TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

MASTERARBEIT

New Heuristic Methods for Tree Decompositions and Generalized Hypertree Decompositions

Ausgeführt am Institut für

Informationssysteme

der Technischen Universität Wien

unter der Anleitung von Univ. Prof. Dr. Georg Gottlob
und Univ. Ass. Dr. Nysret Musliu

durch

Werner Schafhauser

Schulweg 7, 9551 Bodensdorf

Oktober 2006 _____

Abstract

Many practical hard problems in mathematics and computer science may be formulated as constraint satisfaction problems (CSPs). Tree and generalized hypertree decompositions are two important concepts which can be used for identifying and solving tractable classes of CSPs. Unfortunately the task of finding an optimal tree or generalized hypertree decomposition is an $\mathcal{NP}$-complete problem. Thus many heuristic methods have been developed for finding tree decompositions and generalized hypertree decompositions of small width.

In this master thesis we present new heuristic methods for tree and generalized hypertree decompositions. For that purpose we examine already existing heuristic methods for tree decompositions and extend them to an A* algorithm and a genetic algorithm for tree decompositions and to a genetic algorithm and a self-adaptive genetic algorithm for generalized hypertree decompositions. Furthermore we prove that the set of all elimination orderings may act as a search space for the generalized hypertree width and we develop a lower bound heuristic for the generalized hypertree width, which combines lower bound heuristics for tree decompositions with lower bound heuristics for the k-set cover problem. Moreover we show how existing reduction and pruning techniques, for shrinking the search space for the optimal tree decomposition, may also be used for generalized hypertree decompositions. Based on these results we propose a branch and bound algorithm and an A* algorithm for generalized hypertree decompositions.

Computational experiments show that the heuristic methods presented in this thesis are able to compete with other heuristic methods for tree and generalized hypertree decompositions. For many benchmark instances the genetic algorithms and the branch and bound algorithm return improved upper bounds on the treewidth and generalized hypertree width and for some instances the A* algorithms and the branch and bound algorithm are able to fix the exact treewidth and generalized hypertree width.

Kurzfassung

Constraint satisfaction problems (CSPs) bilden eine Problemklasse in der Mathematik und Informatik, die viele praxisrelevante und harte Probleme beinhält. Tree decompositions und generalized hypertree decompositions sind zwei Methoden, mit denen effizient lösbare CSP Instanzen identifiziert und für solche Instanzen effizient Lösungen berechnet werden können. Leider ist das Auffinden der optimalen tree decomposition bzw. generalized hypertree decomposition einer CSP Instanz ein $\mathcal{NP}$-vollständiges Problem. Aus diesem Grund sind in der Vergangenheit bereits viele heuristische Methoden für tree decompositions und generalized hypertree decompositions vorgestellt worden.

Ziel dieser Magisterarbeit ist es, neue heuristische Methoden für tree und generalized hypertree decompositions zu entwickeln. Zu diesem Zweck betrachten wir bereits existierende heuristische Verfahren für tree decompositions und erweitern diese zu einem A* Algorithmus und einem genetischen Algorithmus für tree decompositions bzw. zu einem genetischen Algorithmus und einem selbst adaptierenden genetischen Algorithmus für generalized hypertree decompositions. Weiters beweisen wir, dass elimination orderings einen geeigneten Suchraum für die generalized hypertree width darstellen, und wir entwickeln eine lower bound Heuristik für generalized hypertree width, die lower bound Heuristiken für tree decompositions und für das k-set cover Problem kombiniert. Außerdem zeigen wir, dass existierende Techniken, um den Suchraum für optimale tree decompositions zu verkleinern, auch für generalized hypertree decompositions angewendet werden können. Basierend auf diesen Resultaten entwickeln wir einen branch and bound Algorithmus und einen A* Algorithmus für generalized hypertree decompositions.

Testergebnisse für Benchmark Instanzen zeigen, dass die vorgestellten heuristischen Methoden für tree und generalized hypertree decompositions in der Lage sind, mit anderen Verfahren zu konkurrieren. Die genetischen Algorithmen und der branch and bound Algorithmus finden für viele Instanzen verbesserte obere Schranken für treewidth und generalized hypertree width und für einige Instanzen können die A* Algorithmen und der branch und bound Algorithmus treewidth und generalized hypertree width exakt bestimmen.

Acknowledgements

First and foremost I would like to thank Prof. Georg Gottlob and Dr. Nysret Musliu for supervising and proof-reading this thesis. Within their lectures they sparked my interest in decomposition techniques and heuristics methods. Due to their useful suggestions and their excellent support, writing this master thesis was indeed a great pleasure for me.

I would also like to thank Toni Pisjak and Michael Kipar for the technical support they gave me while I was implementing and testing the algorithms presented within this thesis.

This master thesis was carried out in a project supported by the Austrian Science Fund (FWF) project: Nr. P17222-N04, Complementary Approaches to Constraint Satisfaction (Project time: Sept. 2004-2006).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Constraint Satisfaction Problems (CSPs)

Many areas of our daily lives are affected by constraints. Our lifestyle is largely determined by our income. Laws restrict and regulate the living-together between people within a state. Available time constrains the quantity and quality of work we are able to perform. Thus, every day we try to solve problems in such a way that the problem inherent constraints are satisfied.

Also in science we are often confronted with the task of finding solutions of problems which satisfy constraints. As the complexity of the regarded problems grows we depend on the help of computers in order to solve them. *"Starting with the pioneering work of Montanari [39] researchers in artificial intelligence have investigated a class of combinatorial problems that became known as constraint satisfaction problems (CSPs)."*[33]

Informally speaking, a CSP consists of variables and the values which may be assigned to variables are restricted by one or several constraints. A solution for a CSP is an assignment of allowed values to its variables which satisfies all constraints. Sometimes we are also interested in all such assignments. In mathematics and computer science, especially in the fields of operations research and artificial intelligence, many important real-world problems can be modeled as CSP. For instance, boolean satisfiability problems, scheduling problems, the n-queens problem, boolean conjunctive queries, the graph k-colorability problem and many other interesting problems might be formulated as CSPs.

The main advantage of CSP is that it represents a very general class of problems including many interesting practical problems. By developing methods for solving CSPs we automatically obtain methods for all problems that possess a formulation as CSP. The

main drawback with CSP is that CSP contains many $\mathcal{NP}$-complete problems, implying that all known algorithms that are able to solve CSPs require exponential running time in the worst case.

1.2 Decomposition Methods

In [27] Gottlob et al. write, *"researches in the AI and database community have developed techniques for identifying and solving tractable classes of CSPs, which can be divided into two main groups [40]:*

- ***Tractability due to restricted structure.*** *This includes all tractable classes of CSPs that are identified solely on the base of the structure of the constraint scopes, independently of the actual constraint relations.*
- ***Tractability due to restricted constraint relations.*** *This includes all classes that are tractable due to particular properties of the constraint relations."*

The structure of a CSP is visualized by its constraint hypergraph. Decomposition methods can be used for identifying and solving tractable classes of CSPs by exploiting the structure of the constraint hypergraph, thus they deal with tractability due to restricted structure.

Decomposition methods aim at transforming a CSP instance into another instance which can be solved efficiently. Informally, this is done by decomposing a given CSP into a tree of subproblems. If each of the subproblems is significantly smaller in size than the original CSP we are able to solve the subproblems more efficiently than the original problem. Finally we derive a solution for the original CSP from this tree of subproblems, which again can be done efficiently.

In this master thesis we will consider *tree decompositions* and *generalized hypertree decompositions* among the various decomposition methods that have been developed during the last decades. The notion of tree decompositions was introduced by Robertson and Seymour in [42]. Gottlob, Leone and Scarello proposed a new decompositions method called *hypertree decompositions* in [29] and they showed that hypertree decompositions were able to generalize and beat any other decomposition method in [27]. Generalized hypertree decompositions are derived from hypertree decompositions by dropping one condition of hypertree decompositions [28].

Usually, decomposition methods use a measure called *width* in order to denote the size of the greatest subproblem. The smallest, thus optimal, width of all tree decompositions of a graph is denoted *treewidth* whereas the smallest width of all generalized hypertree decompositions of a hypergraph is denoted *generalized hypertree width*.

In order to solve a CSP we aim at finding a tree decomposition or generalized hypertree decomposition of width near or equal to the treewidth and generalized hypertree width respectively. Both decisions problems, deciding whether there exists a tree decomposition of a graph of width at most k as well as deciding whether there exists a generalized hypertree decomposition of a hypergraph of width at most k are known to be $\mathcal{NP}$-complete, [1] and [26].

1.3 Heuristic Methods

Heuristic methods might help us in order to compute tree decompositions and generalized hypertree decompositions of small width within a reasonable amount of time.

As mentioned in [45], given a problem in computer science, researchers tend to develop algorithms for that problem and try to prove that these algorithms satisfy the following two criteria:

1. good (worst case) running time.
2. a close-to-optimal or optimal solution.

A heuristic method finds a solution to a given problem but it doesn't ensure good running time or it doesn't put a guarantee on the quality of the returned solution or sometimes a heuristic method doesn't satisfy any of the above criteria. Nevertheless heuristic methods are applied to many hard problems in computer science because they often are the only way to achieve good solutions within a short time.

Many heuristic methods have been developed for tree decompositions within the last decades. Bodlaender gives a survey of heuristic methods for tree decompositions in [7] as well as Hicks et al. in [30]. This master thesis presents new heuristic methods for tree decompositions and generalized hypertree decompositions which are based on the following heuristic methods for tree decompositions, generalized hypertree decompositions and related problems:

- A genetic algorithm for triangulating the moral graph of Bayesian networks, a problem strongly related to tree decompositions of graphs, was proposed by Larrañaga et al. in [36].
- Two branch and bound algorithms for tree decompositions are presented in [5] and [24].
- McMahan shows how heuristic methods for tree decompositions may be used in order to generate generalized hypertree decompositions in [37].

Genetic algorithms are a very popular technique for computing solutions for optimization problems although they do not put any guarantee on the quality of the delivered solution. They imitate the principle of evolution by altering and selecting individuals of a population of solutions for a given optimization problem.

Branch and bound algorithms try to reduce the search space that has to be explored for a given optimization problem. They cut off regions in the search space which do not contain solutions that are better than those that have already been found. A branch and bound algorithm is an exact method, if it terminates it will deliver the optimal solution to a problem.

In [37] McMahan combined a technique called *Bucket Elimination*, which originated in constraint satisfaction [16], with several vertex ordering heuristics for tree decompositions and set cover heuristics. The computational results he achieved with his approach were quite promising.

All of the three heuristic methods mentioned above are based on elimination orderings. An elimination ordering is a permutation of the vertices of a graph or hypergraph. It is known that the set of all of its elimination orderings may be used as search space for the treewidth of graphs. Up to the present it is an open question whether elimination orderings can be used as search space for the generalized hypertree width of hypergraphs.

1.4 Research Questions for This Thesis

The intension behind this thesis was to examine and extend existing heuristic methods for tree decompositions and to explore how those methods can be applied directly to generalized hypertree decompositions. Before proceeding further, we will summarize the main objectives of this thesis:

- Develop an A* (pronounced ”A star”) algorithm for tree decompositions, which additionally exploits the techniques used in [5] , [8] and [24] for shrinking the search space. The A* should be able to solve the same problems as the branch and bound algorithms in [5] and [24].
- Develop a genetic algorithm for tree decompositions based on the work that has been carried out in [36] and examine if the genetic algorithm is able to return new upper bounds for known benchmark instances.
- Develop genetic algorithms for generalized hypertree decompositions and examine their performance on known benchmark instances.
- Develop a branch and bound algorithm for generalized hypertree decompositions which is able to compute the generalized hypertree width of a given hypergraph.

- Develop an A* algorithm for generalized hypertree decompositions.

1.5 Main Results

At this point we summarize the main results of this thesis:

- We implement a genetic algorithm for computing treewidth upper bounds based on the genetic algorithm in [36]. Our computational results reveal that the genetic algorithm was able to return improved upper bounds for the treewidth of many graphs of the Dimacs graph coloring benchmark instances [18].
- We propose an A* algorithm for computing the treewidth of graphs which additionally applies reduction and pruning methods presented in [5], [8] and [24] in order to narrow the search space which has to be explored. Computational results show that the algorithm is able to solve nearly all instances of the Dimacs graph coloring benchmarks [18] which have been solved by the algorithms in [5] and [24]. For an additional instance the treewidth could be fixed.
- We prove that the set of all elimination orderings may be used as search space for the generalized hypertree width of a hypergraph.
- We implement a genetic algorithm for computing upper bounds on the generalized hypertree width of hypergraphs. Computational results showed that the genetic algorithm was able to return improved upper bounds on the generalized hypertree width for many benchmark instances [22].
- We implement a self-adaptive island genetic algorithm for generalized hypertree width upper bounds based on [19]. This algorithm is able to adjust its control parameters itself and doesn't require time-consuming experiments in order to obtain suitable values for those control parameters.
- We develop a general technique which combines lower bounds for treewidth and lower bounds for the k-set cover problem to get a lower bound for the generalized hypertree width of hypergraphs and we propose a concrete lower bound heuristic for generalized hypertree width.
- We propose a branch and bound algorithm for generalized hypertree width of hypergraphs which is based on elimination orderings and the developed lower bound heuristic. The branch and bound algorithm will return the generalized hypertree width of a given hypergraph, if it is given enough time. The branch and bound algorithm was able to compute the generalized hypertree width for some benchmark hypergraphs [22]. Furthermore it returned improved upper bounds for some benchmark instances.

- We propose an A* algorithm for generalized hypertree width which is based on the same results as the branch and bound algorithm. The A* algorithm was able to compute the generalized hypertree width for some benchmark hypergraphs [22] and for some instances it returned improved lower bounds on the generalized hypertree width.

1.6 Further Organization

This thesis comprises 10 chapters. In chapter 2 we give preliminary information about CSPs, decomposition methods, tree and generalized hypertree decompositions. In chapter 3 we show that we may obtain a generalized hypertree decomposition of smallest width from at least one elimination ordering. As a consequence the set of all elimination orderings represents a search space for the generalized hypertree width of hypergraphs. In chapter 4 we give an overview of those heuristic methods used within this thesis. In the following chapters we present new heuristic methods for tree and generalized hypertree decompositions. In chapter 5 we propose an A* algorithm for computing the treewidth of graphs, in chapter 6 a genetic algorithm for computing treewidth upper bounds, in chapter 7 we introduce two genetic algorithms for computing upper bounds on the generalized hypertree width of hypergraphs, in chapter 8 we present a branch and bound algorithm for computing the generalized hypertree width and in chapter 9 an A* algorithm for computing the generalized hypertree width. Chapter 10 concludes and describes work that remains to be done.

Chapter 2

Preliminaries

2.1 Graphs and Hypergraphs

Definition 1 (Graph [15])**.** A *graph* $G = (V, E)$ is a structure that consists of a finite set of *vertices* $V = \{v_1, ..., v_n\}$, and a set of *edges*, $E = \{e_1, ..., e_m\}$. Each edge e is incident to an unordered pair of vertices $\{u, v\}$.

Definition 2 (Hypergraph [15])**.** A *hypergraph* is a structure $\mathcal{H} = (V, H)$ that consists of vertices $V = \{v_1, ..., v_n\}$ and a set of subsets of these vertices $H = \{h_1, ..., h_m\}$, $h_i \subseteq V$, called hyperedges. The hyperedges differ from regular edges in that they may "connect" (or are defined over) more than one or two variables. Note that every graph may be regarded as hypergraph whose hyperedges connect two vertices.

Definition 3 (Gaifman graph, primal graph [15])**.** Let $\mathcal{H} = (V, H)$ be a hypergraph. The *Gaifman graph* or *primal graph* of $\mathcal{H}$, denoted $G^*(\mathcal{H})$, is a graph obtained from $\mathcal{H}$ as follows:

1. $G^*(\mathcal{H})$ owns the same set of vertices as $\mathcal{H}$.
2. Two vertices v_i and v_j are connected by an edge in $G^*(\mathcal{H})$ iff v_i and v_j appear together within a hyperedge of $\mathcal{H}$.

Definition 4 (Dual graph [15])**.** A hypergraph $\mathcal{H} = (V, H)$ can be mapped to a regular graph called a *dual graph*, $\mathcal{H}^{dual}$. The vertices of the dual graph are the hyperedges of $\mathcal{H}$, and two vertices are connected in $\mathcal{H}^{dual}$ if their corresponding hyperedges share a vertex in $\mathcal{H}$. Each vertex of the dual graph is labeled by the vertices of the corresponding hyperedge in $\mathcal{H}$.

Definition 1, 2 and 4 were taken almost verbatim from [15]. In order to distinguish graphs from hypergraphs we will often denote graphs as regular graphs.

2.2 Constraint Satisfaction Problems

2.2.1 Basic Definitions

Definition 5 (Constraint Satisfaction Problem [15], [45])**.** A constraint satisfaction problem (or CSP) is a triple $\langle X, D, C\rangle$ consisting of *variables*, *domains* and *constraints*. The set $X = \{x_1, ..., x_n\}$ contains the *variables* of the CSP. The collection $D = \{D_1, ..., D_n\}$ contains the finite *domains* for each variable. The domain of a variable lists the allowed values for that variable. Each *constraint* C_i in $C = \{C_1, ..., C_m\}$ is defined over a subset S_i of variables, $S_i \subseteq X$, denoted the *scope* of constraint C_i. A constraint C_i specifies the allowed combinations of values for the variables in its scope S_i. Thus, a constraint C_i may also be written as a pair $C_i = \langle S_i, R_i\rangle$, where R_i is a relation defined on S_i whose tuples represent the allowed values.

Definition 6 (Solution of CSP)**.** A solution of a CSP is a *complete consistent assignment* from the values of the domains to the corresponding variables. By complete we mean that we assign a value to each variable of the CSP and a complete assignment is consistent if it satisfies all constraints. The problem of deciding whether a CSP instance has a solution is called *constraint satisfiability (CS)*. Sometimes we are also interested in finding *all complete consistent assignments*.

Many interesting real world problems possess a representation as CSP. For instance, map and graph coloring problems, boolean satisfiablity problems, boolean conjunctive queries, the n-queens problem and many more may be formulated as CSPs.

Example 1 (Map 3-Coloring of Australia from [45])**.** The problem of coloring the states and territories of Australia in such a way that neighboring regions have distinct colors may be modeled as CSP. Figure 2.1 shows the map of Australia and a possible valid 3-coloring.

Variables: $X = \{WA, NT, Q, SA, NSW, V, TAS\}$
the federal states and territories of Australia

Domains: $D = \{D_{WA}, D_{NT}, D_Q, D_{SA}, D_{NSW}, D_V, D_{TAS}\}$
$\forall D_i \in D : D_i = \{r, g, b\}$
each state may be colored red (r), green (g) or blue (b)

Constraints: $C = \{C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8, C_9\}$
$C_1 = \langle\{NT, WA\}, R_1\rangle$
$C_2 = \langle\{SA, WA\}, R_2\rangle$
$C_3 = \langle\{NT, Q\}, R_3\rangle$
$C_4 = \langle\{NT, SA\}, R_4\rangle$
$C_5 = \langle\{Q, SA\}, R_5\rangle$
$C_6 = \langle\{NSW, Q\}, R_6\rangle$
$C_7 = \langle\{NSW, V\}, R_7\rangle$
$C_8 = \langle\{NSW, SA\}, R_8\rangle$
$C_9 = \langle\{SA, V\}, R_9\rangle$
$\forall R_i : R_i = \{(r, g), (r, b), (g, r), (g, b), (b, r), (b, g)\}$
neighboring regions must have distinct colors

Solution: $WA = r, NT = g, SA = b, Q = r, NSW = g, V = r, TAS = g$

Figure 2.1: Map of Australia and a valid 3-coloring [45].

Example 2 (Boolean Satisfiability (SAT))**.** Given the boolean formula $\phi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_4}) \wedge (\overline{x_3} \vee \overline{x_5})$ in conjunctive normal form, the boolean satisfiability problem for ϕ asks whether we can assign the values true or false to the variables of ϕ such that ϕ evaluates to true.

Variables:	$X = \{x_1, x_2, x_3, x_4, x_5\}$ the variables in ϕ
Domains:	$D = \{D_{x_1}, D_{x_2}, D_{x_3}, D_{x_4}, D_{x_5}\}$ $\forall D_i \in D : D_i = \{t, f\}$ each variable may be set to true (t) or false (f)
Constraints:	$C = \{C_1, C_2, C_3\}$ $C_1 = \langle\{x_1, x_2, x_3\}, R_1\rangle$ $C_2 = \langle\{x_3, x_5\}, R_2\rangle$ $C_3 = \langle\{x_1, x_4\}, R_3\rangle$ $R_1 = \{(f, f, f), (f, f, t), (f, t, f), (f, t, t), (t, f, t), (t, t, f), (t, t, t)\}$ $R_2 = \{(f, f), (f, t), (t, f)\}$ $R_3 = \{(f, f), (t, f), (t, t)\}$ there is a constraint on each clause containing the value combinations that will make the clause true
Solution:	$x_1 = t, x_2 = t, x_3 = f, x_4 = t, x_5 = f$

2.2.2 Complexity of CSPs

Given a CSP the number of possible complete variable assignments is $O(d^n)$, where d denotes the maximum domain size. For instance, in example 1 we had to color seven states or territories with one out of three allowed colors, and in example 2 we had to solve a SAT instance with five variables over the two boolean values *true* and $false$. This results in 3^7 possible complete assignments for example 1 and in 2^5 complete assignments for example 2 respectively. Checking whether a complete assignment is consistent with the CSP's constraints can be done in polynomial time, thus CSP is a member of $\mathcal{NP}$. $\mathcal{NP}$-hardness of CSP follows from the fact that many $\mathcal{NP}$-complete problems can be transformed into a CSP formulation and this transformation can be done in time polynomial in the size of the original problem. It follows that CSP is an $\mathcal{NP}$-complete problem itself.

If we build the natural join over all constraint relations of a given CSP we will get a relation consisting of all complete consistent assignments for that CSP. The natural join of m constraint relations of size at most n is feasible in $O(n^{m-1}logn)$ time. $O(n^{m-1}logn)$

is an upper bound for both constraint satisfiability and computing all complete consistent assignments of a CSP [10].

2.2.3 Constraint Hypergraphs, Join Trees and Acyclic CSPs

Definition 7 (Constraint Hypergraphs)**.** A CSP can be visualized by its *constraint hypergraph*. Given a CSP we can derive its constraint hypergraph by introducing a vertex for each variable of the CSP. For each constraint we introduce a hyperedge connecting those vertices that correspond to the variables within the scope of the constraint.

Example 3. Figure 2.2 shows the constraint hypergraphs for the (a) map coloring problem in example 1 and for the (b) satisfiability problem in example 2. In the map coloring problem we introduced only binary constraints on each pair of neighboring regions thus the resulting constraint hypergraph is a regular graph.

