality, the latest update of 2010 also in collaboration with academia and industry [mipb]. MIPLIB 2010, the fifth version, is an assortment of real-world mixed

integer programming instances fitted for benchmarking and testing of MILP solvers and solution algorithms [KAA+11].
It consists of 361 instances in total. These are sorted into different test sets to fulfill researchers' particular studies. The benchmark and challenge set is classified by the runtime of the instances. The benchmark set consists of instances where at least one solver can obtain an optimal solution within two hours. The challenge set contains instances that have not been solved to optimality, some of which may also be infeasible. Other test sets include the infeasible or the XXL set, that contains instances with a large number of variables or constraints. At the release of MIPLIB 2010, the 361 instances were categorized into 185 easy, 42 hard and 134 open instances. The runtime of a solver on a modern PC was taken to classify an instance. If an instance can be solved within one hour, the instance is classified as easy. If it can be solved but not within this time limit or requires special algorithms, it is classified as hard. If the instance cannot be solved to optimality, either because the optimality for a feasible solution cannot be proven or it is still unclear whether the instance is infeasible, it is considered as open [KAA+11]. During the last seven years, many instances switched from hard to easy and from open to hard or easy. Only 75 instances remain which are not solved, 54 instances that are classified as hard and the remaining 232 instances are now easy [mipb].
The size of the instances ranges from 10^1 to 10^7 columns and rows – the larger instances usually with a ratio of non-zero elements to zero-elements (the density of the constraint matrix) below 0.05. The average density is 1.6%, so also for smaller instances most of the values in the constraint matrix are zero. Even though instances usually vary in their size and density, they show a specific structure [KAA+11]. On the homepage of MIPLIB [mipb], for every instance a sparsity pattern of the structure of the instance can be found. The sparsity patterns show the positions of non-zero coefficients in the constraint matrix. In Figure 4.1 the sparsity patterns taken from the MIPLIB homepage of the two instances glass4 and ns1830653 are shown.

The problems submitted in different formats were all translated to a unique format, the MPS format that is also used in industry [KAA+11]. In contrast to other existing formats, it is column based. This means that the format specifies information about the individual variables and not about the individual constraints. Figure 4.2 below provides an example of a MILP problem formulated in the MPS file format and its corresponding mathematical representation.

The MPS file format [mipa] is structured as follows. Each file is separated into six fields, that start and end at certain specified columns (in Figure 4.2a only five are visible whereas keywords like 'NAME' or 'ROWS' start in the first column which is not counted as a field). At the top of the file, the name of the instance is stated. Then, the 'ROWS' definitions follow. Here, 'N' denotes the objective function. However, the MPS file format cannot specify whether the objective function needs to be maximized or minimized. 'G', 'L' or 'E' denotes respectively a constraint with a greater or equal, less or equal, or equal symbol. The type of the constraint is followed by its name. After the 'COLUMNS' line, the coefficients of the variables occurring in a certain constraint are described. The first 'COLUMNS' line describes that the variable 'X0' (field two) occurs with coefficient seven

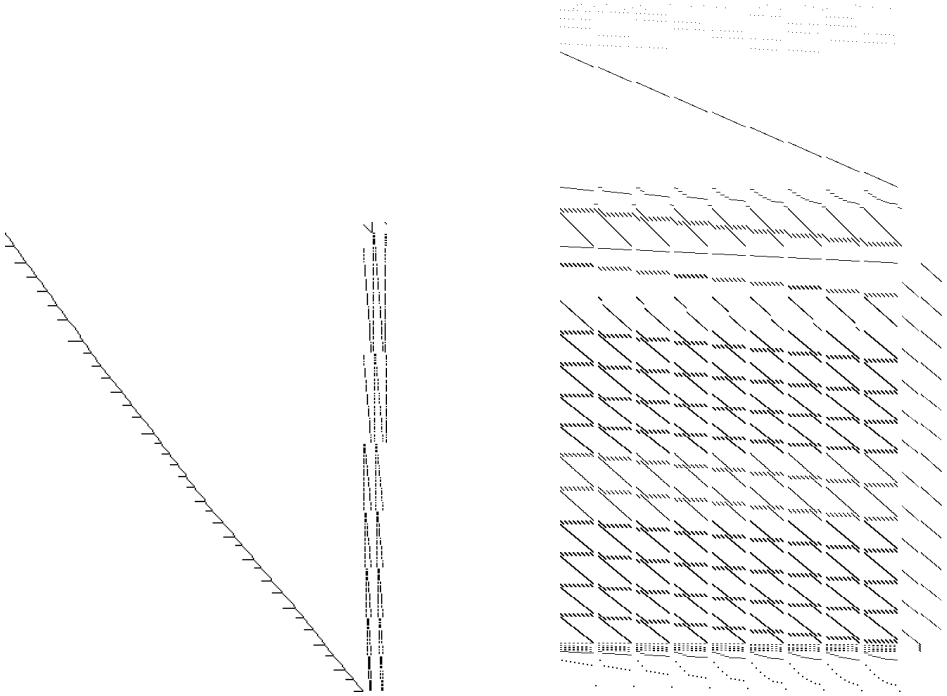


Figure 4.1: The sparsity patterns of the glass4 and ns1830653 instances. The sparsity patterns show the location of non-zero coefficients in the constraint matrix. The sparsities in these representations are amplified, so they are drawn more densely than they truly are [KAA+11].

(field four) in the objective function (field three). The 'INTORG' marker in line eleven denotes the beginning of integer variables. All following variables are integer variables until a line containing the 'INTEND' marker follows. The 'RHS' line denotes the end of the 'COLUMNS' part and defines the value on the right-hand side of a constraint. In a 'RHS' or 'COLUMNS' line, it is possible to use the last two fields to combine two individual lines. The optional 'BOUNDS' part may declare different bound types, for example 'LB', 'UB' or 'BV'. In the case of 'BV', the variable is defined to be binary. More bound types like fixed variable or free variable are possible, depending on the MILP solver used. The 'ENDATA' is the mandatory last line of the file [mipa].

The main problem of the MPS format is that there is no standard specification. Therefore, a MPS file can be interpreted by solvers in different ways. The MPS files in the MIPLIB library also do not follow one clear pattern of defining the problem instance. Especially the 'BOUNDS' section contains sometimes unspecified bound types. The MPS files also differ with respect to the start and end columns of the fields. These differences complicate the parsing of the MILP instances. In addition, a row-based format would be more compact because the sometimes long variable names would not be repeated as often. This may play an important role when parsing the MPS files since a smaller file generally implies less time and memory needed.

```
NAME          ExampleMILP
ROWS
 N  cost
 E  row0
 L  row1
COLUMNS
    X0        cost       7
    X0        row0       1
    X1        cost       3
    X1        row0       1
    MARK      'MARKER'                 'INTORG'
    Y0        cost       149
    Y0        row1       -11
    Y1        cost       221
    Y1        row1       12
    MARKEND   'MARKER'                 'INTEND'
RHS
    RHS       row0       22
    RHS       row1       24
BOUNDS
 BV bnd       Y0
 BV bnd       Y1
ENDATA
```

(a) A MILP instance formulated in the MPS file format

$$
\begin{array}{lll}
\{\min, \max\} & cost: & 7x_0 + 3x_1 + 149y_0 + 221y_1 \\
\text{s.t.} & row0: & x_0 + x_1 = 22 \\
 & row1: & -11y_0 + 12y_1 \leq 24 \\
 & & y_0, y_1 \in \{0, 1\}
\end{array}
$$

(b) The same MILP instance in a mathematical representation

Figure 4.2: Example of a MILP instance in the MPS file format

4.2 LibTW as a library for treewidth

Since one of our goals (and, in some sense, the main goal) of MILP-Struct was to compute the treewidth of graph representations of ILP and MILP instances, one of the existing libraries described in Table 3.1 was used rather than implementing algorithms for computing treewidth from scratch. As such a library had to be included in our framework, the library also has had an influence on the programming language of MILP-Struct. The requirements for such a library were the following:

- It needs to be possible to compute lower and upper bounds for treewidth. As the instances in MIPLIB have up to 10^7 rows or columns in the constraint matrix [KAA+11], one cannot hope to compute the exact treewidth. Lower and upper bound algorithms for treewidth only provide heuristics, so it would be useful to have

a library that supports different algorithms to try out some of these implementations on an instance.

- Another important aspect is ease of use: understanding and maybe extending the library is essential for integrating it into a larger framework. Especially a good documentation is of utmost importance for using the library in a correct and efficient manner.
- Most importantly, the software license needs to allow the usage and linking of the library. In addition, a software license that allows the modification and distribution is needed for publishing the library or in the case that smaller changes need to be made.

Out of the six libraries in Table 3.1, only QuickBB, LibTW and Toto provide means for computing lower and upper bounds for treewidth. Of these three, LibTW is the only library that is open-source and provides good documentation of how to use it in a Java-based application. In addition, LibTW provides several lower and upper bound algorithms for treewidth. Therefore, the decision was made to use LibTW as a library for computing bounds on treewidth.

In Section 3.2 and 3.3 state-of-the-art methods for computing lower and upper bounds for treewidth are presented. Out of these, the following lower bound algorithms are implemented in LibTW [vDvdHS06]:

- Min-Degree, the degree of the vertex that has the least degree
- Ramachandramurthi, that computes $\gamma(G)$
- MCS, a specific order in which vertices are visited
- MMD, based on vertex degrees and deleting vertices
- MMD+, based on vertex degrees and contracting vertices

Some of these algorithms do not state explicitly which vertex is chosen to be handled next in case of a draw [vDvdHS06]. For computational reasons, LibTW does not offer the possibility to branch on all such vertices, but instead offers the variant All-Start that branches only on the first vertex. For the MMD+ algorithm, LibTW offers the strategies least-c, max-d and min-d. Altogether, there are twelve lower bound algorithms [vDvdHS06]:

- Min-Degree
- Ramachandramurthi
- MCS

- All-Start MCS
- MMD
- All-Start MMD
- MMD+(least-c)
- MMD+(min-d)
- MMD+(max-d)
- All-Start MMD+(least-c)
- MinorMinWidth
- All-Start MinorMinWidth

The evaluation of [vDvdHS06] shows that whereas the runtimes of the algorithms are quite similar, some of these algorithms produce better lower bounds than others. The MMD+(least-c), the MinorMinWidth (a variant of the MMD+(min-d) algorithm) and their All-Start variants produce the highest lower bounds. Using one of those algorithms is therefore recommended. This result is analogous to the evaluation from [BKW04].