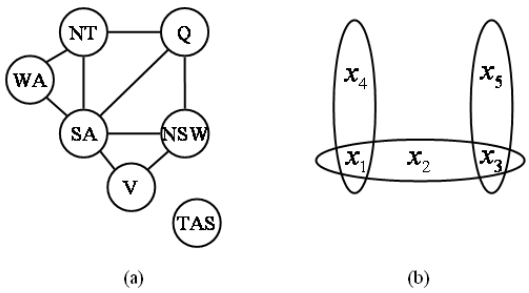

Figure 2.2: Constraint hypergraphs for the problems in Example 1 and 2.

Definition 8 (Join Tree [15])**.** Given a CSP, its constraint hypergraph $\mathcal{H}$ and the dual graph $\mathcal{H}^{dual}$, a *join tree* for the CSP is a subgraph of $\mathcal{H}^{dual}$ which

1. is a tree consisting of the same set of vertices as $\mathcal{H}^{dual}$.
2. which satisfies the connectedness condition for join trees. The connectedness condition for join trees requires that for each variable Y of the CSP the vertices in the join tree containing Y form a subtree of the join tree.

Note that there is a one-to-one correspondence between the constraints of the CSP and the vertices of the join tree.

Example 4. Figure 2.3 [15] shows (a) a hypergraph, (b) its the dual graph and (c) a join tree.

Definition 9 (Acyclic CSP [15])**.** A CSP which has a join tree is called an *acyclic* CSP.

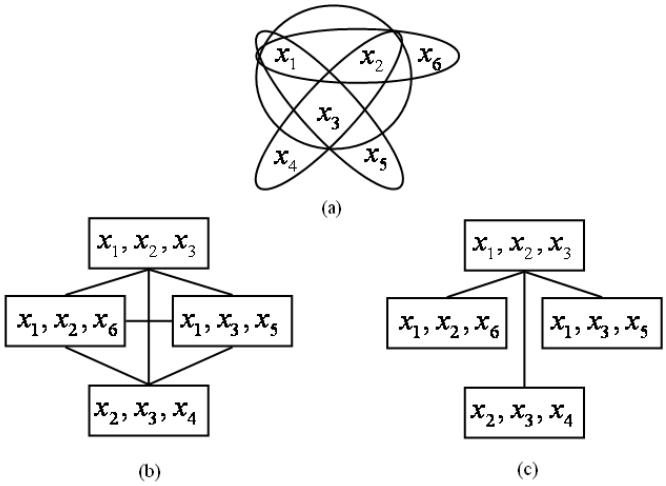

Figure 2.3: A hypergraph (a), its dual graph (b) and a join tree (c) [15].

It is well known that acyclic CSPs can be solved efficiently [14]. Given an acyclic CSP and a join tree we are able to derive a solution from the join tree by applying the algorithm *Acyclic Solving* (Figure 2.4) as presented in [15].

Algorithm Acyclic Solving determines whether there is a solution for an acyclic CSP by processing the join tree in bottom-up fashion. In each step those tuples are deleted from the constraint relation of a parent vertex v_j that do not match any tuple in the constraint relation of the current vertex v_i. If the CSP has no solution the empty relation will be created eventually at some vertex.

For computing a complete consistent assignment of the CSP, algorithm Acyclic Solving processes the join tree of reduced constraint relations in top-down fashion. Starting at the root it selects a tuple which is consistent with the values that have already been found and assigns new values according to the selected tuple. Figure 2.5 visualizes the effect of algorithm Acyclic Solving when applied to a CSP and its join tree. Crossed out tuples are eliminated by the semi-joins in the bottom-up phase. Gray tuples are selected in the top-down phase.

Algorithm: Acyclic Solving

Input: a constraint satisfaction problem $\langle X, D, C\rangle$, $C = \{C_1, ..., C_m\}$ and a join tree T
Output: a solution to the problem if existing

1. Let $d = (v_1, ..., v_m)$ be an ordering of the vertices in T such that v_1 is T's root and each vertex precedes all of its children in d.
2. Associate each constraint $C_i = \langle S_i, R_i\rangle$ with its corresponding vertex v_i in T.
3. */* BOTTOM-UP PHASE - eliminate not matching tuples */*
   for $i = m$ **to** 2 **do**
   Let v_j be the parent vertex of v_i in T
   $R_j := R_j \ltimes R_i$ */* update relation R_j associated parent v_j */*
   if R_j is the empty relation **then exit** */* the CSP has no solution */*
4. */* TOP-DOWN PHASE - find a complete consistent assignment */*
   Select a tuple in R_1
   for $i = 2$ **to** m **do**
   Select a tuple in R_i that is consistent with all previous assignments.

Figure 2.4: Algorithm Acyclic Solving [15].

Algorithm Acyclic Solving can be implemented in such a way that its running time is in $O(mnlogn)$, where m is the number of constraints and n is the size of the largest constraint relation, thus acyclic CSPs can be solved efficiently. Furthermore, recognizing whether a CSP is acyclic and computing the join tree of an acyclic CSP can also be done efficiently [15].

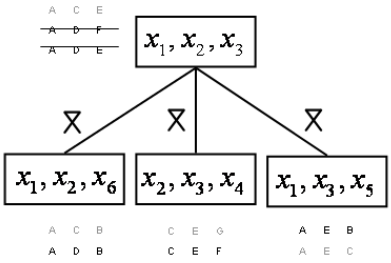

Figure 2.5: Acyclic Solving applied to a CSP and its join tree.

2.3 Decomposition Methods

We have seen that acyclic CSPs can be solved efficiently but we are still confronted with a worst case running time of $O(n^{m-1}logn)$ for solving CSPs in general.

Decomposition methods may be used for identifying and solving tractable classes of CSPs by exploiting the structure of the constraint hypergraph. Given an CSP instance I, a decomposition method transforms I into a solution-equivalent and acyclic CSP instance I'. If I' can be found in time polynomial in $\mid I \mid$ and if the size of the largest relation in I' is polynomial in $\mid I \mid$ we will be able to solve I' in polynomial time since I' is acyclic [27]. Decomposition methods use a measure called *width* in order bound the size of the largest relation in I'.

2.3.1 Tree Decompositions

Basic Definitions

Definition 10 (Tree [29])**.** Let $\mathcal{H} = (V, H)$ be a hypergraph. A *tree for a hypergraph* $\mathcal{H}$ is a pair $\langle T, \chi \rangle$ where $T = (N, E)$ is a rooted tree, and χ is a labeling function which associates to each vertex $p \in N$ the set $\chi(p) \subseteq V$.

Definition 11 (Tree Decomposition, width, treewidth [29])**.** A *tree decomposition* of a hypergraph $\mathcal{H}$ is a tree $TD = \langle T, \chi \rangle$ for $\mathcal{H}$ which satisfies the following two conditions.

1. for each hyperedge $h \in H$, there exists $p \in vertices(T)$ such that $h \subseteq \chi(p)$.
2. for each variable $Y \in V$, the set $\{p \in vertices(T) \mid Y \in \chi(p)\}$ induces a (connected) subtree of T (connectedness condition).

The *width* of a tree decomposition $\langle T, \chi \rangle$ is $max_{p \in vertices(T)} |\chi(p) - 1|$. The *treewidth* of $\mathcal{H}$ is the minimum width over all its tree decompositions.

Given a CSP, a tree decomposition of its constraint hypergraph may be regarded as a join tree of a solution-equivalent acyclic CSP. Thus a vertex of the the tree decomposition represents a subproblem of the new acyclic CSP and the width of a tree decomposition acts as an upper bound on the size of the greatest subproblem. The treewidth of the constraint hypergraph is the minimal width which can be achieved by one of its tree decompositions. When solving a CSP from one of its tree decompositions, the first condition for tree decompositions ensures that all constraints of the original CSP must appear in at leat one subproblem of the new acyclic CSP. The second condition for tree decompositions guarantees that any variable must be assigned the same value in each subproblem in which it appears.

Example 5. Figure 2.6 shows (a) a constraint hypergraph and (b) a tree decomposition of *width* = 2 for the CSP presented below. Each tree vertex contains the variables/vertices which are associated to it by the labeling function χ.

Variables: $X = \{x_1, x_2, x_3, x_4, x_5, x_6\}$

Domains: $D = \{D_{x_1}, D_{x_2}, D_{x_3}, D_{x_4}, D_{x_5}, D_{x_6}\}$
$D_{x_1} = \{a, b\}$, $D_{x_2} = D_{x_3} = ... = D_{x_6} = \{b, c\}$

Constraints: $C = \{C_1, C_2, C_3\}$
$C_1 = \langle\{x_1, x_2, x_3\}, R_1\rangle$
$C_2 = \langle\{x_1, x_5, x_6\}, R_2\rangle$
$C_3 = \langle\{x_3, x_4, x_5\}, R_3\rangle$
$R_1 = \{(a, b, c), (a, c, b), (b, b, c)\}$
$R_2 = \{(a, b, c), (a, c, b)\}$
$R_3 = \{(c, b, c), (c, c, b)\}$

The concept of tree decompositions was introduced by Robertson and Seymour in [42] and was originally defined only for regular graphs. Since every graph may be regarded as hypergraph with two vertices in each of its hyperedges, Definition 11 covers the definition for tree decompositions of graphs and extends the concept of tree decompositions onto arbitrary hypergraphs. Yet another connection between tree decompositions of graphs and tree decompositions of hypergraphs is given by Lemma 1, taken from [33].

Lemma 1 ([33])**.** *$\langle T, \chi\rangle$ is a tree decomposition of hypergraph $\mathcal{H}$ iff it is a tree decomposition of $G^*(\mathcal{H})$, the primal graph or Gaifman graph of $\mathcal{H}$.*

Computational Results for Tree Decompositions

Given a CSP instance *I*, its constraint hypergraph and a corresponding tree decomposition of *width* = *k*, a solution for CSP can be computed in time $O(nd^{k+1})$, where *n* denotes the number of variables of the CSP and *d* the maximum domain size [15]. Thus we are interested in finding a tree decomposition whose width is close-to or equal the treewidth. Unfortunately computing the treewidth of a graph is an $\mathcal{NP}$-hard problem. The formal decision problem of treewidth asks if there exists a tree decomposition of width at most *k*, for some integer *k*. Arnborg et al. proved $\mathcal{NP}$-completeness for that problem if *k* is part of the input [1]. If we regard *k* as a constant Bodlaender [6] presented a linear time algorithm which decides whether there exists a tree decomposition of width at most *k*. Moreover this linear time algorithm is also able to compute a tree decomposition of

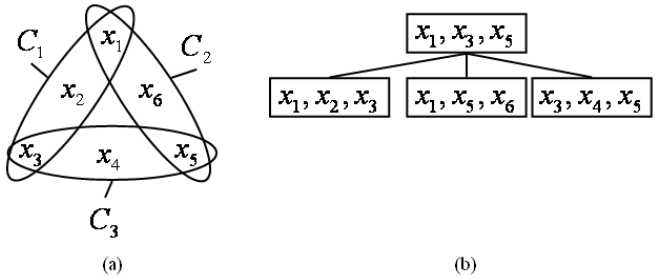

Figure 2.6: Constraint hypergraph (a) and a possible tree decomposition of *width=2* (b).

width at most k. In practice this linear time algorithm performs very slowly even for small values for k due to a huge constant factor.

2.3.2 Generalized Hypertree Decompositions

In [29] Gottlob et al. proposed a new decomposition method which they called *hypertree decompositions*. Hypertree decompositions were originally introduced in database theory as a decomposition method for identifying and solving tractable classes of boolean conjunctive queries. In [27] Gottlob et al. showed that hypertree decompositions may be applied to CSP as well and gave a comparison of structural CSP decompositions methods which revealed that hypertree decompositions strongly generalize all other observed decomposition methods. This means that whenever another decomposition method guarantees polynomial runtime for classes of CSPs then also hypertree decompositions are able to solve these classes in polynomial time but there are classes of CSPs that can be solved in polynomial time by hypertree decompositions but cannot be solved efficiently by any other decomposition method explored in [27]. The quality of a hypertree decomposition is again measured by its *width*, and the smallest width a hypertree decomposition can achieve for a hypergraph $\mathcal{H}$ is denoted *hypertree width*, $hw(\mathcal{H})$. For fixed k, the problem of checking whether the hypertree width of a hypergraph is at most k is feasible in polynomial time, as well as computing a hypertree decomposition of width at most k [29].

Generalized hypertree decompositions represent a variation of hypertree decompositions. They are obtained by dropping one condition in the definition of hypertree decompositions, thus they generalize the concept of hypertree decompositions as indicated by their name.

Basic Definitions

Definition 12 (Hypertree [29])**.** Let $\mathcal{H} = (V, H)$ be a hypergraph. A *hypertree for a hypergraph* $\mathcal{H}$ is a triple $\langle T, \chi, \lambda \rangle$, where $T = (N, E)$ is a rooted tree, and χ and λ are labeling functions which associate to each vertex $p \in N$ two sets $\chi(p) \subseteq V$ and $\lambda(p) \subseteq H$.

Definition 13 (Generalized Hypertree Decomposition [28])**.** A *generalized hypertree decomposition* of a hypergraph $\mathcal{H}$ is a hypertree $GHD = \langle T, \chi, \lambda \rangle$ for $\mathcal{H}$ which satisfies the following three conditions:

1. for each edge $h \in H$, there exists $p \in \mathit{vertices}(T)$ such that $h \subseteq \chi(p)$.
2. for each variable $Y \in V$, the set $\{p \in \mathit{vertices}(T) \mid Y \in \chi(p)\}$ induces a (connected) subtree of T (connectedness condition).
3. for each $p \in \mathit{vertices}(T)$, $\chi(p) \subseteq \mathit{var}(\lambda(p))$.

The *width* of a generalized hypertree decomposition $\langle T, \chi, \lambda \rangle$ is $max_{p \in vertices(T)} |\lambda(p)|$. The *generalized hypertree-width*, $ghw(\mathcal{H})$, of $\mathcal{H}$ is the minimum width over all its generalized hypertree decompositions.

The first and the second condition for generalized hypertree decompositions are identical with the conditions for tree decompositions, thus a generalized hypertree decomposition of a hypergraph $\mathcal{H}$ is a tree decomposition of $\mathcal{H}$ at the same time. The third condition says that for each vertex of the generalized hypertree decomposition the variables within the χ-set of the vertex must be contained by at least one hyperedge in the λ-set of the vertex.

Note that the above definition does not require that every hyperedge has to be associated with at least one vertex. This is necessary in order to guarantee problem-equivalence between a constraint hypergraph and its generalized hypertree decomposition. Lemma 2 shows that each generalized hypertree decomposition can be changed efficiently in order to satisfy this additional requirement.

Definition 14 (from Definition 4.2 in [29])**.** A generalized hypertree decomposition $\langle T, \chi, \lambda \rangle$ of hypergraph $\mathcal{H}$ is a *complete generalized hypertree decomposition* of $\mathcal{H}$ if, for each $h \in H$, there exists $p \in \mathit{vertices}(T)$ such that $h \subseteq \chi(p)$ and $h \in \lambda(p)$.

Lemma 2 (from Lemma 4.4 in [29])**.** *Given a hypergraph $\mathcal{H}$, every k-width generalized hypertree decomposition GHD of $\mathcal{H}$ can be transformed in logspace into a k-width complete generalized hypertree decomposition GHD', whose size is $O(|\mathcal{H}| + |GHD|)$.*

Like a tree decomposition, also a complete generalized hypertree decomposition may be regarded as a join tree of a solution-equivalent acyclic CSP. Again, a vertex of

the complete generalized hypertree decomposition represents a subproblem of the new acyclic CSP. Each subproblem is defined by the vertices within its χ-set and by the hyperedges within its λ-set. In contrary to the width of tree decompositions the width of a generalized hypertree decomposition is the maximum number of hyperedges or constraints associated with a vertex, which measures the complexity of the subproblem more accurately. A subproblem consisting of many variables which are restricted by few constraints is easily solvable.

The generalized hypertree width of a constraint hypergraph denotes the width of an optimal generalized hypertree decomposition.

Example 6. Figure 2.7 shows a (complete) generalized hypertree decomposition for the constraint hypergraph of the CSP in example 5. The variables/vertices which appear in each tree vertex are those which are associated to it by the labeling function χ. The constraints/hyperedges within each tree vertex are those which are associated by the labeling function λ. It might be that a constraint/hyperedge associated with a tree vertex contains a variable/vertex which does not belong to the variables/vertices associated with the tree vertex. Such variables/vertices are marked with a "_" in the contraints/hyperedges.

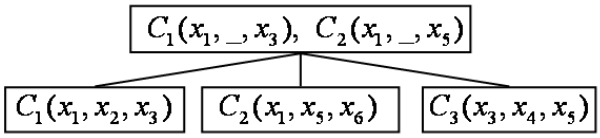

Figure 2.7: Generalized hypertree decomposition of *width=2*.

Computational Results for Generalized Hypertree Decompositions

Given a CSP instance I, its constraint hypergraph and a corresponding complete generalized hypertree decomposition of $width = k$, a solution for CSP can be computed in time $O(|I|^{k+1} log|I|)$, which is polynomial in the size of the CSP instance [27]. Computing all complete consistent assignments is feasible in output-polynomial time [27].

The generalized hypertree width of a hypergraph $\mathcal{H}$ doesn't exceed both its treewidth and its hypertree width, $ghw(\mathcal{H}) \leq hw(\mathcal{H}) \leq tw(\mathcal{H})$, thus problems may be solved more efficiently from generalized hypertree decompositions. Unfortunately, deciding whether $ghw(\mathcal{H}) \leq k$ is an $\mathcal{NP}$-complete problem [26] and moreover deciding whether $ghw(\mathcal{H}) \leq k$ remains $\mathcal{NP}$-complete even for fixed k [47].

2.4 Solving CSPs from Tree Decompositions and Generalized Hypertree Decompositions

Figure 2.8 shows a tree decomposition for the CSP in example 5 and visualizes how a solution for the CSP is derived from it. For solving the CSP from the tree decomposition we use steps 4. and 5. of algorithm *Join Tree Clustering* in [15]. Given the CSP and a tree decomposition Join Tree Clustering applies the following strategy:

1. Each constraint is placed in one vertex of the tree decomposition containing its scope. Afterwards each vertex represents the subproblem of finding all complete assignments for the variables of the vertex which are consistent with the associated constraints.

2. Each subproblem is solved independently. The solution for each subproblem is shown within the relation associated with the vertices in Figure 2.8. Now we have obtained a join tree of the solution equivalent acyclic problem.

3. Apply algorithm Acyclic Solving for finding a complete consistent assignment.

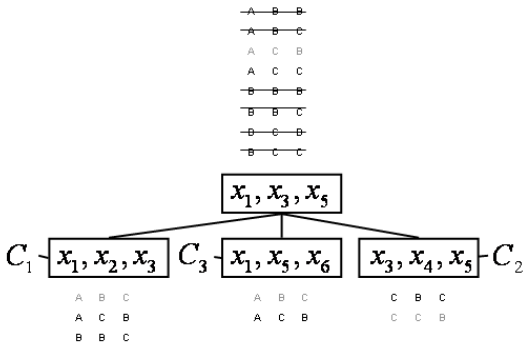

Figure 2.8: Solving example 5 from a tree decomposition.

Figure 2.9 shows a complete generalized hypertree decomposition for the CSP in example 5 and visualizes how a solution for the CSP is derived from it. Given a CSP and a generalized hypertree decomposition we are able to derive a solution for the CSP as follows:

1. Compute a complete generalized hypertree decomposition.
2. For each vertex p compute a new constraint relation R_p which is the projection on the variables in $\chi(p)$ of the join of the constraint relations in $\lambda(p)$, $R_p := \pi_{\chi(p)} \bowtie_{h \in \lambda(p)} h$. Associate R_p with vertex p. Figure 2.8 shows the resulting relations. Now we have obtained a join tree of the solution equivalent acyclic problem.
3. Apply algorithm Acyclic Solving for finding a complete consistent assignment.

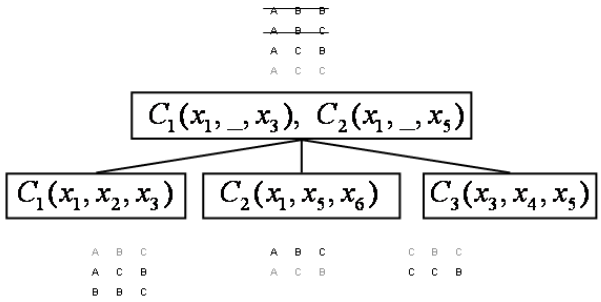

Figure 2.9: Solving example 5 from a (complete) generalized hypertree decomposition.

2.5 Bucket Elimination

In [37] McMahan showed how a method named *bucket elimination* [15] may be used for the creation of tree decompositions and generalized hypertree decompositions. Bucket elimination itself originates from constraint satisfaction. Informally speaking, bucket elimination algorithms tend to solve CSP by creating a tree decomposition and solving the problem on that tree decomposition. An example for a bucket elimination algorithm is algorithm *Adaptive Consistency* in [15]. In order to obtain a tree decomposition bucket eliminations requires an elimination ordering, which is a permutation of the vertices of the constraint hypergraph.

2.5.1 Creating Tree Decompositions via Bucket Elimination

Definition 15 (Elimination Ordering)**.** Given a hypergraph $\mathcal{H} = (V, H)$, an *elimination ordering* for $\mathcal{H}$ is an ordering $\sigma = (v_1, ..., v_n)$ of the vertices in V.

Given a constraint hypergraph $\mathcal{H} = (V, H)$ and an elimination ordering $\sigma = (v_1, ..., v_n)$ of the hypergraph's vertices algorithm *Bucket Elimination* [37] (Figure 2.10) returns a tree decomposition for $\mathcal{H}$.

Initially, the algorithm creates a bucket for each vertex of the constraint hypergraph and puts the vertices of each hyperedge into the bucket of the maximum vertex within this hyperedge. The maximum vertex of $V' \subseteq V$ is the vertex with the highest index according to σ.

Afterwards the buckets are processed in order given by σ. When processing bucket B_{v_i}, we look at the set $A := \chi(B_{v_i}) - \{v_i\}$. Intuitively, A contains only vertices that will be processed after B_{v_i}. The set A is copied to the bucket B_{v_j} of its maximum vertex v_j. Additionally B_{v_i} and B_{v_j} are connected by an edge.

Finally we get a tree decomposition, where the buckets and the introduced edges act as a tree and the contents of the buckets represents the vertices within the χ-sets. Figure 2.11 shows a hypergraph (a) and the tree decomposition (b) returned by algorithm Bucket Elimination using the elimination ordering $\sigma = (x_6, x_5, x_4, x_3, x_2, x_1)$.

Fortunately, there exists at least one elimination ordering which forces algorithm Bucket Elimination to return a tree decomposition of optimal width (treewidth) [12], [34], [43]. The set of all elimination orderings of a hypergraph $\mathcal{H}$ may act as search space for the optimal tree decomposition. As a consequence of [1] finding an elimination ordering resulting in an optimal tree decomposition is an $\mathcal{NP}$-complete problem.

2.5.2 Bucket Elimination for Generalized Hypertree Decompositions

In [37] McMahan showed how bucket elimination can be extended in order to obtain generalized hypertree decompositions. The main idea behind his approach is that every generalized hypertree decomposition $\langle T, \chi, \lambda \rangle$ may be considered as a tree decomposition which satisfies an additional property.

- for each $p \in vertices(T), \chi(p) \subseteq var(\lambda(p))$.

This property requires that the hyperedges in the λ-sets contain the variables in the corresponding χ-sets. Thus each tree decomposition can be transformed into a generalized hypertree decomposition by attaching hyperedges to the decomposition vertices which contain their variables.

Algorithm: Bucket Elimination

Input: a (constraint) hypergraph $\mathcal{H} = (V, H)$
an elimination ordering $\sigma = (v_1, ..., v_n)$ of the vertices in V
Output: a tree decomposition $\langle T, \chi \rangle$ for $\mathcal{H}$

1. Initially $B = \emptyset$, $E = \emptyset$
   for each vertex v_i introduce an empty bucket B_{v_i}, $\chi(B_{v_i}) := \emptyset$
2. Fill the buckets $B_{v_1}, ..., B_{v_n}$ as follows:
   for each hyperedge $h \in H$
   Let $v \in h$ be the maximum vertex of h according to σ
   $\chi(B_v) := \chi(B_v) \cup vertices(h)$
3. **for** $i = n$ **to** 2 **do**
   Let $A := \chi(B_{v_i}) - \{v_i\}$
   Let $v_j \in A$ be the next vertex in A following v_i in σ
   $\chi(B_{v_j}) := \chi(B_{v_j}) \cup A$ */* add vertices in A to bucket B_{v_j} */*
   $E := E \cup (B_{v_i}, B_{v_j})$ */* connect buckets B_{v_i}, B_{v_j} */*
4. **return** $\langle (B, E), \chi \rangle$, where $B = \{B_{v_1}, ..., B_{v_n}\}$

Figure 2.10: Algorithm Bucket Elimination [37].

In order to ensure that the width of the resulting generalized hypertree decomposition is small we have to attach as few hyperedges as possible for each tree decomposition vertex. This optimization problem for each tree decompositions vertex p is formalized as set cover problem [32] in the following way:

Given	$V = \{v_1, ..., v_n\}$	a set of vertices
	$H = \{h_1, ..., h_m\}, \forall i : h_i \subseteq V$	a set of hyperedges

For each tree decomposition vertex p find $C \subseteq H$
such that $\chi(p) \subseteq vertices(C)$ and $|C|$ is minimal.

Also the set cover problem is an $\mathcal{NP}$-complete problem [32] but minimal set cover can be formulated as an IP-program [46] and so we are able to obtain exact solutions for small and middle instances for the set cover problem by the help of an IP-solver within a reasonable amount of time. Moreover there is a greedy algorithm [11] for the set cover problem which in practice returns a close-to-optimal solution for many instances.

In [37] McMahan used amongst others the greedy set covering heuristic [11] in order to solve the set cover problems which arise during bucket elimination. The greedy set covering heuristic [11] successively chooses the hyperedge which covers most of the uncovered vertices. Figure 2.11 shows a generalized hypertree decomposition for a hypergraph obtained via "covering" the vertices of one of its tree decompositions.

If the set cover problems that arise during the elimination process are solved exactly, e.g. by using an IP-solver the quality of the resulting tree decomposition is again determined by the underlying elimination ordering. Unlike for tree decompositions it has not been shown whether there exists an elimination ordering that produces an optimal generalized hypertree decomposition for bucket elimination combined with exact set covering. In the next chapter we will prove that at least one elimination ordering will result in a generalized hypertree decomposition of optimal width.

2.5.3 Vertex Elimination

Given a hypergraph $\mathcal{H}$ and an elimination ordering σ, the tree decomposition returned by algorithm Bucket Elimination may also be constructed via a method called *vertex elimination* [44]. Algorithm *Vertex Elimination* (Figure 2.12) describes this technique in pseudo code. Vertex elimination acts on the primal graph of a given hypergraph and eliminates the vertices in order given by an elimination ordering. We say that a vertex is eliminated from a graph when all its neighbors within the graph are connected with each other and the vertex is removed from the graph. The name elimination ordering originates from this process which is also used in order to obtain triangulations of graphs

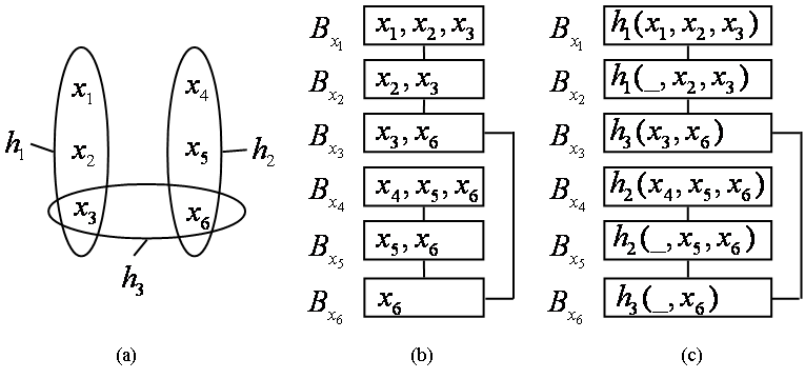

Figure 2.11: A hypergraph (a), a tree decomposition (b) and a generalized hypertree decomposition (c) obtained via bucket elimination from the ordering $\sigma = (x_6, x_5, x_4, x_3, x_2, x_1)$.

[44], [25]. Many heuristic methods for tree decompositions such as [5] and [24] use vertex elimination and also the heuristic methods presented in this thesis will apply this method.

Algorithm: Vertex Elimination

Input: a (constraint) hypergraph $\mathcal{H} = (V, H)$
an elimination ordering $\sigma = (v_1, ..., v_n)$ of the vertices in V
Output: a tree decomposition $\langle T, \chi \rangle$ for $\mathcal{H}$

1. Initially $B = \emptyset$, $E = \emptyset$
   for each vertex v_i introduce an empty bucket B_{v_i}, $\chi(B_{v_i}) := \emptyset$
2. Compute $G = G^*(\mathcal{H})$ the primal graph of $\mathcal{H}$.
3. **for** $i = n$ **to** 1 **do**
   / eliminate vertex v_i */*
   $\chi(B_{v_i}) = \{v_i\} \cup N(v_i)$
   Introduce an edge between each pair of non adjacent neighbors of v_i in G
   Let v_j be the next vertex in $N(v_i)$ following v_i in σ
   $E := E \cup (B_{v_i}, B_{v_j})$
   Remove v_i from G
4. **return** $\langle (B, E), \chi \rangle$, where $B = \{B_{v_1}, ..., B_{v_n}\}$

Figure 2.12: Algorithm Vertex Elimination [44].

Chapter 3

Elimination Orderings - Generalized Hypertree Width

In section 2.5.2 we described how to build a generalized hypertree decomposition for a hypergraph $\mathcal{H}$ from an elimination ordering. In this chapter we prove that a generalized hypertree decomposition for $\mathcal{H}$ whose width equals $ghw(\mathcal{H})$ can be obtained from at least one elimination ordering. This result is particularly important for heuristic methods, because it implies that the set of all elimination orderings for a hypergraph $\mathcal{H}$ is an appropriate search space for generalized hypertree width.

In [3] F. Bacchus proves that the elimination width of a hypergraph equals its treewidth, which implies that a tree decomposition of width equal to the treewidth can be obtained from an elimination ordering. Our proof is in fact a modification of the proof provided by F. Bacchus, thus during the rest of the chapter we will clarify in detail which ideas have been adopted from [2] and [3], how we have modified and extended them and which work has been done by us.

3.1 Problem Description

Definition 16. Let $\mathcal{H} = (V, H)$ be a hypergraph and let $\sigma = (v_1, ..., v_n)$ be an ordering of the vertices in V, where v_i is the $i - th$ element in the ordering. This induces a sequence of hypergraphs $\mathcal{H}_n$,$\mathcal{H}_{n-1}$, ...,$\mathcal{H}_1$ where $\mathcal{H}_n = \mathcal{H}$ and $\mathcal{H}_{i-1}$ is obtained from $\mathcal{H}_i$ as follows: all hyperedges in $\mathcal{H}_i$ containing v_i are merged into one hyperedge and then v_i is removed. Thus the underlying vertices of $\mathcal{H}_{i-1}$ are $v_1, ..., v_{i-1}$. The set $clique(v_i, \sigma, \mathcal{H})$ denotes the set containing v_i and all vertices that are adjacent to v_i in $\mathcal{H}_i$ and the collection $cliques(\sigma, \mathcal{H}) := \{clique(v_i, \sigma, \mathcal{H}) \mid v_i \in V\}$ contains all sets that are produced by

removing the vertices of $\mathcal{H}$ in the order given by σ. (Parts of this definition were taken almost verbatim from [3].)

Definition 17. The width of $C \subseteq V$ in $\mathcal{H}$, $width(C, \mathcal{H})$, is the size of the smallest subset of hyperedges in $\mathcal{H}$ such that each vertex in C is contained in at least one hyperedge of the subset. The width of $\mathcal{H}$ under σ is the maximum width of a set in $cliques(\sigma, \mathcal{H})$, $width(\sigma, \mathcal{H}) := \max_{C \in cliques(\sigma, \mathcal{H})} width(C, \mathcal{H})$.

The elimination process in definition 16 produces the same labels as bucket elimination. As a consequence the $width(\sigma, \mathcal{H})$ is the width of the generalized hypertree decomposition produced by bucket elimination if the set cover problem for each vertex is solved exactly. Note that the labels obtained via the process in definition 16 are also the same labels as returned from vertex elimination (section 2.5.3), because the adjacency relations in the hypergraphs produced by the process in definition 16 are equivalent to the adjacency relations within the regular graphs in each elimination step of vertex elimination.

Now the question arises whether there is an elimination ordering such that $width(\sigma, \mathcal{H}) = ghw(\mathcal{H})$. If the answer for that question is yes we might obtain the generalized hypertree decomposition of smallest width (ghw) via bucket elimination or vertex elimination if the arising set cover problems are solved exactly.

3.2 The Leaf Normal Form for Tree Decompositions

Definition 18 (Leaf Normal Form [2])**.** A *tree decomposition* $TD = \langle T, \chi \rangle$ for a hypergraph $\mathcal{H} = (V, H)$ is in *leaf normal form* if the following two conditions hold on $TD = \langle T, \chi \rangle$.

1. there is a one-to-one mapping $leaf : H \rightarrow leaves(T)$ between the hyperedges of hypergraph $\mathcal{H}$ and the leaf vertices of tree decomposition $TD = \langle T, \chi \rangle$ such that for each hyperedge $h \in H$ it holds that $\chi(leaf(h)) = h$.
2. each internal vertex $p \in vertices(T)$ has variable Y in its label iff p lies on a path between two leaves with Y in their labels.

The definition of tree decompositions in leaf normal form was taken almost verbatim from definition for tree decompositions for hypergraphs in [2]. This definition in [2] requires additionally that a tree decomposition has to be a binary tree. This requirement has been dropped in definition 18 because for our purpose a tree decomposition in leaf normal form need not be a binary tree. Furthermore F. Bacchus proposes two

rules for transforming arbitrary tree decompositions into tree decompositions satisfying his definition. The first rule is used for transforming an arbitrary tree decomposition into a binary one and has no impact on our considerations. The second rule describes how to introduce leaf vertices, corresponding to hyperedges, which are needed for the one-to-one mapping between the leaves of the tree decomposition and the hyperedges of the hypergraph. Algorithm *Transform Leaf Normal Form* (Figure 3.1) uses this second transformation rule in step 2.

We claim that the following algorithm is able to transform a tree decomposition $TD = \langle T, \chi \rangle$ into a tree decomposition in leaf normal form $TD' = \langle T', \chi' \rangle$ such that for each vertex $p' \in \mathit{vertices}(T')$ there is a vertex $p \in \mathit{vertices}(T)$ with $\chi'(p') \subseteq \chi(p)$.

Algorithm: Transform Leaf Normal Form

Input:	hypergraph $\mathcal{H} = (V, H)$, tree decomposition $TD = \langle T, \chi \rangle$ for $\mathcal{H}$
Output:	a tree decomposition in leaf normal form $TD' = \langle T', \chi' \rangle$ of hypergraph $\mathcal{H}$ satisfying: $\forall p' \in \mathit{vertices}(T')\ \exists p \in \mathit{vertices}(T) : \chi'(p') \subseteq \chi(p)$

1. Initially $TD' := TD, T' := T$, $\forall p \in \mathit{vertices}(T) : \chi'(p) := \chi(p)$.
2. For each hyperedge $h \in H$ we introduce a new leaf l_h and connect it to a vertex p that has already been in T with $h \subseteq \chi'(p) = \chi(p)$. The introduced leaf is labeled with the variables of the corresponding hyperedge, $\chi'(l_h) := h$, and we map hyperedge h to the new leaf l_h, $\mathit{leaf}(h) := l_h$.
3. While there is a leaf $l \in \mathit{leaves}(T')$ that is not reached by the mapping leaf we delete l.
4. For each inner vertex $p' \in \mathit{vertices}(T')$ and for each variable $Y \in \chi'(p')$ we delete Y from $\chi'(p')$ if p' is not on a path between two leaves with Y in their labels.

Figure 3.1: Algorithm Transform Leaf Normal Form.

Example 7. Figure 3.2 shows a hypergraph (a) and its tree decomposition (b). Figure 3.3 visualizes the leaves attached to the tree decomposition by step 2. of algorithm Transform Leaf Normal Form as dashed boxes. The crossed out leaf was deleted during step 3. Figure 3.4 shows the tree decomposition obtained by deleting variables in step 4. This tree decomposition is already in leaf normal form. Crossed out variables were deleted in step 4.

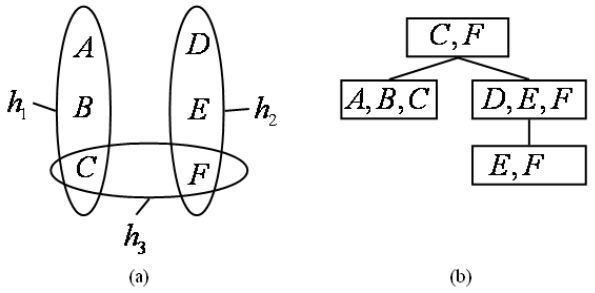

Figure 3.2: Hypergraph (a) and its tree decomposition (b)

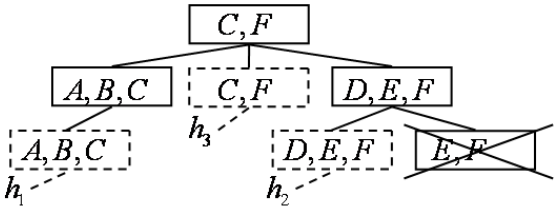

Figure 3.3: Tree Decomposition after step 3. of algorithm Transform Normal Leaf Form

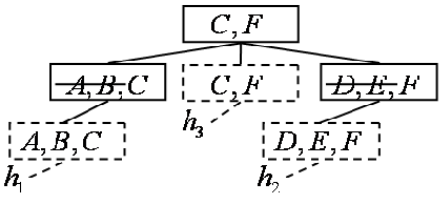

Figure 3.4: Tree Decomposition in leaf normal form after step 4.

3.2.1 Correctness Proof for Algorithm Transform Leaf Normal Form

Within this section we intend to prove that algorithm Transform Leaf Normal Form transforms an arbitrary tree decomposition $TD = \langle T, \chi \rangle$ into a tree decomposition $TD' = \langle T', \chi' \rangle$ in leaf normal form such that for each vertex $p' \in \mathit{vertices}(T')$ there is a vertex $p \in \mathit{vertices}(T)$ with $\chi'(p') \subseteq \chi(p)$. The lemmas and the theorem within this subsection as well as their proofs have been developed entirely by ourselves.

Definition 19. Let $\mathcal{H}$ be a hypergraph and $\langle T, \chi \rangle$ be a tree decomposition for $\mathcal{H}$. For each variable Y in $\mathcal{H}$ the set T_Y consists of the vertices of T which contain Y in their associated χ-set.

Lemma 3. *After step 1. for each vertex $p' \in \mathit{vertices}(T)'$ there is a vertex $p \in \mathit{vertices}(T)$ with $\chi'(p') \subseteq \chi(p)$.*

Proof. Obviously, after step 1. TD' and TD are the same tree decompositions of $\mathcal{H}$ and thus the lemma holds. □

Lemma 4. *After step 3. TD' is a tree decomposition of hypergraph $\mathcal{H} = (V, H)$.*

Proof. In step 2. and step 3. we add and delete leaves from the tree T'. Thus after step 3. T' is still a tree.

Furthermore we have to show that first and the second condition for tree decompositions of hypergraphs are satisfied by TD' after step 3.

In step 2. for each hyperedge $h \in H$ we introduce a leaf vertex l_h with $\chi'(l_h) = h$ and connect it to a vertex p, with $h \subseteq \chi'(p)$, which has already been in TD . Note that p must exist since TD is a tree decomposition of hypergraph $\mathcal{H}$ satisfying the first condition for tree decompositions of hypergraphs. These leaves are not deleted during step 3. Thus after step 3. for each $h \in H$ the vertex $\mathit{leaf}(h) \in \mathit{vertices}(T')$ satisfies $h = \chi'(\mathit{leaf}(h)) \subseteq \chi'(\mathit{leaf}(h))$. We conclude that the first condition for tree decompositions of hypergraphs is satisfied by TD' after step 3.

Assume that the connectedness condition for tree decomposition is satisfied by TD' before but violated after a new leaf l_h and a new edge (l_h, p) have been introduced in step 2. We know that p is chosen such that $h = \chi'(l_h) \subseteq \chi'(p)$. When connecting the leaf l_h to p we connect also the leaf l_h to subtree T'_Y, for each variable $Y \in \chi(l_h)$, and T'_Y is still a tree afterwards. Thus the connectedness condition cannot be violated after any action taken in step 2.

Therefore it must be that the connectedness condition for tree decompositions is satisfied by TD' before but violated after a leaf l has been deleted in step 3. Then after the deletion of l there must be a variable Y such that all vertices containing Y in their

labels don't form a connected subtree of T'. We know that all vertices that contain Y in their labels form the subtree T'_Y before the deletion of l.

If $Y \notin \chi'(l)$ then the deletion of l doesn't change T'_Y and T'_Y is still a tree after the deletion of l.

If $Y \in \chi'(l)$ then by deleting the leaf l from T' we are also deleting the leaf l from the subtree T'_Y and thus all vertices that contain Y in their labels must form a subtree after the deletion of l.

We conclude that after step 3. TD' is a valid tree decomposition of hypergraph $\mathcal{H}$. □

Lemma 5. *After step 3. the mapping $leaf : H \rightarrow leaves(T')$ is a one-to-one mapping such that for each hyperedge $h \in H$ it holds that $\chi'(leaf(h)) = h$.*

Proof. In step 2. we define a one-to-one mapping between the hyperedges of H and the introduced leaves such that for each hyperedge $h \in H$ it holds that $\chi'(leaf(h)) = h$. In step 3. we delete only those leaves in TD' that are not reached by the mapping $leaf$. Thus after step 3. the leaves of T' are those that have been introduced during step 2. We conclude that after step 3. $leaf$ is a one-to-one mapping between the hyperedges of $\mathcal{H}$ and $leaves(T')$. □

Lemma 6. *After step 3 for each vertex $p' \in vertices(T')$ there is a vertex $p \in vertices(T)$ with $\chi'(p') \subseteq \chi(p)$.*

Proof. We know from Lemma 3 that after step 1. for each $p' \in vertices(T')$ there is a $p \in vertices(T)$ with $\chi'(p') \subseteq \chi(p)$.

In step 2. for each hyperedge $h \in edges(H)$ we introduce a leaf l_h with $\chi'(l_h) = h$ and connect l_h to a vertex p that has already been in T with $h \subseteq \chi'(p) = \chi(p)$. Thus for each newly introduced leaf l_h there is a vertex $p \in vertices(T)$ such that $h = \chi'(l_h) \subseteq \chi'(p) = \chi(p)$. Therefore the condition holds also for the newly introduced leaves in T'.

In step 3. we only delete vertices and the remaining vertices still satisfy the property since their labels are not changed.

We conclude that after step 3. for each $p' \in vertices(T')$ there is a $p \in vertices(T)$ with $\chi'(p') \subseteq \chi(p)$. □

Lemma 7. *After step 4. the mapping $leaf : H \rightarrow leaves(T')$ is a one-to-one mapping such that for each hyperedge $h \in H$ it holds that $\chi'(leaf(h)) = h$.*

Proof. From Lemma 5 we know that after step 3. $leaf$ is a one-to-one mapping between the hyperedges of hypergraph $\mathcal{H}$ and leaves of T'. Step 4. alters the labels of the inner vertices of T' but the number of leaves as well as the labels of the leaves remain unchanged. Thus after step 4. the mapping $leaf : H \rightarrow leaves(T')$ is still a one-to-one mapping such that for each hyperedge $h \in H$ it holds that $\chi'(leaf(h)) = h$. □

Lemma 8. *After step 4. TD' is a tree decomposition of hypergraph $\mathcal{H}$.*

Proof. From Lemma 4 we know that TD' is a tree decomposition after step 3.

In step 4. the labels of the vertices of T' are altered but the structure of T' remains unchanged. Thus T' is also a tree after step 4.

Additionally we have to show that the first condition and the second condition for tree decompositions of hypergraphs are satisfied by TD' after step 4.

From Lemma 7 we know that after step 4. the mapping $leaf : H \rightarrow leaves(T')$ is a one-to-one mapping such that for each hyperedge $h \in H$ it holds that $\chi'(leaf(h)) = h$. We conclude that the first condition for tree decompositions of hypergraphs is satisfied after step 4. since for each hyperedge $h \in H$ there is a vertex $leaf(h) \in vertices(T')$ with $h \subseteq \chi(leaf(h))$.

Assume that the connectedness condition for tree decompositions is satisfied by TD' after step 3. but violated after step 4. Then after step 4. there must be a variable Y such that the vertices containing Y in their labels don't form a subtree of T'. Thus the vertices containing Y in their labels can be partitioned into two non-empty sets, C_1 and C_2, such that each vertex in C_1 is not adjacent to any vertex in C_2.

If there isn't any leaf in C_1 then C_1 contains only inner vertices of T'. Let $p' \in C_1$ be such an inner vertex. We know that $Y \in \chi'(p')$ since $p' \in C_1$. Thus p' must lie on a unique path P between two leaves, l_1 and l_2, containing Y in their labels, otherwise Y would have been deleted from $\chi'(p')$ in step 4. Note that in step 4. Y hasn't been deleted from the label of any vertex in P since P is a path between two leaves containing Y. l_1 and l_2 must lie in C_2, according to our assumption that there is no leaf in C_1.

If all vertices in path P contain Y in their labels then each of the vertices in P either is in C_1 or C_2 and there must be at least one edge between a vertex of C_1 and a vertex of C_2, a contradiction to our assumption that such an edge doesn't exist.