The following five upper bound algorithms are implemented in LibTW [vDvdHS06]:

- Greedy-Degree
- Greedy-Fill-In
- Lex-BFS
- MCS
- MCS-Min

Some of these algorithms do not return an upper bound for treewidth directly but only compute the permutation that allows to construct the tree decomposition. LibTW can then transform this permutation to a tree decomposition and thus obtain an upper bound for treewidth [vDvdHS]. Greedy-Degree and Greedy-Fill-In always compute an upper bound at least as low as one of the other three algorithms, thus choosing one of those two algorithms is recommended [vDvdHS06] (see also Section 3.3.4 for performance comparisons).

The main graph structure of LibTW is `NGraph`, that is a generic type like most of the classes in LibTW. It contains as type parameter the same type parameter as the vertices `NVertex` that are contained in the graph. Usually, the type argument is `InputData`, which is also the type that the algorithms are expecting as type. `NGraph` does not

contain any references to the edges in the graph. Instead, every `NVertex` stores a list of neighbour vertices. This implies that when an edge is added or deleted, the list of neighbour relations must be changed in both vertices.
There are different possibilities of how a `NGraph` can be constructed: it can either be read from an input file or one of the graph generator classes can be used. When the graph is read from an input file, only the DGF file format is supported [vDvdHS].

This is also one of the disadvantages of using LibTW: LibTW itself offers no possibility of generating a `NGraph` out of a graph instance that is not written in a DGF file. However, it is possible to implement the interface `GraphInput` to create its own version of generating a `NGraph`, which is the method of how a `NGraph` instance is obtained in MILP-Struct (see Section 4.3).
Another problem of LibTW is that it assumes that the input graphs are connected. In general, this is a reasonable assumption as publicly available graph instances usually are connected. In our case, we create the graph representations of ILP and MILP instances. Depending on whether the objective function is included in the construction of the graph representation, it may happen that the graph is disconnected. The treewidth of the graph is then equal to the maximum treewidth over its components [Bod98]. In such a case, the algorithms in LibTW may not necessarily consider this and may produce worse or, in general, even wrong treewidth bounds. For example, the Min-Degree lower bound returns the degree of the lowest-degree vertex. If a graph consists of two components, the Min-Degree algorithm, that does not consider that the graph is disconnected, returns as lower bound the degree of the lowest-degree vertex in the whole graph. A better lower bound could be obtained if the Min-Degree algorithm is first applied on the first component and then on the second, and then returns as lower bound of the whole graph the maximum of both obtained lower bounds.
The LibTW library was originally used to find out which algorithms produce the best results, with regard to lower and upper bounds and also with regard to the exact algorithms. The graphs that were used for comparing the algorithms were relatively small, i.e. at most 1,000 vertices, such that differences in the runtime were not observed [vDvdHS06]. The implementations are however not very fast for graphs of larger size, for example graphs with 50,000 vertices. This is however the size range that we need to consider when we want to compute bounds for the structural parameters of instances of the MIPLIB library. Some of the comments in the source code explain how the algorithm could have been implemented in a more efficient way. This shows that the implementations are not optimal with respect to the algorithmic runtime.
Even though LibTW provides many good lower and upper bound algorithms, it does not consider lower and upper bound methods that appeared during the last ten years, some of which are described in Section 3.2 and 3.3. Although these algorithms probably do not return much better bounds, it still might be interesting to learn how these perform.

4.3 MILP-Struct framework

In the following, we present the MILP-Struct framework which is able to compute structural parameters of graphical representations of ILP and MILP instances. The MILP-Struct framework implemented in Java provides functionality for parsing ILP and MILP instances, computing graph representations and obtaining lower and upper bounds for different structural parameters. This is the first framework of its kind, i.e. the first framework that combines ILP and MILP with structural parameters such as treewidth. Published under the LGPL license, it may be used and modified by other libraries. It is based upon the LibTW library for the computation of treewidth related properties. It also uses SLF4J bound to log4j [slf, log] for providing logging functionality and Wagu [wag] for presenting output in table format. Before explaining the tasks of specific components of the software system, the general structure of the framework is outlined.

4.3.1 Structure

MILP-Struct consists of different modules for starting the overall computation, handling the parsing and conversion of the linear program to the graph representations and finally computing structural parameters and general statistics of the graph representations and the linear program.

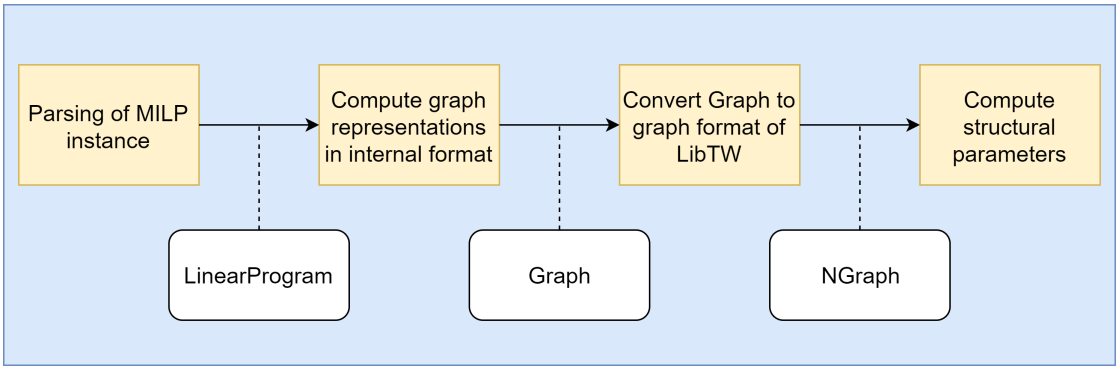


Figure 4.3: The program sequence of the computation of structural parameters for one MILP instance

The program sequence for one single MILP or ILP instance can be seen in Figure 4.3. It starts by reading the MILP instance in the MPS format from a file and generating a `LinearProgram` instance out of it. Note that in the complete computation there is no differentiation between a MILP instance or an ILP instance since both are stored in the same file format and converted also in the same manner. We thus use the term MILP instance for including both ILP and MILP instances. In this step, some statistics about the MILP instance, like the number of variables and constraints, are computed. Out of the `LinearProgram` instance at least one primal, incidence or dual graph representation is computed. The class that stores the graph representation is `Graph`, which is an internal

format of the MILP-Struct framework.
Then, the `Graph` instance for every graph representation is translated to the graph format of LibTW, the abstract class `NGraph`. In this step, the interface `GraphInput` of LibTW is implemented to create the `NGraph` from the `Graph` instance. Instead of first creating a `Graph` instance, we could have directly computed the graph representation as an instance of `NGraph`. But since the `Graph` class is more easy to handle than the abstract class `NGraph` and since the conversion of `Graph` to `NGraph` only takes a small fraction of the whole program runtime, it was decided against this option.
The `NGraph` representation with the computed graph statistics is then serialized to a predefined file. At every execution of MILP-Struct it is checked whether the graph representation was previously serialized. If that is the case, the graph representation is deserialized and does not need to be recomputed.
One or more structural parameters are then computed on the `NGraph` representations. For the primal graph representation, the computation of lower and upper bounds for treewidth and torso-width, and an upper bound for tree-depth are supported. For the incidence and dual graph representation, only the treewidth bounds can be computed. In addition to the structural parameters, statistics about the graph representations are computed. The results of the MILP instance are then converted to a text line in the CSV format.

The process of computing the structural parameters for one MILP instance is embedded in a larger sequence of steps of the overall framework. The execution sequence of a complete program run is represented in Figure 4.4. The program starts by parsing the program arguments and setting configurations for the run. One of those input arguments is the input file. This may either be a path to a text file or MPS file. If it is a text file, the text file is assumed to contain multiple file paths to MPS files. A list that contains these MPS files, or a single MPS file in the case that a MPS file is provided as the input file argument, is constructed.
For all of these MPS file paths, the structural parameters for each single MILP instance are computed as explained in Figure 4.3. The computation is started asynchronously but single-threaded using the Java `ExecutorService`. After a specified time limit, the computation may be interrupted in case it takes too long and the computation for the next MILP instance is started. After finishing the computation for the last MILP instance, the results are written to a CSV file.

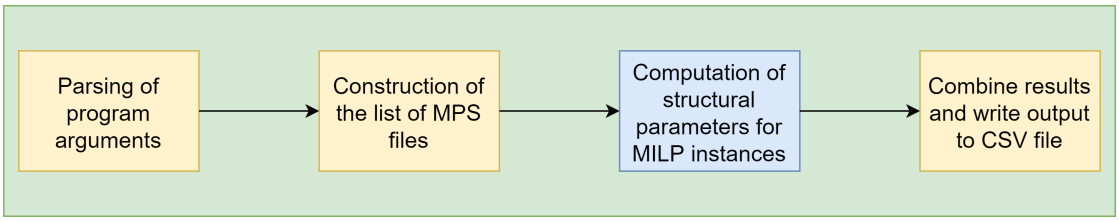


Figure 4.4: The program sequence for the computation of structural parameters of one MILP instance

4.3.2 Parsing of MPS files

MILP-Struct parses MILP instances in the MPS file format. However, as there does not exist one uniform MPS file format standard, the parser is made to support the MPS format from the MIPLIB library [mipa]. Even the instances within the MIPLIB library are not in the same format. Sometimes one of the fields in the 'COLUMNS', 'RHS' or 'BOUNDS' section is completely empty. The content is then just contained in the next field. In order to parse these differences, the six fields and their bounds are mostly ignored. Instead, every line of the MPS file is searched for white spaces and content. The content is then extracted by removing subsequent white spaces. For performance reasons, a regular expression is used for this. In this way, the contents of the fields are returned and empty fields are discarded. We can find out the kind of content, for example whether it is a variable or constraint name, by looking at the position of the contents.
The `LinearProgram` class stores the information that is needed for computing the three types of graph representations. This means that it stores the variables, the objective function and the constraints, so we can determine which constraints or objective function contains which variable (with a non-zero coefficient). The coefficients of the constraint matrix and the bound types for the variables are however not parsed with the consequence that less working memory is needed for the parsing.

4.3.3 Timeouts

The runtime for the computation of the structural parameters varies for the individual MILP instances. Whereas some instances are finished after seconds, others may run for hours without terminating. Hence it is not possible to estimate the runtime of the program in advance. In order to have a guarantee upon the runtime, MILP-Struct offers the possibility to specify a configurable time limit after which the current computation of a MILP-instance is set to be cancelled.
The Java `ExecutorService` is used for implementing this functionality. It provides methods for the creation of asynchronous tasks. These tasks may then be cancelled after a certain time limit. More specifically, the method `invokeAll(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)` is used. It starts the `tasks` asynchronously and interrupts the threads after the specified `timeout`. It has the return type `List<Future<T>>`, that can be used to extract the result of the computation. The `Collection` of tasks is in this case a single instance of `StructuralParametersComputation` wrapped in a list that implements the interface `Callable<String>`. The timeout is a configuration value provided in seconds.
The `invokeAll` method then executes the computation of structural parameters for one MILP instance and cancels the thread after the timeout occurs. However, the thread is not stopped directly. Instead, the thread's status is set to interrupted. By calling `Thread.currentThread().isInterrupted()` one can check whether the thread was interrupted. If it was interrupted, an `InterruptedException` is thrown. This exception is caught and the normal program flow continues, i.e. the computation for the next MILP instance begins.

This implies that `Thread.currentThread().isInterrupted()` needs to be checked regularly, otherwise the thread runs until it is checked the next time.
The returned list of `Future<T>` objects contains the status of the thread and the result of the computation as type `<T>`. By calling `Future.isCancelled()`, it can be distinguished whether the task was cancelled or completed. With `Future.get()` the type parameter `T` of the `Future` is returned. In our case, the type argument is a `String` that contains the statistics including the structural parameters in the CSV format.

4.3.4 Treewidth

The treewidth remains the most prominent structural parameter for many graph related problems. Because of its growing prominence during the last years, there exist many implementations for computing tree decompositions and treewidth. These libraries and tools were described in Section 3.1 and why LibTW was chosen as a library for computing treewidth lower and upper bounds in Section 4.2.
Even though the LibTW source code is available and allows for modifications, it is mainly treated as a black box for computing bounds for treewidth. However, some changes were necessary in order to improve the obtained results.
First, the LibTW library assumes that the input graph `NGraph` is connected. This, however, is not always the case when MILP instances from the MIPLIB library are parsed. There are two possibilities of how the objective function is handled when generating the graph representations. If the input argument `--obj` is specified, the extended versions of the graph representations are constructed. The objective function is then handled like any other constraint. Without this input argument the simplified graph representations are considered. Here it may happen that the resulting graph is not connected. For example, if a constraint only contains variables that occur in the objective function or in the constraint itself (with non-zero coefficients), the simplified dual graph representation would have a vertex without any neighbours.
The reason why both methods are supported in MILP-Struct is that in literature results for both variations of the graph representations exist [GO16, GOR17]. There is also a performance advantage when the objective function is not considered. For some instances, the objective function can be the "binding element" of the whole MILP instance. The simplified graph representations may then contain much fewer edges than the extended graph representations. It may thus be possible to obtain results and not run into a timeout – where the extended graph representations would fail.
In order to obtain better lower and upper bounds for disconnected graph, we can adapt the algorithms to graphs that are not connected. By Lemma 2.2, the treewidth of the graph is equal to the maximum treewidth of its connected components. Observation 4.1 applies this result for treewidth bounds.

Observation 4.1. *Let G be a graph.*

1. *Let k be the maximum of lower bounds for treewidth over all connected components of the graph G. Then the treewidth of G is at least k.*

2. *Let k be the maximum of upper bounds for treewidth over all connected components of the graph G. Then the treewidth of G is at most k.*

In the case of lower bounds for treewidth, this implies that a lower bound of any connected component is already a lower bound of the whole graph. On the other hand, an upper bound for treewidth of a connected component is not always an upper bound of the whole graph. An upper bound for treewidth of every connected component needs to be computed such that an upper bound of the overall graph can be determined. Figure 4.5 represents this property with a simple example.

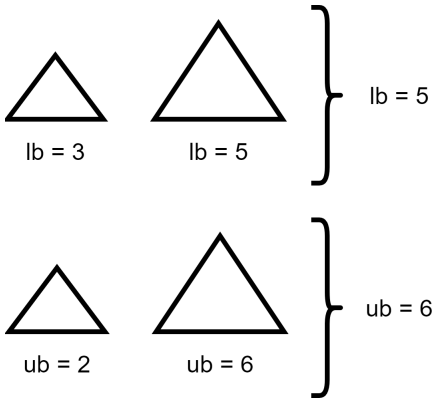


Figure 4.5: Treewidth bounds of connected components and the bound of the overall graph. In the first example, a lower bound for treewidth of three of the first connected component and of five of the second implies that five is a lower bound of the overall graph. For the upper bound, the larger upper bound six has to be taken as treewidth upper bound of the overall graph since two is not a valid upper bound if the treewidth of the second connected component is three or larger.

The data type `NGraph` of LibTW is adapted to graphs which are not connected. For this, `NGraph` stores its connected components in a list of `NGraph`s. It is then possible to apply the lower and upper bound algorithms of LibTW on each connected component and, in the case of lower bounds, obtain a better bound than without considering the connected components.
In the Algorithm 4.1 below, it is shown how the connected components are handled in MILP-Struct to obtain a lower bound of the overall graph. First, the lower bound is initialized to zero. Then, for every connected component of the graph a lower bound is computed by using a lower bound algorithm of LibTW. If the lower bound of the current component is larger than the current lower bound, it is taken as new lower bound of the overall graph. When computing the upper bound for treewidth, the only change is to initialize the upper bound to be the number of vertices of the graph (minus one). An upper bound algorithm is then used on every connected component. The upper bound for treewidth of the input graph is also the maximum over all the computed upper bounds of the connected components.

Algorithm 4.1: Treewidth lower bound with components

```
Input: A graph G
Output: A lower bound for tw(G)
1 begin
2     lowerBound = 0
3     for every connected component of the graph do
4         lowerBoundComponent = computeLowerBound(component)
5         if lowerBoundComponent > lowerBound then
6             lowerBound = lowerBoundComponent
7 return lowerBound
```

By default, the MMD+(least-c) algorithm is used for computing the lower bound for treewidth, and the Greedy-Degree algorithm is used as upper bound algorithm. Both algorithms were chosen because of their expected good results and runtimes. A small improvement was hereby added to the Greedy-Degree algorithm. Remember that the algorithm works by choosing the lowest-degree vertex to be the next vertex that is eliminated [BK10]. The upper bound is set to the largest number of neighbours of a lowest-degree vertex at the time of its elimination. This corresponds to the following upper bound update in the `GreedyDegree` class of LibTW:

```
upperBound = Math.max(upperBound, lowestDegreeVertex.getNumberOfNeighbors())
    ;
```

The `lowestDegreeVertex` is then eliminated. After the elimination of the vertex, the following code is added:

```
if (graph.getNumberOfVertices() <= upperBound) {
    return;
}
```

This allows the upper bound algorithm to terminate earlier without changing the result: if the graph contains at most n vertices and n is the current `upperBound`, then any vertex in the graph can have at most $n-1$ neighbours. Thus, the upper bound will not increase anymore. This change may improve the runtimes of certain graphs immensely. For example, consider the complete graph K_n. The best possible upper bound for treewidth is already obtained in the first iteration, i.e. the lowest-degree vertex is a vertex with $n-1$ neighbours. The vertex is then eliminated and the resulting graph has $n-1$ vertices and thus less or equal than the current upper bound. The algorithm then terminates in the first iteration with the early termination change, whereas it would do n iterations without it.
Another change in the `GreedyDegree` class is that in every iteration it is checked whether the current thread was interrupted. This is necessary as the Greedy-Degree algorithm can take much time for larger graph instances. The `run()` method was thus

changed to throw an `InterruptedException` in case that the thread is cancelled during the computation.

4.3.5 Tree-depth

For the computation of the tree-depth of the graph representations, we use the fact that any depth-first search (DFS) tree is an elimination tree of the graph [NdM12, Chapter 6]. Thus, the height of any DFS tree of the graph is an upper bound for tree-depth. The algorithm for computing an upper bound for tree-depth is based on this property – with some improvements to obtain better upper bounds. First, remember that the graph may not be connected if the objective function is not considered when building the graph representation of the MILP instance. Note that by computing a DFS tree, it is automatically detected whether the graph is not connected in the case that the DFS cannot find every vertex in the graph. Special attention should be paid to the selection of the first and every following vertex in the DFS tree as this influences the height of the resulting tree.

In Algorithm 4.2 the pseudo code of the algorithm used for computing an upper bound for tree-depth can be found. It starts with finding the start vertex, the first root vertex,

Algorithm 4.2: Tree-depth upper bound with components

```
   Input: A graph G
   Output: An upper bound for the tree-depth of G
1  begin
2      path = findMaximalPath(G)
3      startVertex = path.get(path.length() / 2)
4      verticesFound = ∅
5      maxHeight = 0
6      rootVertex = startVertex
7      while verticesFound.size() ≠ graph.size() do
8          height = constructDFSTree(rootVertex, verticesFound)
9          if height > maxHeight then
10             maxHeight = height
11         rootVertex = first vertex in G that is not contained in verticesFound
12     upperBound = maxHeight
13     return upperBound
```

of the DFS tree. This is accomplished by searching a random maximal path, i.e. first a random vertex is taken, and then random neighbours not yet in the path are added until the last chosen vertex has every neighbour already in the path. The start vertex is then taken to be the vertex in the middle of the path. This method is inspired by the property of tree-depth that if a path of length n exists in the graph, the tree-depth is at least $\lceil log_2(n+1) \rceil$. For obtaining an elimination tree of this height, the path of

length n is "folded" into the elimination tree, and the vertex in the middle of the path corresponds to the root vertex [NdM12, Chapter 6]. Note that in this step a lower bound for tree-depth is computed. However, the lower bound is usually very weak as it corresponds to the logarithm of the length of a path. For example, a path of length 1000 would just correspond to a lower bound of ten. As the tree-depth of a graph is always at least the treewidth of the graph plus one [BGHK95], the treewidth of the graph can thus be used as lower bound for tree-depth. This property does not appear explicitly in MILP-Struct, i.e. the tree-depth class provides no lower bound computation method.
After choosing the root vertex, DFS trees are repeatedly constructed until every vertex in the graph is found, i.e. the set of `verticesFound` contains every vertex in the graph. The method `constructDFSTree(rootVertex, verticesFound)` hereby builds up the DFS tree and stores the vertices that are found in the set `verticesFound`.
The height of the DFS tree is returned after the construction of the DFS tree. If this `height` is larger than the `maxHeight` found, the maximal height is updated to be the larger value. The next `rootVertex` is set to be a vertex that has not been found by any DFS tree, i.e. a vertex that is not contained in the set `verticesFound`. After every vertex of the graph has been found by a DFS tree (every connected component was handled), the upper bound `maxHeight` is returned.