Therefore it must be that there is at least one vertex on the path between l_1 and p' which does not contain Y in its label and that there is at least one vertex on the path between l_2 and p' which does not contain Y in its label. Since Y hasn't been deleted from the label of any vertex in P in step 4. and there isn't any other path different from P between l_1 and l_2 in tree T' the connectedness condition for variable Y has already been violated before step 4. which contradicts Lemma 4.

Thus it must be that there is at least one leaf l_1 in C_1 and by applying the same argumentation as above to C_2 we get that there is at least one leaf l_2 in C_2. Since T' is a tree there must be a unique path P between l_1 and l_2. Again, in step 4. Y hasn't been deleted from the label of any vertex in P. If each vertex in P contains Y in its label there must be at least one edge between a vertex in C_1 and a vertex in C_2, a contradiction to our assumption that such an edge doesn't exist. If there is a vertex in P that does not contain Y in its label then it must be that the connectedness condition for variable Y has already been violated before step 4. which contradicts Lemma 4.

We conclude that after step 4. TD is a tree decomposition of hypergraph H.

□

Lemma 9. *After step 4. each internal vertex $p' \in vertices(T')$ has Y in its label iff p' lies on a path between two leaves with Y in their labels.*

Proof. Necessity follows directly from step 4. Assume there is a vertex p' in $vertices(T')$ with $Y \in \chi'(p')$ and p' does not lie on a path between two leaves with Y in their labels then Y must have been deleted from p' in step 4. This contradicts our assumption that $Y \in \chi'(p')$ after step 4.

Sufficiency. Assume that p' lies on a path between two leaves with Y in their labels and $Y \notin \chi'(p')$. Since T' is a tree there is a unique path between two of its leaves. p' lies on a unique path between the two leaves containing Y in their labels and p' doesn't contain Y. Then $TD' = \langle T', \chi' \rangle$ doesn't satisfy the connectedness condition after step 4. which has been proven to hold in Lemma 8 . □

Lemma 10. *After step 4. for each $p' \in vertices(T')$ there is a $p \in vertices(T)$ with $\chi'(p') \subseteq \chi(p)$.*

Proof. We know from Lemma 6 that after step 3. for each $p' \in vertices(T')$ there is a $p \in vertices(T)$ with $\chi'(p') \subseteq \chi(p)$. Since in step 4. variables are only deleted and not added to labels of vertices of T' we conclude that after step 4. for each $p' \in vertices(T')$ there is a $p \in vertices(T)$ with $\chi'(p') \subseteq \chi(p)$. □

Theorem 1. *For every tree decomposition $TD = \langle T, \chi \rangle$ of hypergraph $\mathcal{H}$ there is a tree decomposition $TD' = \langle T', \chi' \rangle$ of hypergraph $\mathcal{H}$ in leaf normal form such that for each $p' \in vertices(T')$ there is a $p \in vertices(T)$ such that $\chi'(p') \subseteq \chi(p)$. Moreover algorithm Transform Leaf Normal Form returns such a tree decomposition.*

Proof. Follows directly from Lemma 7, Lemma 8, Lemma 9 and Lemma 10. □

3.3 From Leaf Normal Forms to Elimination Orderings

In this section we will show that we can derive an elimination ordering σ from a tree decomposition $TD = \langle T, \chi \rangle$ in leaf normal form such that each set within $cliques(\sigma, \mathcal{H})$ is contained within a vertex of TD (see lemma 13).

This result is needed in section 3.4 in order to prove that a generalized hypertree decomposition of $width = ghw$ can be obtained from at least one elimination ordering. The definitions, lemmas and their proofs presented within this section are already implicetly given within the proof for lemma 1 in [3]. Lemma 1 in [3] says that for each tree decomposition for a hypergraph $\mathcal{H}$ of width w there is an elimination ordering π such that the induced width of H under π is at most w. This result implies that an optimal tree decomposition may be obtained from at least one elimination ordering. We partitioned the proof for Lemma 1 in [3] into several definitions and lemmas and modified the proof of lemma 1 in [3] in order to ensure a better understanding for the reader. Nevertheless, the results presented within this section have already been proved by F. Bacchus in [3].

Definition 20. Let $\mathcal{H} = (V, H)$ be a hypergraph and let $\sigma = v_1, ..., v_n$ be an ordering of the vertices in V. We use the notation $x <_\sigma y$ in order to indicate that x precedes y in the ordering. (Note that if $x <_\sigma y$ then y will be removed from $\mathcal{H}$ before x.)

Definition 21. Let $TD = \langle T, \chi \rangle$ be a tree decomposition of hypergraph $\mathcal{H} = (V, H)$ in leaf normal form.

For an arbitrary vertex $p \in vertices(T)$ let

- T_p denote the subtree of T rooted at p
- $vars(p)$ denote the union of the labels of the leaves in T_p
- $depth(p)$ denote the distance from p to the root in T

For an arbitrary vertex $v \in V$ of hypergraph $\mathcal{H}$ let

- $leaves(v)$ denote the set of leaves in T which contain v in their labels
- $dca(v)$ denote the deepest common ancestor of all the leaves in $leaves(v)$
- $depth(v)$ be the depth of the deepest common ancestor of all leaves containing v, $depth(v) := depth(dca(v))$

This definition was taken almost verbatim from Lemma 1 in [3].

Lemma 11. *Let $TD = \langle T, \chi \rangle$ be a tree decomposition of hypergraph $\mathcal{H} = (V, H)$ in leaf normal form. Then $\forall v \in V \; \forall p \in vertices(T)$*

1. $v \in \chi(dca(v))$
2. $leaves(v) \subseteq vertices(T_{dca(v)})$
3. $v \in \chi(p) \Rightarrow v \in vars(p)$
4. *v does not appear in any label of any vertex outside the subtree $T_{dca(v)}$*

Proof. We know that

1. $dca(v)$, the deepest common ancestor of the leaves containing v, must lie on a path between two leaves with v in their labels. Since TD is in leaf normal form it must be that $v \in \chi(dca(v))$.
2. $dca(v)$ is the deepest common ancestor of the leaves of T containing v. Thus the subtree rooted at $dca(v)$ obviously contains all such leaves.
3. TD is in leaf normal form. Thus if $v \in \chi(p)$ there must be a path between two leaves containing v and at least one of those two leaves must be in subtree T_p rooted at p.
4. $dca(v)$ is the deepest common ancestor of all leaves with v in their labels. If any vertex p outside T_p contains v then p must lie on a path between two leaves with v in their labels. If both leaves are in T_p then either p is in T_p or $dca(v)$ isn't a common ancestor of all leaves containing v, a contradiction. If at least one leaf is outside T_p then again $dca(v)$ isn't a common ancestor of all leaves containing v, a contradiction.

□

Lemma 12. *Let $\mathcal{H} = (V, H)$ be a hypergraph, $TD = \langle T, \chi \rangle$ a tree decomposition of H in leaf normal form and let σ be any ordering of the vertices in V such that if $depth(v) < depth(w)$ then $v <_\sigma w$. If $y \in vars(dca(x))$ and $y <_\sigma x$ then $y \in \chi(dca(x))$*

Proof. Assume by way of contradiction that there is a tree decomposition $TD = \langle T, \chi \rangle$ of a hypergraph $\mathcal{H}$ in leaf normal form, an ordering σ such that if $depth(v) < depth(w)$ then $v <_\sigma w$ and two vertices x and y such that $y \in vars(dca(x))$ and $y <_\sigma x$ but $y \notin \chi(dca(x))$. From $y \in vars(dca(x))$ we know that there is a leaf l in $T_{dca(x)}$ containing y and from $y <_\sigma x$ we know that $depth(dca(y)) \leq depth(dca(x))$. Thus it must be that either $dca(y) = dca(x)$ or $dca(y) \notin vertices(T_{dca(x)})$.

If $dca(y) = dca(x)$ we know that $y \notin \chi(dca(y)) = \chi(dca(x))$, a contradiction to Lemma 11.

If $dca(y) \notin vertices(T_{dca(x)})$ then $dca(x)$ lies on the unique path P between the leaf l in $T_{dca(x)}$ and $dca(y)$. Since $y \in \chi(dca(y))$ and $y \notin \chi(dca(x))$, P violates the connectedness condition for tree decompositions of hypergraphs, a contradiction to our assumption that $TD = \langle T, \chi \rangle$ is a tree decomposition of hypergrah $\mathcal{H}$. □

Lemma 13. *Let $\mathcal{H} = (V, H)$ be a hypergraph, $TD = \langle T, \chi \rangle$ a tree decomposition of $\mathcal{H}$ in leaf normal form and let σ be any ordering of the vertices in V such that if $depth(y) < depth(x)$ then $y <_\sigma x$. Then for each $v \in V$ it holds that $clique(v, \sigma, \mathcal{H}) \subseteq \chi(dca(v))$.*

Proof. By induction:

Basis: v_n is removed.

When v_n is eliminated $clique(v_n, \sigma, \mathcal{H})$ contains v_n and all vertices that are initially adjacent to v_n. From Lemma 11 we know that $v_n \in \chi(dca(v_n))$ and since $leaves(v_n) \subseteq vertices(T_{dca(v_n)})$ we know that $T_{dca(v_n)}$ contains all leaves with v_n in their labels. According to the one-to-one mapping $leaf$ between the hyperedges of $\mathcal{H}$ and the leaves of T, all hyperedges in which v_n appears in $\mathcal{H}$ can be found in $leaves(v_n)$.

Thus for each vertex y initially adjacent to v_n we have $y \in vars(dca(v_n))$ and since $y <_\sigma v_n$ it must be that $y \in \chi(dca(v_n))$ (Lemma 12). We conclude that $clique(v_n, \sigma, \mathcal{H}) \subseteq \chi(dca(v_n))$.

Induction step: We assume that $clique(v, \sigma, \mathcal{H}) \subseteq \chi(dca(v))$ for $v \in \{v_{i+1}, ..., v_n\}$. v_i is removed.

When v_i is eliminated $clique(v_i, \sigma, \mathcal{H})$ consists of v_i, of vertices that were initially adjacent to v_i and of vertices that are adjacent to v_i after one of the vertices in $v_{i+1}, ..., v_n$ has been eliminated.

We know that $v_i \in \chi(dca(v_i))$ (Lemma 11).

For each vertex y that was originally adjacent to v_i it must be that y and v_i appear together within an hyperedge of H and thus they are also together within a leaf of T. Since $leaves(v_i) \subseteq vertices(T_{dca(v_i)})$ (Lemma 11) it must be that $y \in vars(dca(v_i))$. We know that $y \in vars(dca(v_i))$ and $y <_\sigma v_i$ thus we conclude that $y \in \chi'(dca(v_i))$ (Lemma 12).

For each vertex y that is adjacent due to the elimination of a vertex v_j with $v_i <_\sigma v_j$, we know that $y, v_i, v_j \in clique(v_j, \sigma, H) \subseteq \chi(dca(v_j))$ according to our induction assumption.

If $dca(v_j) \notin vertices(T_{dca(v_i)})$ then $v_i \notin \chi(dca(v_j))$, since there is no vertex outside the subtree rooted in $T_{dca(v_i)}$ that contains v_i in its label (Lemma 11), a contradiction to $v_i \in \chi(dca(v_j))$.

Thus it must be that $dca(v_j) \in vertices(T_{dca(v_i)})$. Since $y \in \chi(dca(v_j))$ we know that $y \in vars(dca(v_j))$ (Lemma 11). Since $dca(v_j) \in vertices(T_{dca(v_i)})$ it must be that $vars(dca(v_j)) \subseteq vars(dca(v_i))$ and thus $y \in vars(dca(v_i))$. From $y \in vars(dca(v_i))$ and $y <_\sigma v_i$ we conclude that $y \in \chi(dca(v_i))$ (Lemma 12). □

Example 8. Figure 3.5 shows a tree decompositions in leaf normal form and the deepest common ancestors (dca) for each variable. For the variables in the elimination ordering σ it holds that if $depth(y) < depth(x)$ then $y <_\sigma x$. Figure 3.6 shows a tree decomposition (a) derived from σ via bucket elimination/vertex elimination. The variables in each vertex are contained by at least one vertex of the original tree decomposition (b).

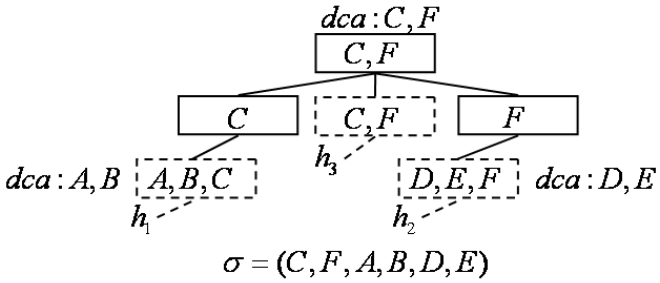

Figure 3.5: Elimination ordering σ derived from deepest common ancestors (dca).

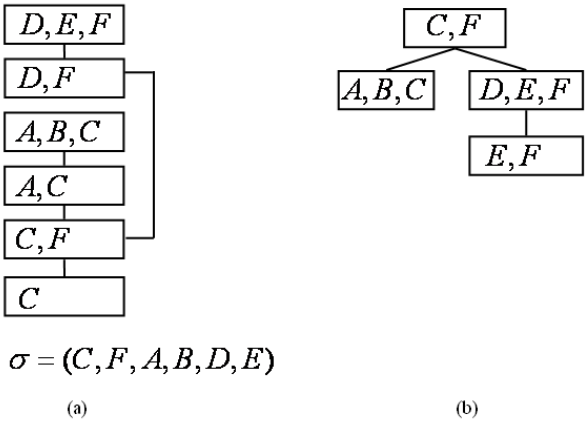

Figure 3.6: Tree decomposition (a) derived from σ and original tree decomposition (b).

3.4 Obtaining an Optimal Generalized Hypertree Decomposition from Elimination Orderings

In this section we prove that for each hypergraph $\mathcal{H}$ there is at least one elimination ordering σ such that the width of $\mathcal{H}$ under σ equals the generalized hypertree width, $width(\sigma, \mathcal{H}) = ghw(\mathcal{H})$. As a consequence of this result our approach to create generalized hypertree decompositions via bucket elimination/vertex elimination is able to create a generalized hypertree decomposition of minimal width. The set of all possible elimination orderings for a hypergraph may be regarded as search space for the generalized hypertree width.

Theorem 2. *Let $\mathcal{H} = (V, H)$ be a hypergraph, $GHD = \langle T, \chi, \lambda \rangle$ be a generalized hypertree decomposition of hypergraph $\mathcal{H}$ and let k be the width of GHD. Then there is an ordering σ of the vertices in V such that the width of $\mathcal{H}$ under σ is at most k, $width(\sigma, \mathcal{H}) \leq k$.*

Proof. As generalized hypertree decomposition GHD is also a tree decomposition of hypergraph $\mathcal{H}$. From Theorem 1 we know that there is a tree decomposition $GHD' = \langle T', \chi' \rangle$ of $\mathcal{H}$ in leaf normal form such that for each $p' \in vertices(T')$ there is a $p \in vertices(T)$ satisfying $\chi'(p') \subseteq \chi(p)$.

According to Lemma 13 there must be an ordering σ of the vertices in V such that for each $C \in cliques(\sigma, \mathcal{H})$ there is a $p' \in vertices(T')$ with $C \subseteq \chi'(p')$. Then for each $C \in cliques(\sigma, \mathcal{H})$ there must also be a $p \in vertices(T)$ with $C \subseteq \chi(p)$ and $width(C, \mathcal{H}) \leq |\lambda(p)|$. $width(C, \mathcal{H})$ was defined to be the size of the smallest subset of hyperedges in $\mathcal{H}$ such that each vertex of C is contained in at least one hyperedge of the subset. Since the hyperedges in $\lambda(p)$ contain all vertices of p, thus also all vertices of C, the cardinality of $\lambda(p)$ may exceed $width(C, \mathcal{H})$, thus $width(C, \mathcal{H}) \leq |\lambda(p)|$.

The width of GHD is the maximum number of hyperedges associated with a vertex in T and the width of GHD is supposed to be k. Since for each set $C \in cliques(\sigma, \mathcal{H})$ there is a $p \in vertices(T)$ with $C \subseteq \chi(p)$ and $width(C, \mathcal{H}) \leq |\lambda(p)|$ we conclude that the width of $\mathcal{H}$ under σ does not exceed the width of GHD, $width(\sigma, \mathcal{H}) \leq k$. □

Theorem 3. *Let $\mathcal{H} = (V, H)$ be a hypergraph. Then there must be an ordering σ of the vertices in V such that $width(\sigma, \mathcal{H}) = ghw(\mathcal{H})$.*

Proof. The generalized hypertree width of a hypergraph $\mathcal{H}$, $ghw(\mathcal{H})$, is defined to be the minimum width over all generalized hypertree decompositions of $\mathcal{H}$. Thus there must be a generalized hypertree decomposition GHD of $\mathcal{H}$ such that $width(\mathcal{H}) = ghw(\mathcal{H})$. From Theorem 2 we know that there must be an ordering σ such that $width(\sigma, \mathcal{H}) \leq ghw(\mathcal{H})$.

In [37] it is shown that given an ordering of the vertices of a hypergraph a generalized hypertree decomposition consisting only of the sets produced by that ordering can be constructed. If $width(\sigma, \mathcal{H}) < ghw(\mathcal{H})$ there would be a generalized hypertree decomposition of width smaller than $ghw(\mathcal{H})$, which contradicts the minimality of $ghw(\mathcal{H})$. Thus it must be that $width(\sigma, \mathcal{H}) = ghw(\mathcal{H})$. □

Chapter 4

An Overview of Heuristic Methods used in This Thesis

In this chapter we give a short description of the heuristic methods which are applied within this master thesis. These heuristic methods are *branch and bound algorithms, A* algorithms* and *genetic algorithms*. Furthermore we are going to review already available heuristic methods for tree decompositions and related problems.

4.1 Branch and Bound Algorithms

A simple but inefficient method for finding the optimal solution for an optimization problem is *exhaustive search*. Exhaustive search checks each solution for a given optimization problem and returns the optimal solution after the whole search space has been visited. Since the search space for an optimization problem is usually very large, exhaustive search may not return the optimal solution within a reasonable amount of time.

A *branch and bound algorithm* tries to overcome this problem by identifying and omitting regions within the search space which don't contain solutions that are better than the best solution we have already found during the search. A branch and bound algorithm "shrinks" the search space by the help of the following two techniques:

1. *Branching*: The overall search space is partitioned into several smaller sub regions. This process is recursively applied on the subregions until we end up in single solutions for the optimization problem. The produced subregions may be visualized as a tree, the *branch and bound tree* or the *search tree*.

2. *Bounding*: For each subregion a lower (minimization problem) or an upper bound (maximization problem) on the value of the best solution within the subregion is computed which is used for narrowing the search space.

In order to discard those regions from the search space that will not lead to better solutions a branch and bound algorithms applies a strategy called *pruning*. For minimization problems this is done by maintaining a global variable upper bound containing the value of the best solution found so far. The upper bound may be initialized with the value of a randomly or heuristically created solution. Whenever a solution of better quality is explored the upper bound will be updated. For each subregion we compute a lower bound on its optimal solution. If the lower bound for a subregion exceeds the current upper bound we may discard this subregion.

Example 9. Figure 4.1 shows a search tree representing all possible elimination orderings for three vertices and visualizes an imaginary execution of a branch and bound algorithm on this tree. The value in a tree node is the lower bound for the subregion represented by the tree rooted at this node. Initially $ub = 20$, then a solution of value 14 is found, thus $ub = 14$. Whenever the lower bound of a node is greater or equal to the current value for *ub* the solutions below this node are excluded from the branch and bound search (dashed nodes).

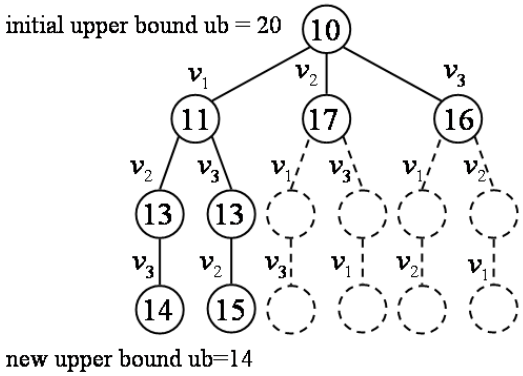

Figure 4.1: Branch and bound search.

Branch and bound algorithms are exact methods. If they are given enough time they will return an optimal solution of the problem they are designed for. A branch and bound

algorithm may also be used as approximation for the best solution. For instance, if we restrict the running time of a branch and bound algorithm then the branch and bound algorithm will return the value of the best solution which has been found until the time limit was exceeded. The pruning power of a branch and bound algorithm depends on the quality of the upper and lower bound heuristics which are applied within the branch and bound algorithm. A key issue for developing branch and bound algorithms for minimization problems, such as finding the treewidth and generalized hypertree width of a hypergraph, is the design of good lower bounds.

4.2 A* Algorithms

A algorithms* (pronounced A star) are graph search algorithms which find an optimal path from a start vertex to a goal. Like branch and bound algorithms also A* algorithms are exact methods. In the previous section we mentioned that a branch and bound algorithm partitions the search space successively into sub regions and this partitioning process is visualized as branch and bound tree. We may transform a branch and bound tree into a graph search problem as shown in Figure 4.2. The start for our search is the root of the tree. The costs for getting from vertices to their successors are associated with the edges. Finally we introduce a goal vertex covering all leaves of the tree.

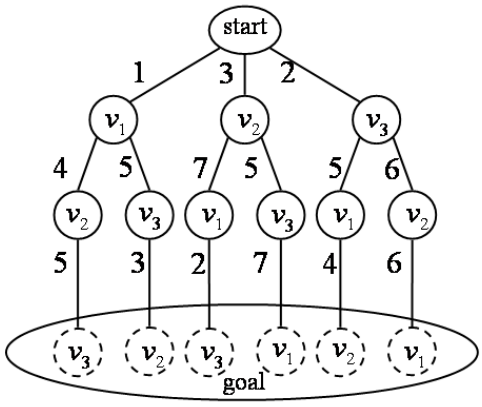

Figure 4.2: Graph search problem derived from branch and bound tree in Figure 4.1.

A* algorithms are *best-first search* methods. A best-first search algorithm selects the next vertex that will be visited within the search according to an evaluation function

$f(n)$, which is an estimate for the costs of a path from the start to the goal containing the vertex n. The vertex with the lowest value for $f(n)$ is chosen to be visited next and its children will be evaluated according to $f(n)$ (Figure 4.3). In practice, the unvisited vertices are stored within a priority queue ordered by their values for $f(n)$. The evaluation function $f(n)$ for A* algorithms is an lower bound for the lowest costs of a path from the start to the goal containing the vertex n. A* algorithms compute $f(n)$ in the following way:

$$f(n) = g(n) + h(n)$$

$g(n)$ denotes the costs for reaching n and $h(n)$ is an estimate for the lowest costs for getting from n to the goal. In order to find the optimal solution it is reasonable to continue the search at the vertex with the lowest value for $f(n) = g(n) + h(n)$ [45]. Moreover it turns out that if the search graph is a tree, such as a branch and bound tree, and if $h(n)$ is an *admissible heuristic*, which means that $h(n)$ never overestimates the lowest costs for reaching the goal from n, the path returned by an A* algorithm will be the optimal solution. The main drawback with A* algorithms is that there may be an exponential number of vertices within its priority queue such that an A* algorithm may run out of memory before it has completed its search.