When constructing the DFS tree, one may use different strategies of how to select the next vertex. At first, the strategy to select the next neighbour vertex returned by an `Iterator` was used. Two other possibilities are to select the vertex with the current lowest or largest degree. Choosing the lowest-degree vertex leads to worse results for the height of the DFS tree. An explanation is that choosing the lowest-degree vertex puts low-degree vertices higher up in the DFS tree (if the root is at the top), even though these vertices should occur at the bottom of the DFS tree. The path graph P_n is an example for this bad behaviour. This strategy would always select one of the two border vertices which results in a DFS tree with the largest height n – a major difference to the height of the minimum elimination tree $\lceil log_2(n+1) \rceil$.
The second selection strategy, choosing the vertex with the largest degree, results in DFS trees with smaller height. For star graphs, that are graphs that have exactly one vertex that is connected to every other vertex, this method produces a DFS tree of the smallest height two. Note that also for path graphs, choosing the largest-degree strategy results in a smaller height of the elimination tree than the lowest-degree strategy. Also in comparison to a random selection strategy, choosing the vertex with largest degree produces seemingly slightly better results, even though the difference is not as large as between the lowest and largest-degree vertex strategy. In MILP-Struct, the DFS tree is therefore constructed by always choosing the vertex with the largest degree as next vertex.

There is one last improvement added to the tree-depth upper bound computation in Algorithm 4.2. As the computation of a DFS tree is very fast, the algorithm is not only executed once. Instead, the algorithm is repeated for a specific number of times `NUM_DFS_TREE_GENERATION`. By default, this value is set to 100. The tree-depth

upper bound is then set to the minimum of the upper bounds returned. By changing the `NUM_DFS_TREE_GENERATION` field, we can influence the runtime and performance of the upper bound. If the number of iterations is increased, better results are returned in general (even though there is some randomness involved), but it takes more time to obtain the results. It would also be possible to adapt the number of iterations to the size of the instance. For example, the number of iterations could be set to a smaller value for large instances in order to obtain a faster algorithm.

4.3.6 Torso-width

In the context of mixed ILP instances, that contain both integer and non-integer variables, the torso-width is the only structural parameter for which results for the fixed-parameter tractability of MILP exist. The MILP-Struct framework can compute an upper bound for the ∞-torso-width. More specifically, it computes the ∞-torso T that is obtained by collapsing exactly the non-integer vertices in the primal graph of the MILP instance. Remember that the torso-width of the torso T is then exactly the treewidth of T and that it is an upper bound for the ∞-torso-width. The ∞-torso-width could be computed by taking the minimum of the torso-width over any torso that is obtained by collapsing at least all non-integer vertices. For computational reasons, we only compute one such torso, namely the torso that collapses exactly the non-integer vertices.

Regarding the computation of the torso T, that is obtained by collapsing exactly the non-integer vertices, we stick to its definition. Because collapsing a set of vertices X results in the same graph as eliminating every vertex in X, one easy approach for computing T is to eliminate every vertex in the set of non-integer vertices.
However, implementing the construction of the torso in this way is very inefficient with regard to time and space spent at the computation. Consider the example primal graph in Figure 4.6. Here the white vertices correspond to non-integer variables and the black vertices to integer variables in the MILP instance. In the example, in each step a vertex is eliminated according to its number. Since the result graph at the bottom does not contain any non-integer vertices, it corresponds to a ∞-torso. The first two elimination steps only produce edges which do not appear in the resulting graph. Altogether nine edges are added of which six are removed again during the elimination process. In total, there are only three edges that must be added to the graph because they also appear in the result graph. However, the intermediate steps add and remove a lot of unnecessary edges.
This small example shows that instead of eliminating one vertex after another in a graph G, we should rather "collect" the set of neighbour integer vertices I from a connected set of non-integer vertices N and then do two steps to obtain the graph $G \circ N$:

1. Delete the set N of the graph, i.e. obtain the graph $G[V(G) \setminus N]$
2. Make I form a clique in the graph $G[V(G) \setminus N]$

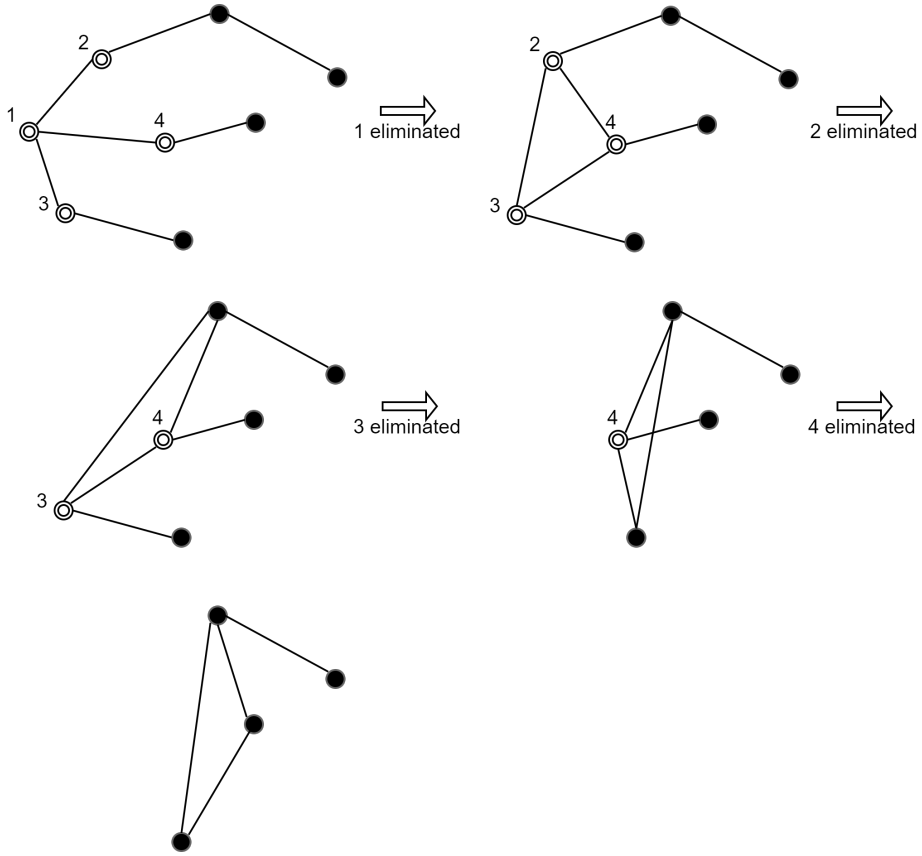


Figure 4.6: Primal graph of a MILP instance where the four non-integer vertices are eliminated

This procedure needs to be repeated until all connected sets of non-integer vertices have been handled. We apply this approach in the MILP-Struct framework for constructing the torso T. The pseudo code of the implementation is presented in Algorithm 4.3. The algorithm starts by checking whether there still exists a non-integer vertex not yet handled. If this is the case, this vertex is put into the `nodesToHandle` set, that only contains non-integer vertices where the `handled` flag was already set to `true`.
The two sets `curNonIntSet` and `curIntSet` are initialized in line five and six. The `curNonIntSet` is a connected set of non-integer vertices that needs to be collapsed. The `curIntSet` contains the integer vertices adjacent to the vertices in `curNonIntSet`. These two sets are filled within the `while` block of line seven.
While the `nodesToHandle` set is not empty, the first vertex `w` is removed from the set and added to the set `curNonIntSet`. Then, for every neighbour `n` of `w` it is checked whether the neighbour is an integer vertex. In that case, it is added to the set of current integer vertices. Otherwise, it must be a non-integer vertex, and it is checked whether the `handled` flag was already set. If it has not been handled, the `handled` flag is set to true and added to the list of vertices that need to be handled.

Algorithm 4.3: ∞-torso of G

Input: A graph G
Output: The ∞-torso of G that is obtained by collapsing the set of non-integer vertices

```
1  begin
2      while there exists a non-integer vertex v in G with v.handled = false do
3          v.handled = true
4          nodesToHandle = {v}
5          curIntSet = ∅
6          curNonIntSet = ∅
7          while nodesToHandle is not empty do
               /* Handle non-integer vertex w                              */
8              remove a vertex w from nodesToHandle
9              curNonIntSet = curNonIntSet ∪ {w}
10             for every neighbour n of w do
11                 if n is an integer vertex then
12                     curIntSet = curIntSet ∪ {w}
13                 else if n.handled = false then
14                     n.handled = true
15                     nodesToHandle = nodesToHandle ∪ {n}
16         delete the vertices in curNonIntSet from G
17         add edges to the vertices in curIntSet such that curIntSet forms a clique
```

After every vertex in the current part of connected non-integer vertices is handled, the vertices in `curNonIntSet` are deleted from the graph G, and the set `curIntSet` is made adjacent to form a clique. Note that the `handled` flag is set to `true` whenever a non-integer vertex is added to the set `nodesToHandle`. This has the advantage that while the vertex is in the set `nodesToHandle`, i.e. it has not yet been handled, the check for its `handled` flag in line 13 implies that it will not be tried to be added again to the `nodesToHandle` set. From a computational perspective, this implies that the vertex is not searched unnecessarily in the `nodesToHandle` set.
In the implementation of the algorithm, the vertices are not deleted directly from the graph G. They are only stored in a set which marks their deletion. The reason for this is that while iterating over the vertices of the graph G (in line two), vertices cannot be deleted from the graph. Therefore, the vertices marked for deletion are removed after every non-integer vertex has been handled.

As we want to compute an upper bound for the ∞-torso-width, the treewidth of the torso that is obtained by Algorithm 4.3 must be computed. The treewidth lower and upper bound algorithm from Section 4.3.4, which consider that the graph may be disconnected, are used for this. Note that the upper bound algorithm for treewidth then returns an

upper bound for the ∞-torso-width. Unfortunately, we cannot state such a property for the result of the treewidth lower bound algorithm.

CHAPTER 5

Results

This chapter presents the computed results of MILP-Struct on the instances from the MIPLIB library. After explaining the evaluation setup in Section 5.1, the results of applying our framework on the benchmark set of the MIPLIB library are presented in Section 5.2. In the last section, we analyze two specific instances from the challenge set of the MIPLIB library in more detail and evaluate the computed structural parameters with respect to the difficulty of the two instances.

5.1 Evaluation setup

All results were run on an Windows 10 Acer Aspire VN7-571G-77Q2. It has an Intel(R) Core(TM) i7-5500U 2.4 GHz Dual Core CPU and 8 GB physical memory. The Java version used is 1.8.0_101. The results in the following two sections were obtained by increasing the available memory of the JVM; the maximum heap size was set to 7 GB and the maximum stack size to 2 GB.

5.2 Benchmark set

In this section we present the results of MILP-Struct applied on the benchmark set of the MIPLIB library. The benchmark test set contains only instances that are solvable and classified as easy [mipb]. The results of applying MILP-Struct on the benchmark set are presented in Table 5.1. The extended graph representations are here considered; recall that this is a prerequisite for using some of the known algorithms [GO16].
The first four columns correspond to information about the linear program instance. The following five columns are about the extended primal graph representation, and the last four columns contain the computed bound values of the extended incidence and dual graph representation.

Table 5.1: Results for the benchmark set of the MIPLIB library

Linear Program				Extended Primal Graph					Ext. Inc. Graph		Ext. Dual Graph	
Name	Vars	Cons	Prop Int	LB TW	UB TW	TD	LB TO	UB TO	LB TW	UB TW	LB TW	UB TW
acc-tight5	1339	3052	1	154	980	1315	154	980	80	900	177	1313
aflow40b	2728	1442	0,5	1364	1364	2692	1363	1363	39	39	39	39
air04	8904	823	1		8903				163	620	202	625
app1-2	26871	53467	0,495	269	269	324			7	8		
ash608gpia-3col	3651	24748	1	94	502	2631	94	502	91	511		
bab5	21600	4964	1						91	325	95	350
beasleyC3	2500	1750	0,5	1249	1250	1793	1249	1249	12	34	12	33
biella1	7328	1203	0,8338		7327		6109	6109	114	471	156	400
bienst2	505	576	0,0693	54	310	399	34	34	26	85	59	112
binkar10_1	2298	1026	0,074	1741	1741	1742	169	169	11	15	11	13
bley_xl1	5831	175620	1	457		3193	457	2651	349			
bnatt350	3150	4923	1	33	152	250	33	152	33	149	243	1627
core2536-691	15293	2539	0,9994						228	1431	307	1388
cov1075	120	637	1	119	119	120	119	119	95	120	474	604
csched010	1758	351	0,8288	167	725	1502	1456	1456	50	72	56	72
danoint	521	664	0,1075	94	292	462	55	55	35	142	76	167
dfn-gwin-UUM	938	158	0,0959	103	669	885	89	89	27	90	28	79
eil33.2	4516	32	1	4515	4515	4516	4515	4515	32	32	32	32
eilB101	2818	100	1	2817	2817	2818	2817	2817	66	75	67	75
enlight13	338	169	1	168	168	170	168	168	11	41	15	37
enlight14	392	196	1	195	195	197	195	195	11	40	16	42
ex9	10404	40962	1						404		1912	
glass4	322	396	0,9379	19	19	26	301	301	18	18	133	283
gmu-35-40	1205	424	0,9959	1049	1049	1050	1049	1049	22	35	25	32
iis-100-0-cov	100	3831	1	99	99	100	99	99	80	92	1480	3766
iis-bupa-cov	345	4803	1	344	344	345	344	344	120	222	1954	
iis-pima-cov	768	7201	1	767	767	768	767	767	155	351	2938	
lectsched-4-obj	7901	14163	1	1049	1049	1191	1049	1049	174	231	1528	
m100n500k4r1	500	100	1	499	499	500	499	499	35	83	45	83
macrophage	2260	3164	1	1581	1581	1732	1581	1581	10	16	88	182
map18	164547	328818	0,0009	1675	1800		135	135	118			
map20	164547	328818	0,0009	1675	1800		135	135	119			

Table 5.1: Results for the benchmark set of the MIPLIB library

Linear Program				Extended Primal Graph					Ext. Inc. Graph		Ext. Dual Graph	
Name	Vars	Cons	Prop Int	LB TW	UB TW	TD	LB TO	UB TO	LB TW	UB TW	LB TW	UB TW
mcsched	1747	2107	0,9989	1576	1576	1577	1574	1574	27	98	50	143
mik.250-1-100.1	251	151	0,996	250	250	251	249	249	52	52	101	101
mine-90-10	900	6270	1	899	899	900	899	899	53	174	103	880
mine-166-5	830	8429	1	829	829	830	829	829	78	260	185	
msc98-ip	21143	15850	0,9597						105	683	604	
mspp16.mps												
mzzv11	10240	9499	1	335		8322			190	1465	387	
n3div36	22120	4484	1						60	60	60	60
n3seq24	119856	6044	1						1070			
n4-3	3596	1236	0,0484	3538	3538	3576	173	173	77	171	96	171
neos13	1827	20852	0,9934	1826	1826	1827	1814	1814	13	13		
neos18	3312	11402	1	84	107	815	84	107	51	69	353	1932
neos-476283	11915	10015	0,469						1669		3605	
neos-686190	3660	3664	1	3540	3540	3659	3540	3540	69	177	340	
neos-849702	1737	1041	1	200	1274	1541	200	1274	94	697	165	425
neos-916792	1474	1909	0,4864	756	756	990	716	716	266	582	368	582
neos-934278	23123	11495	0,863						209		264	1883
neos-1109824	1520	28979	1	1511	1511	1512	1511	1511	70	266	473	
neos-1337307	2840	5687	1	1846	1846	2817	1846	1846	75	131	124	187
neos-1396125	1161	1494	0,1111	257	416	823	92	92	42	195	83	288
neos-1601936	4446	3131	0,8785	527			3369	3369	178	1744	474	1468
net12	14115	14021	0,1136	621	1152	1664	868	1129	157	549	555	
netdiversion	129180	119589	1						118	1071	211	
newdano	505	576	0,1109	54	315	399	55	55	26	87	59	112
noswot	128	182	0,7812	24	24	46	24	24	10	19	17	24
ns1208400	2883	4289	0,999	472	2416	2844	502	2362	197	2135	413	1220
ns1688347	2685	4191	1	582	1554	2579	582	1554	225	502	278	753
ns1758913	17956	624166	0,9925	202	269	1394						
ns1766074	100	182	0,9	90	91	100	89	89	17	20	64	128
ns1830653	1629	2932	0,895	746	1197	1460	1295	1295	215	1035	654	1357
opm2-z7-s2	2023	31798	1	2022	2022	2023	2022	2022	151	1150	526	
pg5_34	2600	225	0,0385	2599	2599	2600	99	99	26	27	27	27

Table 5.1: Results for the benchmark set of the MIPLIB library

Linear Program				Extended Primal Graph					Ext. Inc. Graph		Ext. Dual Graph	
Name	Vars	Cons	Prop Int	LB TW	UB TW	TD	LB TO	UB TO	LB TW	UB TW	LB TW	UB TW
pigeon-10	490	931	0,8163	99	119	130	173	249	55	119	182	545
pw-myciel4	1059	8164	1	89	424	960	89	424	35	68	68	359
qiu	840	1192	0,0571	398	609	839	47	47	20	52	40	208
rail507	63019	509	0,9998						143	307	146	316
ran16x16	512	288	0,5	511	511	512	255	255	17	17	17	17
reblock67	670	2523	1	669	669	670	669	669	44	229	65	407
rmatr100-p5	8784	8685	0,0114		8783		99	99	88	100	494	
rmatr100-p10	7359	7260	0,0136		7358		99	99	79	100	414	
rmine6	1096	7078	1	1095	1095	1096	1095	1095	63	331	154	1276
rocII-4-11	9234	21738	0,984	202	573	1649			45	64		
rococoC10-001000	3117	1293	1	211	1646		211	1646	60	83	64	83
roll3000	1166	2295	0,6329	490	494	612	491	491	104	308	215	256
satellites1-25	9013	5996	0,9441	467	1020	7344	467	1020	151	794	267	1894
sp98ic	10894	825	1						135	156	135	156
sp98ir	1680	1531	1	1679	1679	1680	1679	1679	168	195	187	195
tanglegram1	34759	68342	1						18	20		
tanglegram2	4714	8980	1	4489	4489	4516	4489	4489	13	14	2596	
timtab1	397	171	0,4307	127	127	129	170	170	12	16	53	65
triptim1	30055	15706	0,9998			18757			346		515	
unitcal_7	25755	48939	0,1109			13452	2855	2855	73	343	123	1201
vpphard	51471	47280	1						148		477	
zib54-UUE	5150	1809	0,0157	259	988	3677	80	80	57	135	56	135

The linear program section contains information about the number of variables, the number of constraints and the proportion of integer variables to non-integer variables. The proportion is rounded to four decimal places. The columns 'LB TW' and 'UB TW' denote the lower and upper bound for treewidth of the corresponding graph representation. The columns 'LB TO' and 'UB TO' contain the lower and upper bounds for the torso-width of the ∞-torso that is obtained by collapsing the non-integer vertices in the primal graph representation. The column 'TD' contains the computed primal tree-depth upper bounds. If a table cell is empty, the time or memory limit was exceeded. The timeout configuration for each of the three graph representations was set to one hour. For each individual computation of the treewidth lower bound, treewidth upper bound and tree-depth upper bound, the timeout was set to 15 minutes. As the torso-width bounds are computed together, the timeout was set to 30 minutes for computing the torso and the treewidth lower and upper bound of the torso. By default, the number of iterations in the tree-depth upper bound algorithm is set to 100.
Note that the algorithms for computing lower or upper bounds are only heuristics. The use of efficient data structures like hash sets may produce different iteration orderings; two runs of MILP-Struct thus do not necessarily return the same bound values. MILP-Struct only guarantees that a correct bound value is returned. However, the computed bound values usually do not differ much.

The results presented in Table 5.1 are the raw data output computed by MILP-Struct. Some of these bounds may be improved by exploiting the relations between the structural parameters.