If we have a branch and bound algorithm for an optimization problem we may obtain an A* algorithm for that problem:

- Our search graph is the modified branch and bound tree (Figure 4.2).
- For $g(n)$ we use the costs of the partial solution represented by vertex n.
- For $h(n)$ we use a lower bound on the value of the partial solutions within the subtree rooted at n.

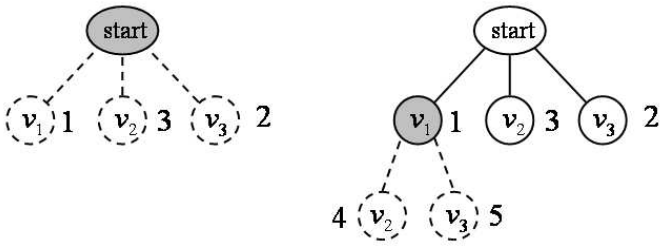

Figure 4.3: Best-first search.

4.3 Genetic Algorithms

Genetic algorithms (GAs) were developed by Holland [31] in the 1970s. They represent a class of *evolutionary algorithms* (EAs). Evolutionary algorithms try to find a good solution for an optimization problem by imitating the principle of evolution. They alter and select individuals from a population of solutions for the optimization problem. In the following we introduce and describe frequently used terms within evolutionary algorithms:

population	... set of candidate solutions
individual	... a single candidate solution
chromosome	... set of parameters that determine the properties of a solution
gene	... single parameter
allele	... concrete value for a parameter
genotype	... all concrete parameter values of a chromosome
phenotype	... all properties of a candidate solution

Figure 4.4 shows the structure of a genetic algorithm [38]. A genetic algorithms tends to optimize the value of an objective function of an optimization problem, in terms of genetic algorithms also called *fitness function*. At the beginning a genetic algorithm creates an initial population containing randomly or heuristically created individuals. These individuals are evaluated and assigned a *fitness* value, which is the value of the fitness function for the solution represented by the individual. The population is evolved over a number of generations until a halting criterion is satisfied. In each generation the population undergoes *selection*, *recombination*, also denoted *crossover*, and *mutation*.

During the process of selection the genetic algorithm decides which individuals from the current population are allowed to enter the next population. This decision is based on the fitness value of the individuals and individuals of better fitness should enter the next population with higher probability than individuals of lower fitness. Not selected individuals are discarded and won't be evolved further.

The process of recombination or crossover combines different properties of several parent solutions within one or more children solutions, also denoted *offsprings*. If only good or the best properties of the parents are combined the resulting child may be fitter than any of its parents. Recombination is the characteristic operator of genetic algorithms. Within the other class of evolutionary algorithms, *evolution strategies* (ES), recombination isn't applied.

During the process of mutation the individuals are slightly altered. Mutation is used to introduce new genetic material into the population.

Genetic Algorithm

$t = 0$
initialize $population(t)$
evaluate $population(t)$

while $\neg terminated$ **do**
  $t = t + 1$
  select $population(t)$ from $population(t-1)$
  recombine $population(t)$
  mutate $population(t)$
  evaluate $population(t)$

Figure 4.4: The structure of a genetic algorithm [38].

In practice parameters are used in order to control the behavior of a genetic algorithm. Typical *control parameters* are mutation rate, crossover rate, population size and parameter for selection techniques. The crossover rate specifies how many individuals undergo crossover during an iteration, the mutation rate specifies the probability that a individual is mutated during an iteration whereas the population size determines how many individuals appear together within the population. Parameters for selection techniques are used to control the degree how much individuals of higher fitness are preferred to individuals of small fitness. The choice of the control parameters has a crucial effect on the behavior of the algorithm.

4.3.1 Problem Representation

When designing a genetic algorithm for a given optimization problem we have to think about how a solution for the problem may be represented within the genetic algorithm. Popular problem representations are bit-strings, permutations, finite state machines and symbolic expressions [38]. We intend to develop genetic algorithms for tree and generalized hypertree decompositions. In chapter 3 we have seen that the set of all elimination orderings, which is the set of all permutations of the vertices of a hypergraph, represents a suitable search space for the optimal tree or generalized hypertree decomposition. Therefore we will describe standard crossover and mutation operators for permutations in the following two sections.

4.3.2 Crossover Operators for Permutations

The crossover operators described within this section are taken from [36]. Figure 4.5 gives an overview of the presented crossover operators.

Partially-Mapped Crossover (PMX)

The partially-mapped crossover operator determines a crossover area within the parent solutions by randomly selecting two positions within the permutations. The genes in the crossover areas define a mapping. The crossover areas are exchanged. If a gene outside the crossover area appears in the exchanged area it is replaced by the value defined by the mapping.

Cycle Crossover (CX)

The cycle crossover operator can be described as follows. If the first parent is written above the second parent we can consider the ordering as a single permutation. Then we have to determine the first cycle of that permutation. Within the offspring the elements of the cycle have the same position as in the first parent. The other elements have the same positions as in the second parent.

Order Crossover (OX1)

The order crossover operator determines a crossover area within the parents by randomly selecting two positions within the permutation. The genes in the crossover area of the parent are copied to the offspring. Starting at the end of the crossover area all genes outside the area are inserted in the same order in which they occur in the other parent.

Order-Based Crossover (OX2)

The order-based crossover operator selects at random several positions in the parent string by tossing a coin for each position. The genes of one parent at these positions are deleted in the other parent. Afterwards they are reinserted in the order of the other parent.

Position-Based Crossover (POS)

The position-based crossover operator also starts with selecting a random set of positions in the parent strings by tossing a coin for each position. The elements at the selected

positions are exchanged between the parents in order to create the offsprings. The genes missing after the exchange are reinserted in the order of the other parent.

Alternating-Position Crossover (AP)

The alternating-position crossover operator creates an offspring by selecting alternately the next element of the first parent and the next element of the second parent, omitting the elements already present in the offspring.

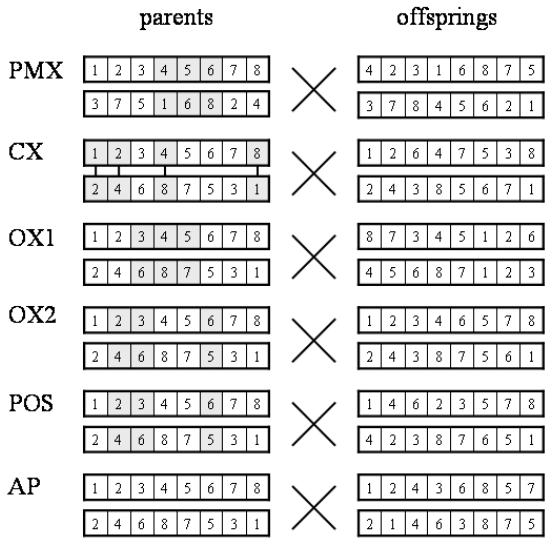

Figure 4.5: Crossover operators for permutations.

4.3.3 Mutation Operators for Permutations

The mutation operators described within this section are taken from [36]. Figure 4.6 gives an overview of the presented mutation operators.

Displacement Mutation Operator (DM)

The displacement operator selects a random substring of the solution. This substring is moved to a random position of the solution.

Exchange Mutation Operator (EM)

The exchange mutation operator randomly selects two elements in the solution and exchanges them.

Insertion Mutation Operator (ISM)

The insertion mutation operator randomly chooses an element in the solution and moves it to a randomly selected position.

Simple-Inversion Mutation operator (SIM)

The simple-inversion mutation operator selects randomly two cutpoints in the string that represents the individual and reverses the substring between these two cutpoints.

Inversion Mutation Operator (IVM)

The inversion mutation operator selects a substring, removes it from the string and randomly inserts it at a randomly selected position in reversed order.

Scramble Mutation Operator (SM)

The scramble mutation operator selects a random substring and reorders the elements in it at random.

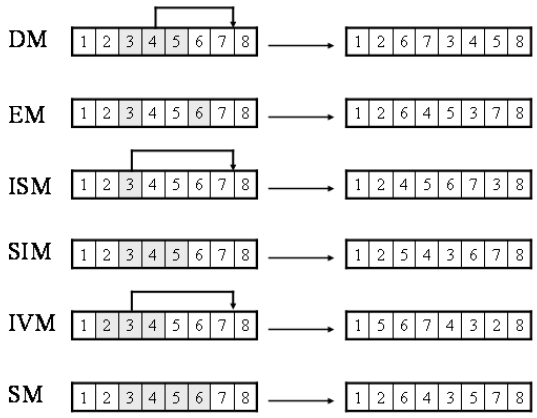

Figure 4.6: Mutation operators for permutations.

4.4 A Review of Existing Branch and Bound Algorithms for Tree Decompositions

Within this section we give a description of two branch and bound algorithms for computing the treewidth of regular graphs, algorithm *BB-tw* introduced by Bachoore and Bodlaender in [5] and algorithm *QuickBB* introduced by Gogate and Dechter in [24]. Our review will concentrate only on those ideas in [5] and [24] which will be adopted for the development of new heuristic methods for tree decompositions and generalized hypertree decompositions in this thesis later.

4.4.1 The Basic Branch and Bound Algorithm

Both branch and bound algorithms BB-tw [5] and QuickBB [24] compute the treewidth of a regular graph based on elimination orderings. Figure 4.1 shows a branch and bound tree representing all possible elimination orderings for three vertices which is searched by the branch and bound algorithms. At the beginning the branch and bound algorithms create a solution heuristically. The width of that solution acts as the first upper bound on the treewidth and is stored within the variable *ub*. Starting at the root the tree is searched in depth-first-search fashion. If the search visits a new search node a vertex is eliminated from the graph. In each node of the branch and bound tree the following values are computed:

- the degree of the vertex that is eliminated in the branching step, denoted d.
- the maximum degree of all eliminated vertices along the path from the root to the current node in the tree, denoted g. g is the width of the partial elimination ordering consisting of the vertices on the path from the root to the current node and is computed as $g = max(d, g')$, where g' denotes the width of the partial solution represented by the parent node of the current search node.
- a lower bound h on the treewidth of the graph obtained after eliminating all vertices of the path from the root to the current node in the tree.
- a lower bound on the width of the tree decomposition of the original graph which may be reached when continuing the search in the subtree rooted at the current search node, $f = max(g, h)$.

If $f \geq ub$ then the subtree rooted at the current node will be discarded from the search. The search continues at the parent node of the current search node. If the search ends in a leaf of the branch and bound tree and the width of the solution represented by that leaf is smaller than *ub*, then *ub* is set to this new upper bound on the treewidth.

The above description refers to the branch and bound search of algorithm QuickBB [24], but also algorithm BB-tw [5] applies the same search principle as algorithm QuickBB.

4.4.2 Upper and Lower Bound Heuristics

For computing a first upper bound algorithm QuickBB uses the min-fill heuristic. BB-tw uses heuristics introduced in [4] for getting the first upper bound. The following description of the min-mill heuristic is taken almost verbatim from [24].

min-fill heuristic: Order the vertices from 1 to n as follows. First select a vertex v which adds the least number of edges when eliminated from the graph and place it at position n. Eliminate v from the graph and introduce an edge between each pair of not adjacent neighbors of v. Now select any vertex that adds the least number of edges when eliminated and place it at the next position in the ordering. Repeat the process breaking ties arbitrarily.

For computing a lower bound on the treewidth of a graph Gogate and Dechter propose a new heuristic named minor-min-width [24] which is implemented in QuickBB. Bodlaender et al. developed the same heuristic independently in [9] and named it MMD+(least-c). Algorithm **minor-min-width** in Figure 4.7 was taken from [24] and presents the the lower bound heuristic in pseudo code notation.

Algorithm: minor-min-width

Input: a graph G
Output: a lower bound on the treewidth of G

1. $lb = 0$
2. **repeat**
   (a) Contract the edge between a minimum degree vertex v and $u \in N(v)$ such that the degree of u is minimum in $N(v)$ to form a new graph in G'. Ties are broken randomly.
   (b) $lb = max(lb, degree_G(v))$.
   (c) Set G to G'.
3. **until** no vertices remain in G.
4. **return** lb

Figure 4.7: Algorithm minor-min-width [24].

Bachoore and Bodlaender use the degeneracy heuristic [9] and the Ramachandra-

murthi γ parameter of the graph [41] for a getting lower bound in BB-tw. We present another heuristic taken from [35] which we will use as lower bound heuristic for the treewidth of graphs. The heuristic is denoted *minor-γ_R* within this thesis and is presented below in Figure 4.8.

Algorithm: minor-γ_R

Input: a graph G
Output: a lower bound on the treewidth of G

1. $lb = 0$
2. **repeat**
   (a) Sort the vertices in G according to their degrees in ascending order.
   (b) Determine the first vertex v in this sequence that is not adjacent to all its predecessors.
   (c) $\gamma_R = degree_G(v)$
   (d) Contract the edge between v and $u \in N(v)$ such that the degree of u is minimum in $N(v)$ to form a new graph in G'. Ties are broken randomly.
   (e) $lb = max(lb, \gamma_R)$.
   (f) Set G to G'.
3. **until** no vertices remain in G.
4. **return** lb

Figure 4.8: Algorithm minor-γ_R [35].

4.4.3 Reduction Techniques for Graphs

In [8] Bodlaender et al. present techniques for removing vertices from graphs without changing their treewidth. Both algorithms BB-tw [5] and QuickBB [24] apply these rules in order to reduce the search space for the branch and bound search.

Simplicial Vertices

Definition 22 (Simplicial vertex [24])**.** A vertex v of graph G is *simplicial* if all its neighbors induce a clique in G.

In [8] it is shown that when removing a simplicial vertex from a graph the treewidth of the resulting graph doesn't exceed the treewidth of the original graph. Thus whenever

a simplicial vertex appears in the graph during the branch and bound search the simplicial vertex is removed in the next step.

Strongly Almost Simplicial Vertices

Definition 23 (Almost simplicial vertex [24])**.** A vertex v of graph G is *almost simplicial* if all but one of its neighbors induce a clique.

Definition 24 (Strongly Almost Simplicial vertex [5])**.** An almost simplicial vertex v of graph G is *strongly almost simplicial* if the degree of v in G doesn't exceed any lower bound on the treewidth of G.

In [8] it is shown that when removing a strongly almost simplicial vertex from a graph the treewidth of the resulting graph doesn't exceed the treewidth of the original graph. Thus whenever a strongly almost simplicial vertex appears in the graph during the branch and bound search the strongly almost simplicial vertex is removed in the next step.

4.4.4 Reducing the Search Space

In [24] Gogate and Dechter define a subset of the set of all possible elimination orderings of a graph, denoted *treewidth elimination set* as follows:

Definition 25 (Treewidth elimination set [24])**.** Let P be the set of all possible orderings $\sigma = (v_1, ..., v_n)$ of vertices of a graph G constructed in the following manner. Select an arbitrary vertex and place it at position n. For $i = n-1$ to 1, if there exists a vertex v such that $v \notin N(v_{i+1})$, make it simplicial and remove it from G. Otherwise, select an arbitrary vertex v and remove it from G. Place v at position i. P is called the *treewidth elimination set* of G.

Moreover in [24] they present a lemma implying that it is sufficient to consider only elimination orderings within the treewidth elimination set in order to get the treewidth of a graph. Therefore algorithm QuickBB [24] uses only the elimination orderings in the treewidth elimination set for the branch and bound search.

4.4.5 Pruning Rules

Suppose that we are in a node of the branch and bound tree, let g be the width of the partial solution represented by that node and let n' denote the number of vertices that haven't been eliminated yet. We know that the width of a solution within the subtree

rooted at the current search node may be at most $w = max(g, n' - 1)$. If w is smaller than the width of the best solution found so far, $w < ub$, we know that we will find a better solution within the subtree rooted at the current search node. There are two cases. If $n' - 1 \leq g$ then we don't have to continue the search within the subtree. If $n' - 1 > g$ then at least one solution within the subtree will lead to a new upper bound of at most $n' - 1$ but we have to continue the search within the subtree for finding its best solution. This observation was made in [5] and leads to the following pruning rule for each node of the branch and bound search.

Pruning Rule 1 (PR 1) [5]
compute $w := max(g, n' - 1)$
if $w < ub$ then $ub = w$
if $n' - 1 \leq g$ then exclude the subtree rooted at the current node from the search

In [24] Gogate and Dechter present several pruning rules for reducing the search space of algorithm QuickBB. Bachoore and Bodlaender formulated another pruning rule in [5] and claimed that this rule will have similar pruning power.

Pruning Rule 2 (PR 2) [5]

Suppose v and w are successive vertices in an elimination ordering σ, and v and w are not adjacent or v and w are adjacent and each vertex has a neighbor preceding v, w in σ that is not a neighbor of the other in the graph obtained by eliminating the vertices in σ until v. Then the ordering σ', obtained by swapping v and w in σ, has the same width as σ. Thus, we prune the search tree as follows: for such a pair of vertices v, w, when we have looked at a branch representing the elimination orderings ending with $w, v, x_i, ..., x_n$ we prune the branch representing the orderings ending with $v, w, x_i, ..., x_n$.

Pruning Rule 2 was taken almost verbatim from [5]. Figure 4.9(a) shows a part of a branch and bound tree affected by this pruning rule. From Figure 4.9(b) we see why this pruning technique may be applied to the branch and bound search. The graph obtained by eliminating v and w is always the same no matter in which order v and w were eliminated. If v and w are not adjacent the same sets are created when eliminating v before w and when eliminating w before v. If they are adjacent the maximum cardinality of the sets created when eliminating v and w is independent from the order of their elimination.

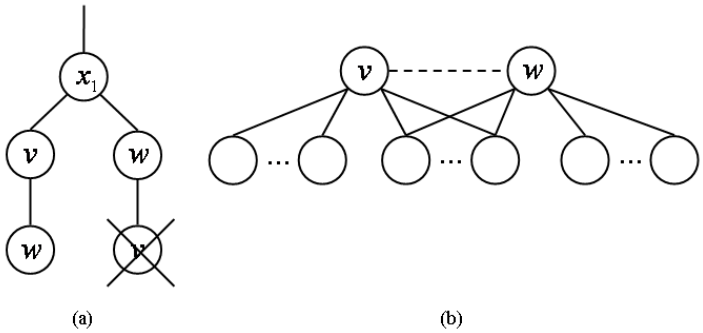

Figure 4.9: Pruning rule 2 may be applied to the vertices v and w in (b) [5]. As a consequence a subtree of the search tree may be excluded from the search (a).

4.5 A Genetic Algorithm for Triangulating the Moral Graph of Bayesian Networks

In [36] Larrañaga et al. introduced a genetic algorithm for decomposing Bayesian networks. A Bayesian network consists of vertices representing stochastic variables and directed arcs representing dependencies between the variables and may be visualized as directed acyclic graph. The variables of a Bayesian network have a finite set of states, which are comparable with the domains of a CSP, and for a variable v_i the number of possible states for that variable is denoted n_i. From the directed acyclic graph representing the Bayesian network we are able to derive a so called moral graph, which is a regular graph. The genetic algorithm presented in [36] aims at computing a "good" triangulation of that moral graph by the help of elimination orderings which is equivalent to finding a "good" tree decomposition for the moral graph. Speaking in terms of tree decompositions the genetic algorithm doesn't tend to find a tree decomposition of smallest width but it assigns a weight to a tree decomposition $TD = \langle T, \chi \rangle$ according to the following formula:

$$w(TD) = log_2 \sum_{u \in T} \prod_{v_i \in \chi(u)} n_i$$

The genetic algorithm presented in [36] uses amongst others the crossover operators described in section 4.3.2 and the mutation operators presented in section 4.3.3. Larrañaga et al. tested the genetic algorithm with two benchmark graphs trying several combinations of crossover and mutation operators in combination with different population sizes, mutation rates and selection biases. For many combinations the returned results were better than for other triangulation methods. Only the results returned by simulated annealing for the benchmarks were equivalent with the best results of the genetic algorithm.

Chapter 5

An A* Algorithm for Treewidth

Within this chapter we present an A* algorithm which is able to compute the treewidth of a regular graph. This algorithm will also compute the treewidth of a hypergraph if it is applied on the primal graph of the hypergraph.

5.1 Algorithm A*-tw

In order to compute the treewidth of a graph the A* algorithm searches the search tree representing the elimination orderings in the treewidth elimination set (Definition 25). Additionally it uses the second pruning rule (PR 2 from section 4.4.5) and the reduction rules for simplicial and strongly almost simplicial vertices for narrowing the search space (section 4.4.3). As upper bound heuristic for the treewidth we use the min-fill-heuristic and as a lower bound we take the maximum of the values returned by the minor-min-width heuristic and the minor-γ_R heuristic (section 4.4.2). The resulting algorithm is named *A*-tw* and is described in Figure 5.1.

The algorithm uses a single priority queue, denoted *queue*, for storing search states, representing the nodes of the search tree. A search state contains the variables g, h, f, where g is the width of the partial solution represented by the search state, h is the lower bound on treewidth of the graph obtained by eliminating the vertices of the partial solution and f is a lower bound on the width of all elimination orderings ending with the partial solution. Furthermore a state contains links to its *children* within the search tree. Such a link is represented by the vertex which will be eliminated next in the child state. We say that we visit a state if we remove the state from the priority queue and by evaluating a state we mean that we assign to it the values for g, h, f and its children before inserting it into the priority queue. The queue orders the states after their value f in ascending order.

First of all an upper and a lower bound on the treewidth of the graph are computed. If the upper bound on the instance equals the lower bound it is returned as the treewidth. Otherwise we evaluate the initial state representing the root of the search tree by setting $g = 0$, and assigning the value for the lower bound to h and f. If there is a simplicial or strongly almost simplicial vertex this vertex is the only child of the root state. If there is no such vertex there is a child for each vertex within the graph. Finally the initial state is inserted into the priority queue and the A* search begins.

During an iteration of the A* search the state s at top of the priority queue, having the lowest value for f, is visited. We create a graph G^s representing the graph that is obtained by eliminating the vertices of the partial solution represented by s. If $s.g \geq |G^s| - 1$ we have visited a state representing a solution and thus we return $s.g$ as the treewidth of the graph. Otherwise the children of s are evaluated and inserted into the priority queue.