1. The incidence treewidth is at most the primal treewidth plus one (Theorem 2.1).
2. The incidence treewidth is at most the dual treewidth plus one (Theorem 2.1).
3. The primal treewidth is at most the primal tree-depth minus one (Section 3.4).
4. If the proportion of integer variables is one, the primal graph and the torso obtained by collapsing exactly the set of non-integer vertices are the same (Section 2.6).

In general, the incidence treewidth bounds are already much smaller than the primal treewidth bounds, especially for graphs where the bounds are quite high. Nevertheless, for a single instance, the ash608gpia-3col instance, we can exploit the relation between the primal and incidence treewidth. As the incidence treewidth upper bound is 511 and the primal treewidth upper bound is 502, the incidence treewidth can be at most 503.
In the same way, the incidence treewidth upper bound can be improved when the dual treewidth upper bound is larger. Here, it happens more often that the incidence treewidth upper bound is more than one larger than the dual treewidth upper bound. Some examples include the enlight13, ns1208400 and roll3000 instance, whereas the ns1208400 instance produces a notably much higher incidence treewidth upper bound than dual upper bound (2135 and 1220). Again, we can improve the incidence treewidth upper bound by taking the dual treewidth upper bound value plus one.

Regarding the treewidth lower bounds of the three graph representations, the incidence treewidth lower bound is always at most one larger than the primal or dual treewidth lower bound. Thus, it is not possible to improve one of the lower bounds. This may be a sign that the used lower bound algorithm produces tight lower bounds for the treewidth. The primal tree-depth upper bound of every instance in the benchmark set is larger than the primal treewidth upper bound. Sometimes, the tree-depth upper bound is even significantly larger than the treewidth or treewidth upper bound. Examples include the two instances aflow40b and ash608gpia-3col. In other cases, like the enlight13 or binkar10_1 instance, the tree-depth is very close to the treewidth.

The torso-width of the torso that is obtained by collapsing the set of non-integer vertices is often strongly dependent on the number of integer variables of the MILP instance. The computed torso-width lower and upper bound are usually equal to the number of integer variables minus one (which do not appear directly in Table 5.1 but can be computed approximately by the proportion of integer variables). This means that in the primal graph representation of the MILP instance no parts of the graph exist which are separated solely by an integer vertex. If the MMD+(least-c) algorithm, that is used for obtaining the treewidth lower bound, returns the same value as the number of integer variables minus one, the vertices of the constructed torso form a clique. Thus, if the number of integer variables is smaller or larger than the treewidth (bounds), then the torso-width of this specific torso is also smaller or larger than the treewidth. This property might be exploited in MILP instances which have a low proportion of integer variables to non-integer variables.
In those cases where the computed torso-width lower and upper bounds are not equal to the number of integer variables, two cases can be distinguished. Either the proportion of integer variables is one, and as a consequence the torso-width bounds are equal to the treewidth bounds, or the proportion is not one. This happens for ten instances altogether. For some of these instances, for example the mcsched and ns1830653 instance, the torso-width lower bound equals the torso-width upper bound.

5.3 Detailed analysis

In order to obtain a more detailed view on the structural parameters and their influence on the practical difficulty of a MILP problem, two instances from the MIPLIB challenge set are analyzed [mipb]. Those two instances are the liu and usAbbrv.8.25_70 instance. Whereas the benchmark set only contains instances which are classified as easy, the liu instance is still unsolved, and the usAbbrv.8.25_70 is classified as hard. Those two instances are of relatively small size compared to other instances in the challenge set [mipb]. The hardness or unsolvability of the instance may thus not solely stem from its size but may have different causes that we try to explore.
The liu problem instance is from the domain of the physical design of VLSI circuits; the second instance usAbbrv.8.25_70 is a railway optimization problem, which was solved for the first time by Gurobi 7.0 (32 threads) in about 29 hours in November 2016 [mipb].

Linear Program

Name	Vars	Cons	Non-Zeroes	Int Vars	Prop Int	Size Obj Fun
liu	1156	2178	10626	1089	0.9420	1
usAbbrv.8.25_70	2312	3291	9628	1681	0.7271	106

Extended Primal Graph

Name	Density	TW LB	TW UB	TD	TO LB	TO UB
liu	0.0149	79	97	100	1086	1086
usAbbrv.8.25_70	0.0055	105	152	675	793	1190

Extended Incidence Graph

Name	Vertices	Edges	Density	TW LB	TW UB
liu	3333	10627	0.0042	68	97
usAbbrv.8.25_70	5604	9734	0.0013	39	125

Extended Dual Graph

Name	Vertices	Edges	Density	TW LB	TW UB
liu	2179	173283	0.0730	250	1638
usAbbrv.8.25_70	3292	36216	0.0067	84	630

Table 5.2: The results of the two instances liu (which is still unsolved) and usAbbrv.8.25_70 (classified as hard) from the MIPLIB challenge set

The detailed properties of the two instances are shown in Table 5.2. The same setup and timeouts as described in Section 5.1 and 5.2 are used, the extended graph representations are again considered. The first two rows describe the properties of the MILP instances. The number of variables and constraints is similar for the two instances, with both having more constraints than variables. They also have a similar number of about 10,000 non-zero coefficients in the constraint matrix (column 'Non-Zeroes', the values are taken from [mipb]). The liu instance has less integer variables than the second instance but a higher proportion of integer variables to the total number of variables (column 'Prop Int'). The last column denotes the number of variables with a non-zero coefficient in the objective function. The liu instance only contains one such variable.
The following three subtables show information about the extended primal, incidence and dual graph representation of the two instances. The number of vertices can be computed by the number of variables and constraints. In the primal graph representation the number of vertices is equal to the number of variables. In the incidence graph representation it is equal to the number of variables plus the number of constraints plus one for the objective function. The liu instance contains two less vertices than expected because two variables only occur with a zero coefficient in the objective function. Those two variables are counted in the number of variables, but they are not considered in the size of the objective function or in the construction of the graph representations. The number of vertices in the dual graph representation is equal to the number of constraints plus one.
The density is computed in the following way: the number of edges in the graph are

divided by the maximum possible number of edges. In the primal and dual graph representation, this is simply two times the number of edges divided by the number of vertices times the number of vertices minus one. As the incidence graph is a bipartite graph, the incidence graph density is computed by dividing the number of edges by the number of constraints times the number of variables.
The remaining columns contain the computed bound values for the treewidth, tree-depth and torso-width.

The density for the (until now) unsolved instance liu is in general higher than for the usAbbrv instance. Especially the dual graph representation has a much larger number of edges and a higher density. The large number of edges in the dual graph representation of the liu instance means that the constraints share a lot of variables. This observation can also be made when we look at the density patterns of the two instances shown in Figure 5.1. Recall that only the constraint matrix and not the objective function is represented in the density pattern. The shape of the liu instance is formed like the letter

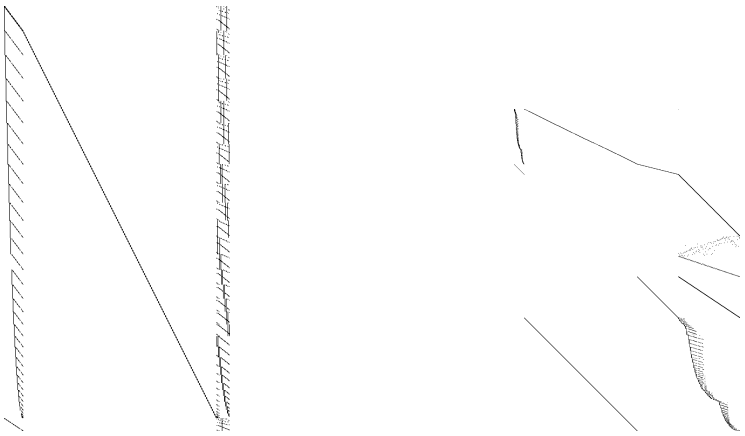


Figure 5.1: The density patterns of the liu instance (left) and usAbbrv.8.25_70 instance (right) [mipb]

'N'. Because of the almost vertical right line formed by many smaller diagonals, a few number of variables occur in a lot of constraints; hence the large number of edges in the dual graph representation of the liu instance. The density pattern of the usAbbrv also consists of diagonals, but these are not distributed in such a connected way. Instead, the density pattern rather consists of several individual parts.

The tree-depth upper bound of the liu instance is only slightly larger than the treewidth upper bound, with an absolute value of three. Considering that the treewidth is at most the tree-depth minus one, this is a rather tight bound. In contrast, the tree-depth of the second instance usAbbrv is a lot larger than its treewidth upper bound and only slightly smaller than the torso-width lower bound. Even increasing the number of iterations for the tree-depth upper bound algorithm (with linear more time effort) does not result in much better bounds, which can be seen in the following table.

Name	Number of iterations	Tree-depth upper bound
liu	100	100
	1000	100
	10000	100
	100000	100
usAbbrv.8.25_70	100	675
	1000	675
	10000	666
	100000	666

The torso-width bounds of the two instances, more exactly the torso-width of the torso that is obtained by collapsing the set of non-integer vertices, is for the liu instance exactly 1086. This means that the integer vertices form a clique in the torso (the two variables that appear with a zero coefficient in the objective function are not contained in the primal graph representation). Except for the dual graph, the torso-width is thus for sure larger than the treewidth of the graph representations. The torso-width of the usAbbrv instance is in between 793 and 1190. It is therefore larger than the treewidth of all three graph representations and the tree-depth of the primal graph representation.

In order to show how the objective function influences the structural parameters, the results for the simplified graph representations of the two challenge instances are shown in Table 5.3. The size of the objective function influences how the values change.

Linear Program

Name	Vars	Cons	Non-Zeroes	Int Vars	Prop Int	Size Obj Fun
liu	1156	2178	10626	1089	0.9420	1
usAbbrv.8.25_70	2312	3291	9628	1681	0.7271	106

Primal Graph

Name	Density	TW LB	TW UB	TD	TO LB	TO UB
liu	0.0149	78	97	100	1086	1086
usAbbrv.8.25_70	0.0036	44	122	710	794	1182

Incidence Graph

Name	Vertices	Edges	Density	TW LB	TW UB
liu	3332	10626	0.0042	68	97
usAbbrv.8.25_70	5603	9628	0.0013	38	119

Dual Graph

Name	Vertices	Edges	Density	TW LB	TW UB
liu	2178	173217	0.0731	251	1621
usAbbrv.8.25_70	3291	35547	0.0066	83	643

Table 5.3: The results for the two instances liu and usAbbrv.8.25_70 where the structural parameters are computed on the simplified graph representations

The results of the liu instance mostly remain the same. The number of vertices and edges of the incidence graph representation decreases by one as expected. The number of vertices of the dual graph representation is also decreased by one, and the number of edges is reduced from 173283 to 173217 edges. This implies that the one variable that occurs in the objective function is the only "binding" variable for 66 constraint links. However, the dual treewidth bounds do not decrease significantly.

For the usAbbrv instance the objective function plays a more significant role in the structure of the MILP instance. 106 out of the 2312 variables have a non-zero coefficient in the objective function. Whereas the number of components is one in the extended graph representations, the number of components is two in all three simplified graph representations. The minimum degree of the primal and dual graph representation changes to zero (computed by MILP-Struct but does not appear in Table 5.3). For this reason, there must be a variable vertex in the extended primal graph that is only due to the objective function connected to another vertex. There also must be a constraint in the MILP instance such that its variables only occur in the objective function (with non-zero coefficients).
For the primal graph representation the amount of edges is reduced approximately to two thirds of the amount of edges of the extended primal graph. The primal treewidth bounds also decrease significantly. At the same time, the tree-depth and torso-width bounds almost stay the same. The values of the simplified incidence graph representation again change in the way one would expect. The number of vertices is decreased by one and the number of edges is decreased by the size of the objective function. Meanwhile, the treewidth bounds almost remain the same. The simplified dual graph has about 700 less edges than the extended dual graph but with almost no decrease in the density or treewidth bounds.

CHAPTER 6

Conclusion

Even though ILP is one of the most famous NP-complete problems [Kar72], in comparison to SAT or CSP very little is known about the structure of ILP instances. The same also holds for MILP instances – while this problem can be viewed as a mixture of ILP and LP, it is far from being easier to solve than ILP. In order to analyze the structure of ILP and MILP instances, the variable-constraint interactions of the instance are considered by means of the primal, incidence and dual graph representation. This approach originally stems from SAT but has also been successfully applied to ILP.
Theoretical results show that ILP and ILP FEASIBILITY are fixed-parameter tractable parameterized by the primal treewidth, incidence treewidth and primal tree-depth with some additional conditions [JK15, GO16, GOR17]. MILP is fixed-parameter tractable by the torso-width, a structural parameter specific for MILP instances [GOR17]. Therefore, as long as these structural parameters are assumed to be small, it is possible to solve the instances in reasonable time.

Whereas for SAT and CSP there already exist studies that show how structural parameters correlate with the solving time of the problem instance [Mat11, BFM05], very little is known in the setting of ILP. In this thesis, we try to close this research gap by analyzing the values of these structural parameters in graph representations of practical ILP and MILP instances. Our goal was to construct a framework which is able to compute a variety of structural parameters of the graphical representations of ILP or MILP instances. With this framework, we wanted to measure structural parameters of practical ILP and MILP instances to analyze the correspondence between the computed parameters and the practical difficulty to solve the instance.
This goal is achieved by our framework MILP-Struct. Being the first of its kind, it can parse ILP or MILP instances in the MPS format, construct the primal, incidence, and dual graph representations and compute the structural parameters of these graph representations. As the treewidth and tree-depth are NP-complete problems themselves [ACP87, Pot88], only lower and upper bounds for the parameters are com-

puted. For the treewidth, many different lower and upper bound methods like degree-based lower bounds or elimination ordering upper bounds exist. The treewidth library LibTW [vDvdHS06], which provides implementations of many of these algorithms, is used for the computation of treewidth bounds; improvements for handling disconnected graphs have been added. For the torso-width, we first construct a torso by collapsing the set of non-integer vertices and then compute treewidth bounds on the resulting torso. For the tree-depth, we use the fact that the height of any tree obtained by a depth-first search is an upper bound on the tree-depth. MILP-Struct can thus compute bounds for the theoretically most significant parameters treewidth, tree-depth and torso-width.

In order to apply our framework on practical integer and mixed integer programs, instances from the well-established MIPLIB library [KAA+11] are used. This library contains a wide range of practical ILP and MILP instances with additional information about the instances and is continuously updated to match the speed improvements of ILP solvers. We present and analyze the computed structural parameters of the benchmark set from the MIPLIB 2010 library. Whereas the incidence treewidth and density is in general quite low for all the analyzed instances, the torso-width for the one specific torso that we analyze is often exactly the number of integer variables of the instance minus one. For the other structural parameters, the computed bound values seldom allow for a fixing of the bound values, i.e. the bounds usually are not very tight. In general, the structural parameters are much smaller than the number of variables though this may not be enough for algorithms that run with time exponential in the parameter.
For two specific instances from the challenge set the influence of the objective function on the structural parameters is analyzed in detail. The construction of the simplified graph representations may result in smaller structural parameters, but the improvement depends upon the size of the objective function and whether the variables also occur in the constraint matrix. Until now it is still an open problem whether the objective function is really needed in the construction of the graph representations so that some theoretical results can be applied [GO16]. As MILP-Struct is able to compute the graph representations with and without the objective function considered, it may also be used if more results in that research area are obtained.

Of course, even given the exact structural parameters one should not expect theoretical algorithms to beat state-of-the-art solvers for ILP. However, MILP-Struct can be used for obtaining an informative overview of the problem instance and structure. This information may then help to estimate whether the ILP or MILP instance can be solved to optimality – and in what time.
Moreover, there is hope that the state-of-the-art heuristics used for finding optimal solutions of ILP or MILP can be enhanced to take into account the tree-like structure or other structural parameters of instances; this would specifically target instances which our framework found to have low treewidth, tree-depth or torso-width. Hence, our tool may be of further use for researchers and practitioners trying to solve specific sets of ILP or MILP instances.

Bibliography

[ACP87] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of Finding Embeddings in a K-tree. *SIAM J. Algebraic Discrete Methods*, 8(2):277–284, April 1987.

[Bal16] Balyo, T.; Heule, M. J. H. Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions. Department of Computer Science Series of Publications B, University of Helsinki, Department of Computer Science, 2016.

[BBHP04] Anne Berry, Jean R. S. Blair, Pinar Heggernes, and Barry W. Peyton. Maximum Cardinality Search for Computing Minimal Triangulations of Graphs. *Algorithmica*, 39(4):287–298, Aug 2004.

[BDD+13] Hans L. Bodlaender, Pål Grønås Drange, Markus S. Dregi, Fedor V. Fomin, Daniel Lokshtanov, and Michal Pilipczuk. A $c^k n$ 5-Approximation Algorithm for Treewidth. In *Proceedings of the 2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, FOCS '13, pages 499–508, Washington, DC, USA, 2013. IEEE Computer Society.

[BDK12] Adam Bouland, Anuj Dawar, and Eryk Kopczynski. On Tractable Parameterizations of Graph Isomorphism. In *IPEC*, volume 7535 of *Lecture Notes in Computer Science*, pages 218–230. Springer, 2012.

[BFK+12] Hans L. Bodlaender, Fedor V. Fomin, Arie M. C. A. Koster, Dieter Kratsch, and Dimitrios M. Thilikos. On Exact Algorithms for Treewidth. *ACM Trans. Algorithms*, 9(1):12:1–12:23, December 2012.

[BFM05] Ramon Béjar, Cèsar Fernández, and Carles Mateu. Statistical Modelling of CSP Solving Algorithms Performance. In Peter van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005: 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005. Proceedings*, pages 861–861. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[BGHK95] Hans L. Bodlaender, John R. Gilbert, Hjalmtyr Hafsteinsson, and Ton Kloks. Approximating Treewidth, Pathwidth, Frontsize, and Shortest Elimination Tree, 1995.

[BGK05] Hans L. Bodlaender, Alexander Grigoriev, and Arie M. C. A. Koster. Treewidth Lower Bounds with Brambles. In *Proceedings of the 13th Annual European Conference on Algorithms*, ESA'05, pages 391–402, Berlin, Heidelberg, 2005. Springer-Verlag.

[Bix12] Robert E. Bixby. A Brief History of Linear and Mixed-Integer Programming Computation, 2012.

[BK10] Hans L. Bodlaender and Arie M.C.A. Koster. Treewidth computations I. Upper Bounds. *Information and Computation*, 208(3):259–275, March 2010.

[BK11] Hans L. Bodlaender and Arie M.C.A. Koster. Treewidth computations II. Lower bounds. *Information and Computation*, 209(7):1103 – 1119, 2011.

[BKM16] Dimitris Bertsimas, Angela King, and Rahul Mazumder. Best subset selection via a modern optimization lens. *Ann. Statist.*, 44(2):813–852, 04 2016.

[BKW04] Hans L. Bodlaender, Arie M. C. A. Koster, and Thomas Wolle. Contraction and Treewidth Lower Bounds. In Susanne Albers and Tomasz Radzik, editors, *Algorithms – ESA 2004: 12th Annual European Symposium, Bergen, Norway, September 14-17, 2004. Proceedings*, pages 628–639. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[Bod93] 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, pages 226–234, New York, NY, USA, 1993. ACM.

[Bod98] Hans L. Bodlaender. A Partial k-arboretum of Graphs with Bounded Treewidth. *Theor. Comput. Sci.*, 209(1-2):1–45, December 1998.

[Bod00] Hans Bodlaender. Necessary Edges in k-Chordalisations of Graphs. *Journal of Combinatorial Optimization, 2003, Vol.7(3), pp.283-290*, 2000.

[CCMN03] François Clautiaux, Jacques Carlier, Aziz Moukrim, and Stéphane Nègre. New Lower and Upper Bounds for Graph Treewidth. In *Proceedings of the 2Nd International Conference on Experimental and Efficient Algorithms*, WEA'03, pages 70–80, Berlin, Heidelberg, 2003. Springer-Verlag.

[CFK+15] Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Daniel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer Publishing Company, Incorporated, 1st edition, 2015.

[CMR00] B. Courcelle, J. A. Makowsky, and U. Rotics. Linear Time Solvable Optimization Problems on Graphs of Bounded Clique-Width. *Theory of Computing Systems*, 33(2):125–150, Apr 2000.

[Cou90] Bruno Courcelle. The Monadic Second-order Logic of Graphs. I. Recognizable Sets of Finite Graphs. *Information and Computation*, 85(1):12–75, 1990.

[csp99] CSPLib: A problem library for constraints. `http://www.csplib.org`, 1999. [Online; accessed 30.12.2017].