For each child state t and its associated vertex v we compute its children according to the treewidth elimination set and pruning rule 2. Afterwards we determine d, the degree of vertex v in G^s, and obtain a graph G^s_v by eliminating v from G^s. The width of the partial solution represented by t is the maximum of the width of the partial solution represented by s and d, thus $t.g = max(s.g, d)$. $t.h$ is assigned a lower bound on the treewidth of graph G^s_v. Both $t.g$ as well as $t.h$ represent a lower bound on the width of all elimination orderings ending with the partial solution represented by t as well as any lower bound of a search state on the path from the root to t, thus we set $t.f = max(t.g, t.h, s.f)$. If there is a simplicial vertex or a strongly almost simplicial vertex within G^s_v this vertex is the only child of state t.

Finally we insert t into the priority queue if $t.f$ is less than the upper bound ub on the treewidth of the original graph. States with $t.f \geq ub$ won't lead to solutions which are better than the upper bound solution we have already computed therefore they are excluded from the search in order to decrease the memory needed by the A* algorithm.

If all search states with $f < ub$ have been visited but none of them represented a solution it must be that ub is the treewidth of the graph and thus ub is returned by the algorithm in that case.

Algorithm: A*-tw

Input: a graph $G = (V, E)$
Output: the treewidth of the graph

$lb = lower\_bound(G)$
$ub = upper\_bound(G)$
if $lb = ub$ **then return** ub /* *treewidth already found* */

/* *evaluate the root of the search tree* */
$r = new\ State()$
$r.h = lb, r.g = 0, r.f = lb$

if $\exists$ a simplicial vertex w in G or an almost simplicial vertex w of degree $\leq t.f$ **then**
    $r.children = \{w\}$
    $r.reduced = true$
else $r.children = V$
$queue.push(r)$

/* *A* search - visit next state in queue* */
while *queue* is not empty **do**

    $s = queue.pop()$
    create G^s

    /* *new lower bound is found* */
    if $s.f > lb$ **then** $lb = s.f$

    /* *optimal solution is found* */
    if $s.g \geq |G^s| - 1$ **then return** $s.g$

    /* *evaluate the children of current search state* */
    for each $v \in s.children$ **do**

        $t = new\ State()$
        $t.children =$ children of v according to treewidth elimination set
        if not $s.reduced$ **then** prune $t.children$ according to PR 2

        $d = degree_{G^s}(v)$
        $G^s_v = eliminate(v, G^s)$
        $t.g = max(s.g, d)$
        $t.h = lower\_bound(G^s_v)$
        $t.f = max(t.g, t.h, s.f)$

        if $\exists$ a simplicial vertex w or an almost simplicial vertex w of degree $\leq t.f$ in G^s_v **then**
            $t.children = \{w\}$
            $t.reduced = true$

        if $t.f < ub$ **then** $queue.insert(t)$

return ub

Figure 5.1: Algorithm A*-tw.

5.2 Implementation Details

5.2.1 Graph Representation

When visiting a state s in the A* algorithm we have to create a graph G^s representing the graph that is obtained by eliminating the vertices of the partial solution represented by s. In the next iteration we visit a different state t and we have to consider its corresponding graph G^t. It would be useful if we could transform G^s into G^t and this transformation process took as few steps as possible.

Within our implementation of the A* algorithm we use a single graph object which may be transformed into the graph that is needed within the current search state. This transformation is done by eliminating vertices from the graph and by restoring eliminated vertices. The graph object consists of the following data structures, requiring $O(|V|^2)$ memory:

- an integer matrix, denoted A. The i-th row of the matrix, $A[i]$ is a list for the vertex i initially containing the vertices adjacent to vertex i in G. If an edge $[i, j]$ is inserted to the graph, when eliminating a vertex, j is appended to the list $A[i]$ and i is appended to $A[j]$.
- an integer matrix, denoted E. $E[i][j]$ contains the length of the i-th adjacency list $A[i]$ after j vertices have been eliminated.
- a boolean matrix , denoted T. T is an adjacency matrix. $T[i][j] = 1$ if vertex i is adjacent to vertex j, otherwise $T[i][j] = 0$.
- a boolean array *eliminated*. *eliminated*$[i] = 1$ iff vertex i has been eliminated from the graph.

By the help of these data structures we are able to eliminate a vertex and also to restore the last eliminated vertex.

When eliminating a vertex v we are able to compute the filled in edges. For each new edge we update the entries in T and append the according vertices in A. The new lengths of the lists in A are saved within E. Finally we delete entries indicating edges containing v in T and set *eliminated*$[v] = 1$.

Assume that we restore the last eliminated vertex v and that v is the j-th vertex that has been eliminated. For each vertex i we look at the list elements that were inserted within the last step, these elements are $A[i][E[j-1]], ..., A[i][E[j]]$. In that way we are able to compute the edges which were inserted due to the elimination of v. We delete the entries corresponding to those edges in T. From the list $A[v]$ we are able

to determine the vertices to which v was adjacent before its elimination, these are the elements in $A[v][0], .., A[v][E[j-1]]$ which have not been eliminated yet. We update the corresponding entries in T and finally we set $eliminated[v] = 0$.

Example 10. Figure 5.2 shows a graph with 6 vertices (a) and the graphs obtained by eliminating vertex 6 and 2 (b) and (c). Vertices adjacent due to edges resulting from the elimination process are inserted into the lists in A (d). The length of the lists in A are stored within E (e). T (f) is the adjacency matrix of the graph in (c).

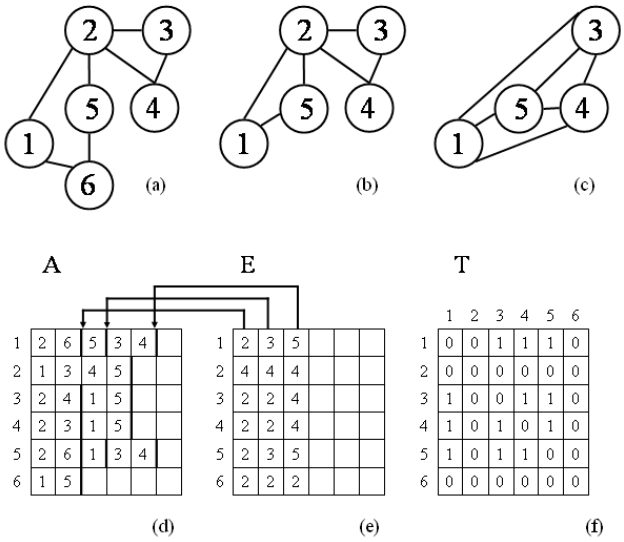

Figure 5.2: Graph sequence obtained by eliminating vertex 6 and 2 and the data structures A, E and T after the elimination of those vertices.

Now, if we want to transform our graph representing G^s into G^t we restore the vertices which have been eliminated to obtain G^s in reverse order. Afterwards we eliminate the vertices needed for obtaining G^t. If the partial elimination orderings associated with G^s into G^t have some postfix in common we need not restore and eliminate the vertices of that common postfix.

5.2.2 Partial Solutions

In order to obtain the partial solution associated with a search state s a state has two additional variables. A variable *vertex* storing the vertex which was eliminated when

that s is evaluated and a variable *predecessor* containing its predecessor state in the search tree. By the help of the predecessor variable we are able to derive the path from a state s to the root and the vertices associated with the states of this path are the vertices in the partial elimination ordering represented by search state s.

5.2.3 Memory Saving Measures

From the pseudo code notation of the A* search algorithm within section 5.1 we know that states with $f \geq ub$ will not be inserted into the priority queue because they won't lead to solutions of width smaller than the upper bound ub we have already found. Within our implementation of the algorithm we delete states with $f \geq ub$ and the memory required by those states is not allocated any more.

Note that after a state has been visited and removed from the priority queue it is not removed from the memory. The state is still needed for obtaining partial elimination orderings as described in the previous subsection. But the vertices stored within *children* associated with a visited state are not needed any more. Thus we will free the memory which was allocated for the *children* in order to reduce the memory demand of the A* search algorithm.

5.3 Computing Lower Bounds with the A* Algorithm

Recall that the f-value for a search state t is computed by $t.f = max(t.g, t.h, s.f)$ where $s.f$ is the f-value of t's predecessor in the search tree s. The value f of a state is a lower bound on the width of all elimination orderings ending with the partial solution represented by that state. Since t is a child of s in the search tree all elimination orderings ending with the partial elimination ordering represented by t are also elimination orderings ending with the partial elimination ordering represented by s, thus $s.f$ is also a lower bound on the width of all elimination orderings associated with t and therefore $s.f$ is also regarded when computing $t.f = max(t.g, t.h, s.f)$.

As a consequence the f-values along a path in the search tree are nondecreasing and also the f-values along the sequence of visited search states are nondecreasing. Thus whenever we visit a search state whose f-value is greater than all f-values considered before we obtain a new lower bound on the treewidth of the graph. If we restrict the running time of the A* algorithm by a time limit then the f-value of the last state visited before the limit was exceeded may act as a lower bound on the treewidth of a graph.

As already mentioned the states within the priority queue are ordered by their f-values. Among states with the same value for f we prioritize those which lie deeper in the search tree in the hope that we will reach a goal state earlier. Once the search

has reached the states whose f-values equal the treewidth a solution might be obtained earlier if we favor states with higher depth.

5.4 Computational Results

Within this section we present the results that our A* algorithm achieved for graphs from the Second Dimacs graph coloring challenge [18] and for some grid graphs. The A* algorithm A*-tw was implemented using C++ and STL. All experiments were run on a machine with an Intel(R) Pentium(R)-4 3.40 GHz processor having 1 GB RAM. The min-fill heuristic for getting an upper bound as well as the minor-min-width and minor-γ_R lower bound heuristics use random numbers for breaking ties. We performed ten runs for each graph instance. Each run was given a time limit of one hour and if the time limited was exceeded the algorithm returned the f-value of the last visited state, which is a lower bound on the treewidth of the instance. For each instance we report the highest value returned from the ten runs.

The tables use the following terminology. The columns *Graph*, *V* and *E* show the instance name of a graph, the number of its vertices and edges. The columns *lb* and *ub* give the lower and upper bounds on the instance which were computed at the beginning of the algorithm. The column *A*-tw* shows the value that was returned by algorithm A*-tw, bold entries indicate that the treewidth for that graph was found. The column *time* gives the time in seconds that was needed for computing the treewidth, a ”*” entry indicates that the algorithm exceeded the one hour time limit and returned only a lower bound on the graph instance.

5.4.1 Dimacs Graph Coloring Instances

The results returned by algorithm A*-tw applied on selected Dimacs graphs are shown in table 5.1. The columns *QuickBB* and *BB − tw* contain the results returned by the branch and bound algorithms QuickBB [24] and BB-tw [5] after three hours on an Intel(R) Pentium(R) 4 2.4 GHz 2GB machine and after one hour on an Intel(R) Pentium(R) 4 2.8 GHz respectively. ”*” entries indicate that the algorithm did not return the treewidth of the graph, ”-” entries indicate that an algorithm was not applied to an instance. Only for the instances myciel5 and queen7_7 A*-tw did not return the treewidth whereas QuickBB and BB-tw could. For the instance miles1000 A*-tw was able to compute the treewidth which hasn’t been fixed before. For the instance DSJC125.5 A*-tw returned a significantly improved lower bound of 82.

Graph	*V*	*E*	*lb*	*ub*	*A*-tw*	*time*	*QuickBB*	*BB – tw*
anna	**138**	**986**	**11**	**12**	**12**	**0.02**	**12**	**12**
david	**87**	**12**	**12**	**13**	**13**	**1.31**	**13**	**13**
huck	**74**	**602**	**10**	**10**	**10**	**0**	**10**	**-**
jean	**80**	**508**	**9**	**9**	**9**	**0**	**9**	**-**
queen5_5	**25**	**320**	**12**	**18**	**18**	**1.35**	**18**	**18**
queen6_6	**36**	**580**	**16**	**26**	**25**	**115.21**	**25**	**25**
queen7_7	49	952	20	37	31	*	35	-
fpsol2.i.1	**496**	**11654**	**66**	**66**	**66**	**3.78**	**66**	**-**
fpsol2.i.2	**451**	**8691**	**31**	**31**	**31**	**2.32**	**31**	**-**
fpsol2.i.3	**425**	**8688**	**31**	**31**	**31**	**2.1**	**31**	**-**
inithx.i.1	**864**	**18707**	**56**	**56**	**56**	**12.97**	**56**	**-**
inithx.i.2	**645**	**13979**	**31**	**31**	**31**	**5.89**	**31**	**31**
inithx.i.3	**621**	**13969**	**31**	**31**	**31**	**5.5**	**31**	**31**
mulsol.i.1	**197**	**3925**	**50**	**50**	**50**	**0.21**	**50**	**-**
mulsol.i.2	**188**	**3885**	**32**	**32**	**32**	**0.17**	**32**	**-**
mulsol.i.3	**184**	**3916**	**32**	**32**	**32**	**0.16**	**32**	**-**
mulsol.i.4	**185**	**3946**	**32**	**32**	**32**	**0.17**	**32**	**-**
mulsol.i.5	**186**	**3973**	**31**	**32**	**31**	**1.18**	**31**	**-**
miles1000	**128**	**6432**	**48**	**50**	**49**	**4.02**	*****	**-**
miles1500	**128**	**10396**	**77**	**77**	**77**	**0.14**	**77**	**-**
miles250	**128**	**774**	**9**	**9**	**9**	**0**	**9**	**-**
miles500	**128**	**2340**	**22**	**23**	**22**	**0.61**	**22**	**-**
miles750	128	4226	34	40	34	*	*	-
myciel3	**11**	**20**	**4**	**5**	**5**	**0**	**5**	**-**
myciel4	**23**	**71**	**8**	**11**	**10**	**0.7**	**10**	**10**
myciel5	47	236	14	21	16	*	19	19
DSJC125.1	125	736	23	66	24	*	*	*
DSJC125.5	125	3891	58	111	82	*	*	*
DSJC125.9	**125**	**6961**	**105**	**119**	**119**	**38.39**	**119**	**-**
DSJR500.1c	**500**	**121275**	**475**	**485**	**485**	**547.98**	**485**	**-**
le450_5a	450	5714	62	315	63	*	*	*
le450_15a	450	8168	75	290	75	*	*	-
le450_25a	450	8260	75	258	77	*	*	-
zeroin.i.1	**211**	**4100**	**50**	**50**	**50**	**0.530**	**-**	**-**
zeroin.i.2	**211**	**3541**	**32**	**33**	**32**	**0.410**	**-**	**-**
zeroin.i.3	**206**	**3540**	**32**	**33**	**32**	**0.390**	**-**	**-**

Table 5.1: Dimacs graph coloring benchmarks.

5.4.2 Grid Graphs

Table 5.2 presents the results returned by algorithm A*-tw applied on several grid graphs. It is folklore that the treewidth of a $n \times n$-grid is n. A*-tw was able to compute the treewidth up to the 6×6-grid if it was given a one hour time limit.

Graph	*V*	*E*	*lb*	*ub*	$A^* - tw$	*time*
grid2	**4**	**4**	**2**	**2**	**2**	**0**
grid3	**9**	**12**	**3**	**3**	**3**	**0**
grid4	**16**	**24**	**4**	**4**	**4**	**0**
grid5	**25**	**40**	**4**	**5**	**5**	**0**
grid6	**36**	**60**	**4**	**6**	**6**	**150.46**
grid7	49	84	4	8	5	*
grid8	64	112	4	10	5	*

Table 5.2: Grid graphs.

Chapter 6

A Genetic Algorithm for Treewidth Upper Bounds

In this chapter we present a genetic algorithm which computes upper bounds on the treewidth of graphs. This algorithm will also compute an upper bound on the treewidth of a hypergraph if it is applied on the primal graph of the hypergraph.

6.1 Algorithm GA-tw

The genetic algorithm for computing upper bounds on the treewidth of graphs is named *GA-tw*. Figure 6.1 presents algorithm GA-tw in pseudo code notation.

The algorithm takes as input a regular graph for which an upper bound on the treewidth should be computed and the control parameters n, p_m, p_c, s and *max_iterations*. The population size n specifies the number of individuals within the population of the genetic algorithm, p_m specifies the mutation rate, p_c the crossover rate As selection technique we use *tournament selection* which requires a parameter s, the group size. *max_iterations* gives the number of generations over which the population is evolved.

An individual solution is an elimination ordering and is represented as permutation of the vertices of the graph. The initial population consists of n randomly created individuals. Each individual is evaluated and is assigned a fitness value. The fitness is the width of the tree decomposition which may be created via bucket or vertex elimination from the elimination ordering. The best fitness of an individual of the initial population is recorded by the genetic algorithm. The population is evolved over *max_iterations* iterations.

Whithin each iteration we select the individuals which will enter the next population via tournament selection. Tournament selection selects an individual by choosing randomly a group of s individuals from the former population and the individual of highest fitness (smallest width) within this group is selected to join the next population. This process is applied until n individuals have been selected.

For recombining the individuals of the solution we apply one of the crossover operators presented in section 4.3.2. The crossover rate p_c determines the number of individuals which undergo recombination, e.g. if $p_c = 0.8$ then 80% of the individuals within the population are recombined with each other whereas 20% remain unchanged.

As mutation operator we apply one of the mutation operators introduced in section 4.3.3. The mutation rate p_m determines the probability that an individual is mutated. For each individual of the population we compute a uniformly distributed random number $x \in [0, 1]$ and if $x < p_m$ we mutate the corresponding individual.

At the end of each iteration the individuals are evaluated again. Whenever an individual's fitness is better than the best fitness found so far its fitness value is recorded as the best fitness (smallest width). Finally the best fitness (smallest width) found by the genetic algorithm is returned as an upper bound on the treewidth of the graph.

Note that since every tree decomposition of a hypergraph is also a tree decomposition of the hypergraph's primal graph and vice versa (Lemma 1 [33]) algorithm GA-tw may also be used in order to compute upper bounds on the treewidth of a hypergraph if it is applied to the primal graph of the hypergraph.

Algorithm: GA-tw

Input: a graph $G = (V, E)$
control paramters for the GA n, p_m, p_c, s and $max\_iterations$
Output: an upper bound on the treewidth of the graph

$t = 0$
initialize ($population(t), n$)
evaluate $population(t)$

while $t < max\_iterations$ **do**
  $t = t + 1$
  $population(t)$ = tournament_selection($population(t-1), s$)
  recombine ($population(t), p_c$)
  mutate ($population(t), p_m$)
  evaluate $population(t)$

return the smallest width found during the search

Figure 6.1: Algorithm GA-tw.

6.2 Implementation Details

We implemented algorithm GA-tw using C++ and STL.

6.2.1 Graph Representation

For the representation of the input graph of algorithm GA-tw we chose the same representation as in section 5.2.1 for the A* algorithm A*-tw.

6.2.2 Evaluating Individuals

In order to evaluate individual solutions, which are in fact elimination orderings, we modified the algorithm for deciding if an elimination ordering is a perfect elimination ordering presented in [25]. Algorithm *Evaluate Individual* in Figure 6.2 produces the same sets of vertices as bucket or vertex elimination. With our graph representation the modified algorithm in Figure 6.2 has running time $O(|V| + |E'|)$ [25], where $|E'|$ is the set containing the original edges of the graph and the edges which are filled in when eliminating the vertices of the graph.

Algorithm: Evaluate Individual
Input: a list of adjacency lists, denoted A, representing graph $G = (V, E)$
an elimination ordering $\sigma = (v_1, ..., v_n)$
Output: the width of the tree decomposition according to σ

$width = 0$, $i = n$
while $width < i$ **do**
$X = \{x \in A[v_i] \mid x <_\sigma v_i\}$
$width = max(|X|, width)$
Let u be the vertex in X which is eliminated next in σ
$A[u] = A[u] \cup (X - \{u\})$
$i = i - 1$

return *width*

Figure 6.2: Evaluation function used in GA-tw.

6.3 Computational Results

First of all we tried to find good values for the control parameters of algorithm GA-tw in order to obtain small upper bounds on the treewidth of graphs in our computational experiments. Afterwards we applied algorithm GA-tw using the obtained parameter values on many graphs of the Second Dimacs graph coloring challenge [18]. For many instances our genetic algorithm was able to return upper bounds on the treewidth of the graph which are better than the upper bounds obtained in [4], [5], [13] and [24].

6.3.1 Comparison of Crossover Operators

We compared the crossover operators of section 4.3.2 with each other by applying them to selected graphs of the Second Dimacs graph coloring challenge [18]. For each crossover operator and each graph we ran our algorithm GA-tw five times with population size $n = 50$ and group size $s = 2$ for tournament selection. Each single run lasted exactly 1000 iterations. Table 6.1 shows the average, the minimum and the maximum width achieved by the crossover operators during the five runs with 100% crossover rate and with 0% mutation rate. Since position-based crossover (POS) achieved the best average width for all instances we chose it as the crossover operator for our further tests.

6.3.2 Comparison of Mutation Operators

In order to compare the mutation operators of section 4.3.3 we applied algorithm GA-tw to several graphs of the Second Dimacs graph coloring challenge [18]. For each mutation operator and each graph we ran our algorithm GA-tw five times with population size $n = 50$ and group size $s = 2$ for tournament selection. Each single run lasted exactly 1000 iterations. Table 6.2 shows the average, the minimum and the maximum width achieved during the five runs with 0% crossover rate and with 100% mutation rate. Since the insertion mutation operator (ISM) achieved the best average width in most cases we chose it as mutation operator for our further experiments.

6.3.3 Determining Suitable Mutation and Crossover Rates

In order to obtain good values for the mutation and crossover rate we considered different combinations of mutations rates, $p_m = 1\%, 10\%, 30\%$, and recombination rates, $p_c = 80\%, 90\%, 100\%$, and applied algorithm GA-tw using those combinations to selected instances of the Second Dimacs graph coloring challenge [18]. For each combination and each graph we ran our algorithm GA-tw five times with population size $n = 200$ and group size $s = 2$ for tournament selection. As crossover operator we

used position-based crossover (POS), as mutation operator the insertion mutation operator (ISM). Each single run lasted exactly 1000 iterations. The average, the minimum and maximum width achieved during the five runs are shown in Table 6.3. The combination of a recombination rate of 100% and a mutation rate of 30% achieved good average results with all instances and performed best with the large instances le450_25d and queen16_16, thus we chose this combination for our further experiments.

6.3.4 Population Size and Tournament Selection Group Size

We considered populations of 100, 200, 1000, and 2000 individuals. Table 6.4 shows the average, minimum and maximum width for Dimacs graphs [18] returned by algorithm GA-tw after five runs of 1000 iterations. A population of size $n = 2000$ achieves the best results in three out of four instances. For such populations a tournament selection group size of $s = 3$ or $s = 4$ seems to be the best choice as it can be seen in Table 6.5.

6.3.5 Final Results for Dimacs Benchmarks Graphs

Finally we applied algorithm GA-tw on 62 graphs of the Second Dimacs graph coloring challenge [18]. We ran GA-tw with the control parameters obtained in the previous subsections, which are a population size of $n = 2000$, a crossover rate of $p_c = 1.0$ or 100%, a mutation rate $p_m = 0.3$ or 30%, and a tournament selection group size of $s = 3$. We performed ten runs for each graph instance. A single run of GA-tw lasted 2000 iterations, thus each run of algorithm GA-tw carried out four million evaluations of individual solutions. As crossover and mutation operators we used position-based crossover (POS) and the insertion mutation opertor (ISM). For each graph we performed ten runs on machine with an Intel(R) Pentium(R)- 4 3.40GHz processor having 1GB RAM.

Table 6.6 shows the results for the considered graphs. The columns *Graph*, *V* and *E* present the graph name and the number of vertices and edges of that graph. *ub* contains the value of the smallest upper bound for a graph reported in [4], [5], [13] and [24]. *min*, *max* and *avg* present the best, worst and average width returned by algorithm GA-tw for an instance whereas *std*. *dev*. contains the standard deviation of the ten results returned by algorithm GA-tw. Column *min-time* presents the time which was needed by algorithm GA-tw for the run which returned the width in column *min*, *avg-time* the average running time of the ten runs.

Compared with the best upper bounds for the considered instances in [4], [5], [13] and [24], algorithm GA-tw found an improved upper bound on the treewidth for 22 graphs, GA-tw was able to return the same upper bound for 31 graphs, and for only 9 graphs the results delivered by GA-tw were worse.

Instance	*Crossover Operator*	*avg*	*min*	*max*
games120	POS	37	35	40
games120	OX2	46.6	40	49
games120	PMX	50.2	45	53
games120	OX1	56.2	56	57
games120	CX	59.2	56	62
games120	AP	60.8	59	62
homer	POS	42.2	37	50
homer	OX2	53.8	45	60
homer	PMX	72.8	65	85
homer	CX	98	91	105
homer	OX1	118.4	114	121
homer	AP	143.8	135	151
inithx.i.3	POS	129.8	50	184
inithx.i.3	OX2	204.4	190	220
inithx.i.3	OX1	321.6	278	338
inithx.i.3	PMX	331.8	283	387
inithx.i.3	CX	368	351	394
inithx.i.3	AP	370.2	322	384
le450_25d	POS	370	364	376
le450_25d	OX2	375.8	370	379
le450_25d	PMX	391.8	388	399
le450_25d	CX	394.2	392	396
le450_25d	OX1	396.2	394	398
le450_25d	AP	401.6	399	403
myciel7	POS	75	70	83
myciel7	OX2	86.8	80	97
myciel7	PMX	108.2	101	115
myciel7	CX	113.4	109	116
myciel7	OX1	119	116	121
myciel7	AP	128.8	124	133
queen16_16	POS	207	202	211
queen16_16	OX2	213	209	219
queen16_16	PMX	217.6	214	221
queen16_16	OX1	224.2	222	225
queen16_16	CX	224.6	223	227
queen16_16	AP	227.4	225	229
zeroin.i.3	POS	40.2	33	45
zeroin.i.3	OX2	51.4	45	60
zeroin.i.3	OX1	93	85	99
zeroin.i.3	PMX	98	94	106
zeroin.i.3	CX	99.4	86	112
zeroin.i.3	AP	101.4	74	123

Table 6.1: Comparison of crossover operators.

Instance	*Mutation Operator*	*avg*	*min*	*max*
games120	ISM	37.4	35	43
games120	EM	38.2	38	39
games120	SM	48.8	47	50
games120	SIM	49.8	48	52
games120	DM	54	52	56
games120	IVM	56.4	55	58
homer	EM	42.8	39	47
homer	ISM	43.6	41	49
homer	SM	81.4	78	85
homer	SIM	91.6	79	100
homer	DM	101.2	94	105
homer	IVM	102.4	96	107
inithx.i.3	ISM	65.8	56	74
inithx.i.3	EM	121.2	93	156
inithx.i.3	SM	208.2	184	265
inithx.i.3	SIM	230.6	205	271
inithx.i.3	DM	243.8	228	275
inithx.i.3	IVM	274.8	264	289
le450_25d	ISM	359.2	349	364
le450_25d	EM	367.2	361	372
le450_25d	DM	384.2	381	388
le450_25d	SM	388.8	385	395
le450_25d	SIM	390.4	388	392
le450_25d	IVM	393.2	391	397
myciel7	ISM	70.4	68	77
myciel7	EM	78.4	71	87
myciel7	SM	99.6	98	101
myciel7	SIM	106.2	106	107
myciel7	DM	110.8	110	112
myciel7	IVM	113.4	112	116
queen16_16	ISM	202.4	197	209
queen16_16	EM	209	204	215
queen16_16	DM	217.6	214	220
queen16_16	IVM	220.2	217	223
queen16_16	SM	220.2	217	224
queen16_16	SIM	222.6	222	224
zeroin.i.3	ISM	34.8	33	37
zeroin.i.3	EM	41.2	40	43
zeroin.i.3	SM	63.6	56	70
zeroin.i.3	SIM	64.8	61	69
zeroin.i.3	DM	81.2	80	84
zeroin.i.3	IVM	85	80	91

Table 6.2: Comparison of mutation operators.

Instace	p_c	p_m	*avg*	*min*	*max*
games120	0.8	0.1	33	32	34
games120	0.9	0.01	33	32	34
games120	1	0.1	33	32	34
games120	0.9	0.3	33.2	33	34
games120	1	0.01	33.4	33	34
games120	0.9	0.1	33.6	32	36
games120	0.8	0.3	33.8	32	38
games120	0.8	0.01	34	34	34
games120	1	0.3	34.4	33	39
homer	1	0.01	31.2	31	32
homer	1	0.3	31.2	31	32
homer	0.9	0.3	31.4	31	32
homer	0.8	0.3	31.6	31	32
homer	0.9	0.01	31.6	31	32
homer	0.9	0.1	31.6	30	34
homer	1	0.1	31.6	31	32
homer	0.8	0.01	32	31	33
homer	0.8	0.1	32	31	34
inithx.i.3	0.8	0.1	35	35	35
inithx.i.3	0.8	0.3	35	35	35
inithx.i.3	0.9	0.01	35	35	35
inithx.i.3	0.9	0.1	35	35	35
inithx.i.3	0.9	0.3	35	35	35
inithx.i.3	1	0.01	35	35	35
inithx.i.3	1	0.1	35	35	35
inithx.i.3	1	0.3	35	35	35
inithx.i.3	0.8	0.01	35.4	35	36
le450_25d	1	0.3	335.6	333	338
le450_25d	0.9	0.3	339.2	334	345
le450_25d	1	0.1	339.8	336	344
le450_25d	0.8	0.3	340.8	336	346
le450_25d	0.9	0.01	341.4	335	349
le450_25d	0.8	0.1	341.8	337	345
le450_25d	1	0.01	342.2	339	344
le450_25d	0.8	0.01	344.2	341	346
le450_25d	0.9	0.1	344.4	336	351
myciel7	0.8	0.01	66	66	66
myciel7	0.8	0.1	66	66	66
myciel7	0.8	0.3	66	66	66
myciel7	0.9	0.01	66	66	66
myciel7	0.9	0.1	66	66	66
myciel7	0.9	0.3	66	66	66
myciel7	1	0.01	66	66	66
myciel7	1	0.1	66	66	66
myciel7	1	0.3	66	66	66
queen16_16	1	0.3	190.6	187	193
queen16_16	0.9	0.01	191.4	189	194
queen16_16	1	0.01	191.6	189	195
queen16_16	0.9	0.3	191.8	190	193
queen16_16	1	0.1	191.8	190	195
queen16_16	0.9	0.1	192	190	195
queen16_16	0.8	0.1	192.8	189	198
queen16_16	0.8	0.3	193.2	190	195
queen16_16	0.8	0.01	194.2	187	197
zeroin.i.3	0.9	0.01	32.4	32	33
zeroin.i.3	0.8	0.1	32.8	32	33
zeroin.i.3	0.9	0.3	32.8	32	33
zeroin.i.3	1	0.1	32.8	32	33
zeroin.i.3	0.8	0.01	33	33	33
zeroin.i.3	0.8	0.3	33	33	33
zeroin.i.3	0.9	0.1	33	33	33
zeroin.i.3	1	0.01	33	33	33
zeroin.i.3	1	0.3	33	33	33

Table 6.3: Comparison of different combinations of mutation rate and crossover rate.

Instance	*n*	*Average*	*min*	*max*
le450_25d	2000	334.8	334	337
le450_25d	1000	335	332	338
le450_25d	200	339.66	336	342
le450_25d	100	342.4	335	349
LE450_5B.col	2000	264.6	258	282
LE450_5B.col	1000	266.2	256	280
LE450_5B.col	200	266.33	254	281
LE450_5B.col	100	273.6	264	293
queen16_16	2000	189.2	187	191
queen16_16	1000	190.8	188	193
queen16_16	200	191	188	194
queen16_16	100	194.4	194	195
zeroin.i.3	200	32.66	32	33
zeroin.i.3	100	33	33	33
zeroin.i.3	1000	33	33	33
zeroin.i.3	2000	33	33	33

Table 6.4: Comparison of different population sizes.

Instance	*s*	*avg*	*min*	*max*
le450_25d	4	331.8	328	336
le450_25d	3	332.2	329	335
le450_25d	2	334.8	334	337
LE450_5B.col	3	257.4	250	267
LE450_5B.col	4	264.4	251	283
LE450_5B.col	2	264.6	258	282
queen16_16	4	187.6	184	194
queen16_16	3	188.2	185	191
queen16_16	2	189.2	187	191
zeroin.i.3	2	33	33	33
zeroin.i.3	3	33	33	33
zeroin.i.3	4	33	33	33

Table 6.5: Comparison of different group sizes for tournament selection.

Graph	*V*	*E*	*ub*	*min*	*max*	*avg*	*std.dev.*	*min-time*	*avg-time*
anna	138	986	12	12	12	12	0.00	00:03:33	00:03:32
david	87	812	13	13	13	13	0.00	00:02:34	00:02:32
huck	74	602	10	10	10	10	0.00	00:02:00	00:01:59
homer	561	3258	31	31	31	31	0.00	00:18:38	00:18:36
jean	80	508	9	9	9	9	0.00	00:02:00	00:01:59
games120	120	1276	33	32	32	32	0.00	00:07:42	00:07:32
queen5_5	25	320	18	18	18	18	0.00	00:00:33	00:00:33
queen6_6	36	580	25	26	26	26	0.00	00:00:51	00:00:51
queen7_7	49	952	35	35	36	35.2	0.42	00:01:32	00:01:34
queen8_8	64	1456	46	45	47	46	0.47	00:02:47	00:02:30
queen9_9	81	2112	58	58	60	58.5	0.71	00:03:50	00:03:50
queen10_10	100	2940	72	72	73	72.4	0.52	00:05:39	00:05:35
queen11_11	121	3960	88	87	90	88.2	1.14	00:08:17	00:07:55
queen12_12	144	5192	104	104	108	105.7	1.34	00:10:33	00:10:52
queen13_13	169	6656	122	121	125	123.1	1.29	00:15:06	00:14:50
queen14_14	196	8372	141	141	148	144	2.16	00:19:41	00:19:24
queen15_15	225	10360	163	162	168	164.8	1.87	00:25:44	00:25:17
queen16_16	256	12640	186	186	191	188.5	1.90	00:34:53	00:31:41
fpsol2.i.1	496	11654	66	66	66	66	0.00	00:33:02	00:32:29
fpsol2.i.2	451	8691	31	32	33	32.6	0.52	00:24:05	00:23:45
fpsol2.i.3	425	8688	31	31	33	32.3	0.67	00:24:22	00:22:49
inithx.i.1	864	18707	56	56	56	56	0.00	00:56:18	00:55:42
inithx.i.2	645	13979	31	35	35	35	0.00	00:38:37	00:38:24
inithx.i.3	621	13969	31	35	35	35	0.00	00:37:41	00:37:17
miles1000	128	6432	49	50	50	50	0.00	00:09:19	00:09:24
miles1500	128	10396	77	77	77	77	0.00	00:07:37	00:07:33
miles250	128	774	9	10	10	10	0.00	00:04:02	00:04:01
miles500	128	2340	22	24	25	24.1	0.32	00:07:22	00:07:16
miles750	128	4226	36	37	37	37	0.00	00:08:56	00:08:50
mulsol.i.1	197	3925	50	50	50	50	0.00	00:11:11	00:11:05
mulsol.i.2	188	3885	32	32	32	32	0.00	00:09:44	00:09:48
mulsol.i.3	184	3916	32	32	32	32	0.00	00:09:39	00:09:32
mulsol.i.4	185	3946	32	32	32	32	0.00	00:09:38	00:09:33
mulsol.i.5	186	3973	31	31	31	31	0.00	00:09:44	00:09:31
myciel3	11	20	5	5	5	5	0.00	00:00:14	00:00:14
myciel4	23	71	10	10	10	10	0.00	00:00:34	00:00:34
myciel5	47	236	19	19	19	19	0.00	00:01:20	00:01:18
myciel6	95	755	35	35	35	35	0.00	00:03:52	00:03:48
myciel7	191	2360	54	66	66	66	0.00	00:12:37	00:12:24
school1	385	19095	188	185	199	192.5	5.66	01:18:04	01:21:35
school1_nsh	352	14612	162	157	170	163.1	5.40	01:10:39	01:09:05
zeroin.i.1	211	4100	50	50	50	50	0.00	00:10:41	00:10:30
zeroin.i.2	211	3541	32	32	33	32.7	0.48	00:09:54	00:09:46
zeroin.i.3	206	3540	32	32	33	32.9	0.32	00:09:45	00:09:38
le450_5a	450	5714	256	243	263	248.3	7.12	01:47:13	01:51:24
le450_5b	450	5734	254	248	253	249.9	1.60	01:52:12	01:49:50
le450_5c	450	9803	272	265	272	267.1	2.28	01:38:37	01:35:37
le450_5d	450	9757	278	265	268	265.6	1.07	01:30:02	01:25:08
le450_15a	450	8168	272	265	275	268.7	3.71	01:54:36	01:42:21
le450_15b	450	8169	270	265	271	269	1.63	01:47:03	01:39:08
le450_15c	450	16680	359	351	359	352.8	2.44	01:23:17	01:22:05
le450_15d	450	16750	360	353	361	356.9	2.56	01:21:04	01:17:57
le450_25a	450	8260	234	225	232	228.2	2.10	01:40:25	01:41:05
le450_25b	450	8263	233	227	239	234.5	3.47	01:40:45	01:46:06
le450_25c	450	17343	327	320	331	327.1	3.78	01:43:09	01:34:08
le450_25d	450	17425	336	327	335	330.1	2.33	01:51:52	01:35:06
DSJC125.1	125	736	64	61	63	61.9	0.74	00:08:21	00:07:47
DSJC125.5	125	3891	109	109	110	109.2	0.42	00:04:21	00:04:19
DSJC125.9	125	6961	119	119	119	119	0.00	00:01:50	00:01:54
DSJC250.1	250	3218	173	169	171	169.7	0.82	00:31:18	00:27:02
DSJC250.5	250	15668	232	230	233	231.4	0.84	00:10:48	00:09:57
DSJC250.9	250	27897	243	243	244	243.1	0.32	00:03:58	00:04:01

Table 6.6: Final results for Dimacs graphs.

Chapter 7

Genetic Algorithms for Generalized Hypertree Width Upper Bounds

In this chapter we present a genetic algorithm for computing upper bounds on the generalized hypertree width of hypergraphs. Furthermore we will propose an extension of that algorithm which will be able to adapt its control parameters itself without external specification.

7.1 The Genetic Algorithm GA-ghw

7.1.1 Algorithm GA-ghw

The genetic algorithm for computing upper bounds on the generalized hypertree width of hypergraphs is named *GA-ghw*. GA-ghw is basically the same algorithm as algorithm GA-tw (Figure 6.1) presented in the previous chapter. The only differences between GA-tw and GA-ghw are that

1. GA-ghw takes a hypergraph as input whereas GA-tw expects a regular graph.
2. GA-ghw evaluates individual solutions in a different way. The fitness of an individual is its width in terms of generalized hypertree decompositions.

7.1.2 Implementation Details

We implemented algorithm GA-ghw using C++ and STL.

Graph Representation

Algorithm GA-ghw takes a hypergraph $\mathcal{H} = (V, H)$ as input and computes its primal graph $G^*(\mathcal{H})$. The primal graph, which is a regular graph, is represented as described in section 5.2.1. Furthermore GA-ghw stores the hyperedges H of the hypergraph and the information which hyperedges contain which vertices.

Evaluating Individuals

When evaluating an individual solution, which is an elimination ordering, algorithm GA-ghw computes the sets of vertices produced by bucket or vertex elimination for the primal graph. The width of a generalized hypertree decomposition is the maximum number of hyperedges associated to a decomposition vertex. Thus, for each vertex set the evaluation function of algorithm GA-ghw (Figure 7.1) computes an upper bound on the minimum number of hyperedges needed for covering the vertex set. For that purpose a heuristic named *Greedy Set Cover* [11] (Figure 7.2) is used. Greedy Set Cover successively takes the hyperedge containing most uncovered vertices until all vertices are covered. Ties are broken at random. The maximum number of hyperedges that was needed in order to cover a vertex set is returned as the fitness value of the evaluated elimination ordering.

Algorithm: Evaluate Individual
Input: a list of adjacency lists, denoted A, representing the primal graph $G^*(\mathcal{H}) = (V, E)$
the set of hyperedges H from the original hypergraph $\mathcal{H} = (V, H)$
an elimination ordering $\sigma = (v_1, ..., v_n)$
Output: the width of the generalized hypertree decomposition according to σ

$width = 0$, $i = n$
while $width < i$ **do**
  $X = \{x \in A[v_i] \mid x <_\sigma v_i\}$
  $\chi(v_i) = \{v_i\} \cup X$
  $k = Greedy\ Set\ Cover(\chi(v_i), H)$
  $width = max(k, width)$
  Let u be the vertex in X which is eliminated next in σ
  $A[u] = A[u] \cup (X - \{u\})$
  $i = i - 1$

return *width*

Figure 7.1: Evaluation function used in GA-ghw.

Algorithm: Greedy Set Cover
Input: a set of vertices to cover $\chi(v_i)$
a set of hyperedges H
Output: an upper bound on the minimum number of hyperedges
needed to cover the vertices in $\chi(v_i)$

$C = \emptyset$
while $\chi(v_i) \not\subseteq C$ **do**
Select an hyperedge $h \in H$ containing the maximum number of uncovered vertices.
Ties are broken randomly.
$C = C \cup h$

return $|C|$

Figure 7.2: Greedy set cover algorithm [11] .

7.1.3 Computational Results

Due to time restrictions, we tested algorithm GA-ghw only on 19 hypergraphs of the CSP hypergraph library from [22]. GA-ghw was executed with the control parameters which were obtained for algorithm GA-tw within the previous chapter, a population size of $n = 2000$, a crossover rate of $p_c = 1.0$ or 100%, a mutation rate $p_m = 0.3$ or 30%, and a tournament selection group size of $s = 3$. For each hypergraph we performed ten runs of GA-ghw. A single run of GA-gwh lasted $2,000$ iterations thus each run of algorithm GA-ghw carried out four million evaluations of individual solutions. As crossover and mutation operators we used position-based crossover (POS) and the insertion mutation operator (ISM).

We tested the ten runs for each hypergraph either on a machine (1) with an Intel(R) Pentium(R)-4 3.40GHz processor having 1GB RAM or on a machine (2) with an Intel(R) Xeon(TM) 3.20GHz processor having 4GB RAM. We converted the times of machine (1) into the times machine of machine (2). On both machines we applied GA-ghw to four different instances using the same random seed. For each instance the run at machine (2) took 79% of the computation time of the run at machine (1).

Table 7.1 enlists the results of GA-ghw for the considered hypergraphs. The columns *Graph*, *V* and *H* present the graph name and the number of vertices and hyperedges of that graph. *ub* contains the value of the smallest upper bound on the generalized hypertree width for a hypergraph reported in [17]. *min*, *max* and *avg* present the best, worst and average width returned by algorithm GA-ghw for an instance whereas *std.dev.* contains the standard deviation of the ten results returned by algorithm GA-ghw. Within the column *min-time* we present the time which was needed by algorithm GA-ghw for

Hypergraph	*V*	*H*	*ub*	*min*	*max*	*avg*	*std.dev.*	*min-time*	*avg-time*
adder_75	526	376	2	3	3	3	0.00	05:02:54	05:02:51
adder_99	694	494	2	3	3	3	0.00	08:33:10	08:33:53
b06	50	48	5	4	4	4	0.00	00:09:39	00:09:44
b08	179	170	10	9	9	9	0.00	01:04:39	01:04:44
b09	169	168	10	7	7	7	0.00	01:14:43	01:14:54
b10	200	189	14	11	12	11.8	0.42	01:51:47	01:51:39
bridge_50	452	452	2	6	6	6	0.00	06:33:56	06:33:25
c499	243	202	13	11	12	11.7	0.48	02:13:10	02:13:13
c880	443	383	19	17	18	17.2	0.42	06:54:25	06:55:26
clique_20	190	20	10	11	12	11.2	0.42	01:30:19	01:30:43
grid2d_20	200	200	11	10	10	10	0.00	01:36:00	01:35:32
grid3d_8	256	256	20	21	22	21.3	0.48	04:53:40	04:49:49
grid4d_4	128	128	17	15	16	15.3	0.48	01:24:17	01:24:42
grid5d_3	122	121	18	16	18	16.7	0.82	01:25:32	01:24:51
nasa	579	680	21	19	22	19.9	0.74	17:13:13	17:19:44
NewSystem1	142	84	3	3	4	3.1	0.32	00:36:45	00:36:59
NewSystem2	345	200	4	4	4	4	0.00	03:01:16	03:01:36
s444	205	202	6	5	5	5	0.00	01:46:54	01:47:07
s510	236	217	23	17	17	17	0.00	02:40:09	02:41:52

Table 7.1: GA-ghw results for selected benchmark hypergraphs.

the run which returned the width in column *min*, column *avg-time* presents the average time of the ten runs.

Compared with the best upper bounds known for the considered instances algorithm GA-ghw found an improved upper bound on the generalized hypertree width for 12 graphs , GA-ghw was able to return the same upper bound for 2 graphs, and for 5 graphs the width returned by GA-ghw was worse than the best upper bound known so far.

7.2 Extending GA-ghw to a Self-Adaptive Island GA

As already mentioned in section 4.3 the behavior of a genetic algorithm depends on the values chosen for its control parameters. Adjusting the parameters of a genetic algorithm takes a lot of time and the optimal parameter values of a genetic algorithm may differ for several problem instances or genetic operators. It might even be that the optimal parameter values vary at different stages of the search performed by a genetic algorithm. In order to overcome these problems several genetic algorithms have been introduced, which can automatically adjust their parameter values. A classification and an overview of such algorithms is given by Eiben et al. in [20]. In [19] Takashima et al. propose a genetic algorithm, named SAIGA (self-adaptive island genetic algorithm), which adapts the population size, the crossover rate, the mutation rate and the group size for tournament selection. Since these are also the control parameters of algorithm GA-ghw we will extend algorithm GA-ghw by the help of the ideas proposed in [19].