[DF92] Rodney G. Downey and Michael R. Fellows. Fixed-parameter intractability. In *[1992] Proceedings of the Seventh Annual Structure in Complexity Theory Conference*, pages 36–49, Jun 1992.

[DF95a] Rodney G. Downey and Michael R. Fellows. Fixed-Parameter Tractability and Completeness I: Basic Results. *SIAM J. Comput.*, 24(4):873–921, August 1995.

[DF95b] Rodney G. Downey and Michael R. Fellows. Fixed-parameter tractability and completeness II: On completeness for W[1]. *Theoretical Computer Science*, 141(1):109 – 131, 1995.

[DF12] Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. Springer Publishing Company, Incorporated, 2012.

[DF13] Rodney G. Downey and Michael R. Fellows. *Fundamentals of Parameterized Complexity*. Springer Publishing Company, Incorporated, 2013.

[DGG+08] Artan Dermaku, Tobias Ganzow, Georg Gottlob, Ben McMahan, Nysret Musliu, and Marko Samer. Heuristic Methods for Hypertree Decomposition. In Alexander Gelbukh and Eduardo F. Morales, editors, *MICAI 2008: Advances in Artificial Intelligence: 7th Mexican International Conference on Artificial Intelligence, Atizapán de Zaragoza, Mexico, October 27-31, 2008 Proceedings*, pages 1–11, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[Die12] Reinhard Diestel. *Graph Theory*, volume 173 of *Graduate texts in mathematics*. Springer, 2012.

[dli] Dlib C++ Library. `http://dlib.net`. [Online; accessed 09.10.2017].

[FG06] Jörg Flum and Martin Grohe. *Parameterized Complexity Theory (Texts in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[FHL05] Uriel Feige, MohammadTaghi Hajiaghayi, and James R. Lee. Improved Approximation Algorithms for Minimum-weight Vertex Separators. In *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*, STOC '05, pages 563–572, New York, NY, USA, 2005. ACM.

[FKT04] Fedor V. Fomin, Dieter Kratsch, and Ioan Todinca. Exact (Exponential) Algorithms for Treewidth and Minimum Fill-In. In Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella, editors, *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004, Turku, Finland, July 12-16, 2004. Proceedings*, pages 568–580, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[FKTV08] Fedor V. Fomin, Dieter Kratsch, Ioan Todinca, and Yngve Villanger. Exact Algorithms for Treewidth and Minimum Fill-In. *SIAM J. Comput.*, 38(3):1058–1079, July 2008.

[FL05] Christodoulos A. Floudas and Xiaoxia Lin. Mixed Integer Linear Programming in Process Scheduling: Modeling, Algorithms, and Applications. *Annals of Operations Research*, 139(1):131–162, Oct 2005.

[FLM+08] Michael R. Fellows, Daniel Lokshtanov, Neeldhara Misra, Frances A. Rosamond, and Saket Saurabh. Graph Layout Problems Parameterized by Vertex Cover. In Seok-Hee Hong, Hiroshi Nagamochi, and Takuro Fukunaga, editors, *Algorithms and Computation: 19th International Symposium, ISAAC 2008, Gold Coast, Australia, December 15-17, 2008. Proceedings*, pages 294–305. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[GD04] Vibhav Gogate and Rina Dechter. A Complete Anytime Algorithm for Treewidth. In *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence*, UAI '04, pages 201–208, Arlington, Virginia, United States, 2004. AUAI Press.

[GN00] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Softw. Pract. Exper.*, 30(11):1203–1233, September 2000.

[GO16] Robert Ganian and Sebastian Ordyniak. The Complexity Landscape of Decompositional Parameters for ILP. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 710–716. AAAI Press, 2016.

[GOR17] Robert Ganian, Sebastian Ordyniak, and M. S. Ramanujan. Going Beyond Primal Treewidth for (M)ILP. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, pages 815–821, 2017.

[hyp] Hypertree. `http://www.dbai.tuwien.ac.at/proj/hypertree/downloads.html`. [Online; accessed 09.10.2017].

[JK15] Bart M. P. Jansen and Stefan Kratsch. A Structural Approach to Kernels for ILPs: Treewidth and Total Unimodularity. In Nikhil Bansal and Irene

Finocchi, editors, *Algorithms - ESA 2015: 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings*, pages 779–791. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[KAA+11] Thorsten Koch, Tobias Achterberg, Erling Andersen, Oliver Bastert, Timo Berthold, Robert E. Bixby, Emilie Danna, Gerald Gamrath, Ambros M. Gleixner, Stefan Heinz, Andrea Lodi, Hans Mittelmann, Ted Ralphs, Domenico Salvagnin, Daniel E. Steffy, and Kati Wolter. MIPLIB 2010: Mixed integer programming library version 5. *Mathematical Programming Computation*, 3(2):103–163, 2011.

[Kar72] Richard M. Karp. Reducibility among Combinatorial Problems. In Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger, editors, *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations, held March 20–22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and sponsored by the Office of Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department*, pages 85–103. Springer US, Boston, MA, 1972.

[Kar84] N. Karmarkar. A New Polynomial-time Algorithm for Linear Programming. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, STOC '84, pages 302–311, New York, NY, USA, 1984. ACM.

[Kin09] Davis E. King. Dlib-ml: A Machine Learning Toolkit. *Journal of Machine Learning Research*, 10:1755–1758, 2009.

[KMP13] Thorsten Koch, Alexander Martin, and Marc E. Pfetsch. Progress in Academic Computational Integer Programming. In Michael Jünger and Gerhard Reinelt, editors, *Facets of Combinatorial Optimization: Festschrift for Martin Grötschel*, pages 483–506. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[KWB05] Arie M. C. A. Koster, Thomas Wolle, and Hans L. Bodlaender. Degree-Based Treewidth Lower Bounds. Technical report, Berlin, Heidelberg, 2005.

[Lap92] Gilbert Laporte. The vehicle routing problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, 59(3):345 – 358, 1992.

[Len83] H. W. Lenstra, Jr. Integer programming with a fixed number of variables. *Math. Oper. Res*, 8(4):538–548, November 1983.

[log] Apache log4j 1.2. `http://logging.apache.org/log4j/1.2/`. [Online; accessed 30.12.2017].

[Luc03] Brian Lucena. A New Lower Bound for Tree-Width Using Maximum Cardinality Search. *SIAM J. Discret. Math.*, 16(3):345–353, March 2003.

[Mat11] Robert Mateescu. Treewidth in Industrial SAT Benchmarks. Technical report, February 2011.

[mipa] `http://miplib.zib.de/miplib3/mps_format.txt`. [Online; accessed 24.10.2017].

[mipb] MIPLIB – the Mixed Integer Programming LIBrary. `http://miplib.zib.de`. [Online; accessed 16.09.2017].

[NdM06] Jaroslav Nešetřil and Patrice Ossona de Mendez. Tree-depth, subgraph coloring and homomorphism bounds. *European Journal of Combinatorics*, 27(6):1022 – 1041, 2006.

[NdM12] Jaroslav Nešetřil and Patrice Ossona de Mendez. *Sparsity: Graphs, Structures, and Algorithms.* Algorithms and Combinatorics. Springer Berlin Heidelberg, 2012.

[Pap81] Christos H. Papadimitriou. On the Complexity of Integer Programming. *J. ACM*, 28(4):765–768, October 1981.

[Pot88] A. Pothen. *The Complexity of Optimal Elimination Trees.* Technical report. Pennsylvania State University, Department of Computer Science, 1988.

[PSP13] Richard Pohl, Vanessa Stricker, and Klaus Pohl. Measuring the Structural Complexity of Feature Models. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE'13, pages 454–464, Piscataway, NJ, USA, 2013. IEEE Press.

[Ram97] Siddharthan Ramachandramurthi. The Structure and Number of Obstructions to Treewidth. *SIAM J. Discrete Math.*, 10:146–157, 1997.

[RS84] Neil Robertson and P.D Seymour. Graph minors. III. Planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49 – 64, 1984.

[RTL76] Donald J. Rose, Robert Endre Tarjan, and George S. Lueker. Algorithmic Aspects of Vertex Elimination on Graphs. *SIAM J. Comput.*, 5(2):266–283, 1976.

[Sch86] Alexander Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, Inc., New York, NY, USA, 1986.

[SG97] Kirill Shoikhet and Dan Geiger. A Practical Algorithm for Finding Optimal Triangulations. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*, AAAI'97/IAAI'97, pages 185–190. AAAI Press, 1997.

[slf] SLF4J. https://www.slf4j.org. [Online; accessed 30.12.2017].

[SS09] Marko Samer and Stefan Szeider. Fixed-Parameter Tractability. In *Handbook of Satisfiability*, chapter 13, pages 425–454. 2009.

[SS10] Marko Samer and Stefan Szeider. Constraint satisfaction with bounded treewidth revisited. *Journal of Computer and System Sciences*, 76(2):103 – 114, 2010.

[ST93] P. D. Seymour and Robin Thomas. Graph Searching and a Min-max Theorem for Tree-width. *J. Comb. Theory Ser. B*, 58(1):22–33, May 1993.

[Sze04] Stefan Szeider. On Fixed-Parameter Tractable Parameterizations of SAT. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing: 6th International Conference, SAT 2003, Santa Margherita Ligure, Italy, May 5-8, 2003, Selected Revised Papers*, pages 188–202. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[TY84] Robert E. Tarjan and Mihalis Yannakakis. Simple Linear-time Algorithms to Test Chordality of Graphs, Test Acyclicity of Hypergraphs, and Selectively Reduce Acyclic Hypergraphs. *SIAM J. Comput.*, 13(3):566–579, July 1984.

[VBS99] Thomas Vossen, Michael Ball, and Robert H. Smith. On the Use of Integer Programming Models in AI Planning. In *In Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 304–309. Morgan Kaufmann, 1999.

[vDvdHS] Thomas van Dijk, Jan-Pieter van den Heuvel, and Wouter Slob. Treewidth.com. http://www.treewidth.com/treewidth/. [Online; accessed 16.09.2017].

[vDvdHS06] Thomas van Dijk, Jan-Pieter van den Heuvel, and Wouter Slob. Computing treewidth with LibTW, 2006.

[vWK17] Rim van Wersch and Steven Kelk. ToTo: An open database for computation, storage and retrieval of tree decompositions. *Discrete Applied Mathematics*, 217(Part 3):389 – 393, 2017.

[wag] Wagu. https://github.com/thedathoudarya/WAGU-data-in-table-view. [Online; accessed 30.12.2017].