7.2.1 Algorithm SAIGA-ghw

Algorithm: SAIGA-ghw

Input: a hypergraph $\mathcal{H} = (V, H)$
Output: an upper bound on the treewidth of the graph

for $i = 1$ to I **do** /* I ... *the number of islands* */
  initialize $param_i$
  initialize $island_i$ with $param_i$

while *evaluations* < *max_evaluations* **do**
  /* *evolution* */
  for $i = 1$ to I **do**
    evolve $island_i$ with $param_i$
  /* *migration* */
  for $i = 1$ to I **do**
    migrate $island_i$
  /* *neighbor orientation* */
  for $i = 1$ to I **do**
    orientate $param_i$ at its neighbors
  /* *mutation* */
  for $i = 1$ to I **do**
    mutate $param_i$

return the smallest width found during the search

Figure 7.3: Algorithm SAIGA-ghw.

The genetic algorithm for computing upper bounds on the generalized hypertree width of hypergraphs which adapts control parameters is named *SAIGA-ghw*. Figure 7.3 shows algorithm SAIGA-ghw in pseudo code notation.

SAIGA-ghw is an island GA [48], it evolves several genetic algorithms in parallel and each single genetic algorithm is regarded as an island. From time to time some solutions migrate between the islands and ensure that islands share some information among each other and thus perform some kind of cooperative search. Within SAIGA-ghw we place the islands in a ring topology.

A single island $island_i$ is controlled by the parameter vector $param_i = (n, p_c, p_m, s)$ which contains values for the control parameters n, the population size, p_c, the recombination rate, p_m, the mutation rate, and s, the group size for tournament selection. Initially the parameter vector is created randomly for each island.

During an iteration of SAIGA-ghw each island $island_i$ is evolved like algorithm GA-ghw (selection, recombination, mutation and evaluation) using the parameters spec-

ified in $param_i$. If a certain number of evaluations occurred within an island the evolution process is halted. If the population size of an island must increase due to an altered parameter vector new randomly created individuals will be added to the island.

After evolving the islands some of the individuals of each island migrate to their neighbors within the ring topology. The migrating individuals as well as their migration destination (left or right neighbor in the ring) are chosen at random. Islands which receive migrating individuals replace randomly selected individuals within their current population with the immigrated individuals.

After the migration phase, each parameter vector $param_i$ is assigned the best fitness of an individual of the current population of $island_i$ as its own fitness value. Afterwards the fitness of each parameter vector $param_i$ is compared with the fitness of its neighbors in the ring topology. If the fitness of the neighbors isn't better than $param_i$'s own fitness $param_i$ remains unchanged. Otherwise the parameter values within $param_i$ are shifted into the direction of the parameter values of the fittest neighbor. This phase is denoted "neighbor orientation". Finally the parameter vectors are mutated themselves.

The main difference between algorithm SAIGA [19] and algorithm SAIGA-ghw is that SAIGA also applies selection and recombination to the parameter vectors and the parameter vectors swapped between the different islands. We omit the selection, recombination and swapping of parameter vectors. Instead we introduced the process of neighbor orientation. With the mutation of parameter vectors and neighbor orientation we would like to achieve the following effects:

1. Due to the mutation of parameter vectors the genetic algorithm also explores the space of possible parameter settings during the search.

2. By neighbor orientation the information on good parameter vectors should be propagated through the ring of islands.

7.2.2 Parameter Representation

Like in [19] we distinguish between the phenotypes of the parameters' population size n, crossover rate p_c, mutation rate p_m and tournament selection group size s and their genotypes n', p'_c, p'_m and s'. Each genotype is represented by a floating point number whithin range 0 and 1. The corresponding phenotypes are computed as follows:

			Phenotype Range
(1)	n	$= 20 + \lfloor exp(8 \cdot n' \cdot log(2)) \rfloor$	$[21, 276]$
(2)	p_c	$= p'_c$	$[0, 1]$
(3)	p_m	$= 0.00005 \cdot exp(p'_m \cdot log(1/0.0001))$	$[0.00005, 0.5]$
(4)	s	$= 2 * \lfloor s' * 4 \rfloor$	$[2, 6]$

Note that the phenotypes for population size (1) and for tournament selection group size (4) are computed differently as in [19].

7.2.3 Initialization of Parameters

The genotype parameters p'_m and s' are assigned uniformly distributed random numbers between 0 and 1, p'_c is assigned a uniformly distributed random number between 0.5 and 1. Then the corresponding phenotypes are computed. For the population size we initialize the phenotype parameter with a normally distributed random integer $N(100, 50)$. The genotype n' for n is computed afterwards.

7.2.4 Mutation of Parameter Vectors

Each single parameter is mutated at a probability of 60%. Mutation adds a normally distributed random number $N(0, 0.1)$ to the genotype parameter which is chosen for mutation. The corresponding phenotypes are computed afterwards. Figure 7.4 presents the pseudo code of the mutation of a parameter vector.

Algorithm: Mutate Parameter Vector

Input: a parameter vector $param_i$
Output: the mutated parameter vector

r = a uniform random number from $[0, 1]$
if $r < 0.6$ **then**
  $n' = n' + N(0, 0.1)$

r = a uniform random number from $[0, 1]$
if $r < 0.6$ **then**
  $s' = s' + N(0, 0.1)$

r = a uniform random number from $[0, 1]$
if $r < 0.6$ **then**
  $p'_c = p'_c + N(0, 0.1)$

r = a uniform random number from $[0, 1]$
if $r < 0.6$ **then**
  $p'_m = p'_m + N(0, 0.1)$

repair genotype parameters if they are not in $[0, 1]$
compute phenotypes

Figure 7.4: Mutation of parameter vector.

7.2.5 Neighbor Orientation

As already mentioned, each parameter vector $param_i$ is assigned the best fitness of an individual of the current population of $island_i$ as its own fitness value. If the fitness of some parameter vector of a neighboring island is better than $param_i$'s fitness, $param_i$ will be shifted into the direction of the parameter values of its fittest neighbor. This shift is done by reducing the difference in the values of the genotype parameters between $param_i$ and its fittest neighbor by 50%. For instance, let s'_i and $s'_{neighbor}$ be the genotype parameters for tournament selection group size of $island_i$ and its fitter neighbor. By setting $s'_i = (s'_i + s_{neighbor'})/2$ we reduce the difference between the parameter values by 50% and shift s'_i into the direction of $s'_{neighbor}$.

7.2.6 Further Details

Within an iteration of SAIGA-ghw each island executes iterations of GA-ghw until more than 1000 are evaluated. In each iteration of SAIGA-ghw 5% of the individuals of an island migrate.

7.2.7 Computational Results

Due to time restrictions, we tested algorithm SAIGA-ghw only on four hypergraphs from [22]. We ran SAIGA-ghw with $I = 20$ island GAs. As crossover operator we chose position-based crossover (POS), as mutation operator the insertion mutation operator (ISM). For each hypergraph we performed 10 runs. A single run lasted $max\_evaluations = 4,000,000$ total evaluations.

We tested the ten runs for each hypergraph either on a machine (1) with an Intel(R) Pentium(R)-4 3.40GHz processor having 1GB RAM or on a machine (2) with an Intel(R) Xeon(TM) 3.20GHz processor having 4GB RAM. Like in section 7.1.3 we converted the times of machine (1) into the times of machine (2).

Table 7.2 presents the results of algorithm SAIGA-ghw for the considered hypergraphs. The columns *Graph*, *V* and *H* present the graph name and the number of vertices and hyperedges of that graph. *min*, *max* and *avg* present the best, worst and average width returned by algorithm GA-ghw for an instance whereas *std.dev.* contains the standard deviation of the ten results returned by algorithm GA-ghw. Within the column *min-time* we present the time which was needed by algorithm GA-ghw for the run which returned the width in column *min*, column *avg-time* presents the average time of the ten runs.

If we compare the results of algorithm SAIGA-ghw on the four hypergraphs with the results delivered by algorithm GA-ghw (Table 7.1) we observe that SAIGA-ghw was

Hypergraph	*V*	*H*	*min*	*max*	*avg*	*std.dev.*	*min-time*	*avg-time*
adder_99	496	694	4	5	4.1	0.32	09:08:48	09:39:13
b06	48	50	4	4	4	0.00	00:20:22	00:30:50
b09	168	169	7	7	7	0.00	01:27:58	01:41:17
s444	202	205	5	6	5.6	0.52	02:10:09	02:18:24

Table 7.2: SAIGA-ghw results for selected benchmark hypergraphs.

able to obtain the same upper bounds on the generalized hypertree width of hypergraph b06 and b09 in each of the ten runs. For instance s444 SAIGA-ghw was also able to return a upper bound of 5 but, unlike algorithm GA-ghw, this upper bound was not returned in each run. When applied to hypergraph adder_99 SAIGA-ghw was not able to find the 3-width upper bound which was always returned by GA-ghw. Note also, that for each instance the average time needed by algorithm SAIGA-ghw exceeds the average time needed by algorithm GA-ghw.

We conclude that the upper bounds returned by algorithm SAIGA-ghw as well as its running time are slightly worse that the upper bounds delivered by algorithm GA-ghw and its time behavior. But we have to bear in mind that the results of GA-ghw in Table 7.1 preceded many time-consuming experiments in order to determine suitable control parameter values (see section 6.3). Thus algorithm SAIGA-ghw may be considered as an alternative to algorithm GA-ghw if we do not have the time for obtaining control parameter values in comprehensive experiments.

Chapter 8

A Branch and Bound Algorithm for Generalized Hypertree Width

In this chapter we propose a branch and bound algorithm which is able to compute the generalized hypertree width of a hypergraph. For that purpose we develop a lower bound heuristic for the generalized hypertree width of hypergraphs in section 8.1, and show how some of the graph reduction and pruning techniques presented in chapter 4 can be used and extended for our branch and bound algorithm in section 8.2 and section 8.3. Finally we present the branch and bound algorithm for computing the generalized hypertree width of hypergraphs in section 8.4, its implementation details in section 8.5 and the results it returned for selected hypergraph benchmarks in section 8.6.

8.1 A Lower Bound Heuristic for Generalized Hypertree Width

Within this section we show how lower bound heuristics for treewidth and lower bound heuristics for the k-set cover problem may be used for obtaining a lower bound heuristic on the generalized hypertree width of hypergraphs. Thus, first of all we give a short introduction on the k-set cover problem.

8.1.1 The *k*-Set Cover Problem

The *k*-set problem may be formulated as minimization problem as follows [21]:

Given	$T = \{t_1, ..., t_n\}$	a set
	$S = \{S_1, ..., S_m\}, \forall i : S_i \subseteq T$	a collection of subsets of T
	k	an integer

Find a subcollection of $C \subseteq S$ such that C covers at least k elements of T such that $|C|$ is minimal. Thus, an instance of the *k*-set cover problem may be represented as a triple $\langle T, S, k \rangle$.

k-Set Cover as Integer Program (IP)

The *k*-set cover problem is an $\mathcal{NP}$-complete problem. It may be formulated as integer programm (IP). An integer programming problem consists of a linear objective function and linear constraint equations and inequations. The variables of the objective function and of the constraints are required to be integers. The *k*-set cover problem may be formulated as integer program as shown below.

min	$\sum_{j=1}^{m} x_j$		
subject to:	$y_i + \sum_{j:t_i \in S_j} x_j \geq 1$	$i = 1, ..., n$	(1)-(n)
	$\sum_{i=1}^{n} y_i \leq n - k$		($n+1$)
	$x_j, y_i \in \{0, 1\}$	IP - program	
	$(x_j, y_i \geq 0$	LP - relaxation)	

This formulation was taken from [21]. We introduce a (binary) integer variable x_j for each subset in S. $x_j = 1$ iff subset S_j is a member of the the resulting cover. Moreover we introduce a (binary) integer variable y_i for each element $t_i \in T$. $y_i = 1$ iff t_i is not covered. The constraint inequation (n+1) says that there may be at most $n - k$ uncovered elements of S. The equations (1)-(n) specify which elements in T are covered by which subsets in S and ensure that each element must be covered by at least one subset or y_i has to be 1. The solution which satisfies all constraints and minimizes the objective function represents the solution for the *k*-set cover problem.

Although integer programming is known to be $\mathcal{NP}$-hard many problem instances formulated as integer program may be solved exactly by the help of IP-solvers. If we allow that the variables of a given integer program are real numbers we obtain the so-called LP-relaxation of the integer program. The exact solution to the LP-relaxation is a lower

bound for the exact solution of the corresponding integer program (for minimization problems as the k-set cover problem). The LP-relaxation belongs to the class of linear programs (LPs) in which all variables are real numbers. Solving a linear program (LP) is feasible in polynomial time, thus also computing the solution for the LP-relaxation of an integer program. For further information on integer and linear programing see [46].

Set Cover as IP-Program

Also the set cover problem (section 2.5.2) may be formulated as IP [21]. We will use the following IP-formulation of the set cover problem in order to cover the vertex sets derived from an elimination ordering exactly with an IP-solver.

$$\begin{aligned} \min \quad & \textstyle\sum_{j=1}^{m} x_j \\ \text{subject to:} \quad & \textstyle\sum_{j:t_i \in S_j} x_j \geq 1 \quad i = 1, ..., n \\ & x_j \in \{0, 1\} \end{aligned}$$

8.1.2 From Treewidth Lower Bounds to Generalized Hypertree Width Lower Bounds

Theorem 4. *Let lb be a lower bound for the treewidth of a hypergraph $\mathcal{H} = (V, H)$. Then the exact solution or any lower bound of the k-set cover problem $\langle V, H, lb + 1\rangle$ is a lower bound on the generalized hypertree width for $\mathcal{H}$, $ghw(\mathcal{H})$.*

Proof.

1. Let lb be a lower bound for the treewidth of a hypergraph $\mathcal{H} = (V, H)$. Then we know that every tree decomposition for $\mathcal{H}$ has a vertex p with at least $k = lb + 1$ vertices in its label $\chi(p)$.
2. Every generalized hypertree decomposition for a hypergraph $\mathcal{H}$ is also a tree decomposition for $\mathcal{H}$. Thus it must be that every generalized hypertree decomposition, also one having a width of $ghw(\mathcal{H})$, has a vertex p with at least $k = lb + 1$ vertices in its label $\chi(p)$.
3. Thus, the minimum number of hyperedges needed to cover at least $k = lb + 1$ vertices of $\mathcal{H}$ is a lower bound for the generalized hypertree width of $\mathcal{H}$.

4. The problem of finding the minimum number of hyperedges needed to cover at least $k = lb+1$ vertices can be formulated as an instance of the k-set cover problem with V as basic set, H as collection of subsets of V and $lb + 1$ as the minimum number of elements that have to be covered, $\langle T = V, S = H, k = lb + 1\rangle$.

5. Thus, the exact solution or any lower bound for the k-set cover problem $\langle T = V, S = H, k = lb + 1\rangle$ represents a lower bound for $ghw(\mathcal{H})$.

□

On the basis of Theorem 4 we developed a lower bound heuristic for the generalized hypertree width of a hypergraph $\mathcal{H}$. We named the heuristic *tw-ksc-width* because it combines lower bound heuristics for treewidth of hypergraphs with a lower bound heuristic for the k-set cover problem. Figure 8.1 presents the pseudo code for tw-ksc-width.

From Lemma 1 [33] we obtain that the lower bound of the treewidth of a hypergraph's primal graph is also lower bound of the treewidth of the hypergraph. For that reason tw-ksc-width applies the minor-γ_R and the minor-min-width heuristics (section 4.4.2) to the primal graph $G^*(\mathcal{H})$. A variable k is assigned the maximum of the values returned by the two heuristics plus one. Afterwards tw-ksc-width computes a solution for the LP-relaxation of the k-set cover problem $\langle V, H, k\rangle$. The returned solution of that problem is ceiled because the exact solution for a k-set cover problem must be an integer value. This ceiled value is returned as a lower bound on the $ghw(\mathcal{H})$.

Algorithm: tw-ksc-width

Input: a hypergraph $\mathcal{H} = (V, H)$
Output: a lower bound for $ghw(\mathcal{H})$

1. $k = max(minor\text{-}min\text{-}width(G^*(\mathcal{H})), \text{minor-}\gamma_R(G^*(\mathcal{H})) + 1$
2. lp_relax = the solution for the LP-relaxation of the k-set cover problem $\langle V, H, k\rangle$
3. $lb = \lceil lp\_relax \rceil$
4. **return** lb

Figure 8.1: Algorithm tw-ksc-width.

8.2 Reduction Techniques

Definition 26 (Simplicial Vertex in Hypergraph)**.** A vertex v of a hypergraph $\mathcal{H}$ is simplicial if each pair of its neighbors appear together within a hyperedge.

Lemma 14. *Let v be a simplicial vertex of hypergraph $\mathcal{H}$ and let $\mathcal{H}-\{v\}$ be the hypergraph obtained by deleting v from $\mathcal{H}$. Then it holds that $ghw(\mathcal{H}-\{v\}) \leq ghw(\mathcal{H})$ and that $ghw(\mathcal{H}) = max(k, ghw(\mathcal{H}-\{v\}))$, where k is the minimum number of hyperedges needed to cover v and its neighbors in $\mathcal{H}$.*

Proof. Let $\langle T, \chi, \lambda\rangle$ be a generalized hypertree decomposition of $\mathcal{H}$ whose width equals $ghw(\mathcal{H})$. Obviously by deleting v from the χ-sets of $\langle T, \chi, \lambda\rangle$ we obtain a generalized hypertree decomposition of $\mathcal{H}-\{v\}$ of width at most $ghw(\mathcal{H})$.

In chapter 3 we proved the a generalized hypertree decomposition whose width equals $ghw(\mathcal{H})$ can be obtained from an elimination ordering σ. Whenever the first vertex of the set consisting of v and its neighbors is eliminated according to σ a χ-set will be produced in the resulting generalized hypertree decomposition containing v and its neighbors. At least k hyperedges are needed to cover that χ-set. It follows that $ghw(\mathcal{H}) = max(k, ghw(\mathcal{H}-\{v\}))$. □

The above lemma implies that whenever a simplicial vertex appears in a hypergraph associated with a subproblem within a branch and bound search the simplicial vertex may be removed in the next step. Note that a vertex v is a simplicial vertex of a hypergraph $\mathcal{H}$ iff v is a simplicial vertex of the primal graph $G^*(\mathcal{H})$ since the adjacency relations in $\mathcal{H}$ and $G^*(\mathcal{H})$ are equivalent. This result is important for the branch and bound algorithm we will propose in this chapter because it will be based on the primal graph $G^*(\mathcal{H})$.

8.3 Pruning Rules

In section 4.4.5 we described two pruning rules from the branch and bound algorithm for treewidth in [5]. Now we examine if and how these rules may be modified and used for a branch and bound algorithm which computes the generalized hypertree width of a hypergraph. Our branch and bound algorithm will use the tree representing all possible elimination orderings as search tree. For obtaining the vertex sets which are produced by elimination orderings we will eliminate vertices from the primal graph of the input hypergraph.

Suppose that we are in a node of the branch and bound tree, let g be the width of the partial solution represented by that node and let n' denote the number of vertices that haven't been eliminated yet. We know that the width of a solution within the subtree rooted at the current search node may be at most $w = max(g, n')$ because we need at most n' hyperedges in order to cover the n' vertices. If we solve the set cover problem for the n' remaining vertices of the hypergraph exactly or we compute an upper bound on that set cover problem and it turns out that we need k hyperedges in order to cover the n' vertices we conclude that the width of a solution within the subtree rooted at the current search node may be at most $w = max(g, k)$.

If w is smaller then the width of the best solution found so far, $w < ub$, we know that we will find a better solution within the subtree rooted at the current search node. There are two cases. If $n' \leq g$ and $k \leq g$ respectively, then we don't have to continue the search within the subtree. If $n' > g$ and $k > g$ respectively then at least one solution within the subtree will lead to a new upper bound of at most n' and k respectively but we have to continue the search within the subtree for finding its best solution.

We conclude that we are able to derive two pruning rules for a branch and bound algorithm for generalized hypertree width from Pruning Rule 1 in section 4.4.5 [5]. The two rules are denoted Pruning Rule 1 (PR 1) and Pruning Rule 1' (PR 1') and are given below.

Pruning Rule 1 (PR 1)
compute $w := max(g, n')$
if $w < ub$ then $ub = w$
  if $n' \leq g$ then exclude the subtree rooted at the current node from the search

Pruning Rule 1' (PR 1')
compute an exact solution or an upper bound k for the set cover problem defined by the n' remaining vertices and H
compute $w := max(g, k)$
if $w < ub$ then $ub = w$
  if $k \leq g$ then exclude the subtree rooted at the current node from the search

From Pruning Rule 2 in 4.4.5 [5] we are able derive two pruning rules for a branch and bound algorithm for generalized hypertree width. The two rules are denoted Pruning Rule 2a (PR 2a) and Pruning Rule 2b (PR 2b) and are given below. From Figure 8.2 we see why Pruning Rule 2a and Pruning Rule 2b are applicable for a branch and bound algorithm for generalized hypertree width. No matter if we eliminate v before w or w before v we obtain the same graphs after the elimination of the two vertices.

If v and w are not adjacent the same vertex sets are created when they are eliminated and thus the order in which v and w are eliminated doesn't have an effect on the width. These considerations are expressed within Pruning Rule 2a.

If v and w are adjacent then when eliminating the vertex v before w we obtain the sets consisting of $v, N_v, N_{v,w}$ and $w, N_v, N_w, N_{v,w}$ and when eliminating vertex w before v we have $w, N_w, N_{v,w}$ and $v, N_v, N_w, N_{v,w}$ in the created sets (see Figure 8.2). Thus we have to check whether $v, N_v, N_{v,w}$ and $w, N_v, N_w, N_{v,w}$ or $w, N_w, N_{v,w}$ and $v, N_v, N_w, N_{v,w}$ may be covered with fewer hyperedges. These considerations are expressed within Pruning Rule 2b.

Pruning Rule 2a (PR 2a)

Suppose v and w are successive vertices in an elimination ordering σ, and v and w are not adjacent in the graph obtained by eliminating the vertices in σ up to v. Then the ordering σ', obtained by swapping v and w in σ, has the same width as σ. Thus, we prune the search tree as follows: for such a pair of vertices v, w, when we have looked at a branch representing the elimination orderings ending with $w, v, x_i, ..., x_n$ we prune the branch representing the orderings ending with $v, w, x_i, ..., x_n$.

Pruning Rule 2b (PR 2b)

Suppose v and w are successive vertices in an elimination ordering σ with $w <_\sigma v$ and v and w are adjacent in the graph obtained by eliminating the vertices in σ up to v. Let σ' be the ordering obtained by swapping v and w in σ. Let $cover_{w,v}$ be the minimum number of hyperedges needed to cover one of the two sets created when v is eliminated before w and let $cover_{v,w}$ be the minimum number of hyperedges needed to cover one of the two sets created when w is eliminated before v. If $cover_{w,v} \leq cover_{v,w}$ then the width of σ doesn't exceed the width of σ'. Thus, we prune the search tree as follows: for such a pair of vertices v, w with $cover_{w,v} \leq cover_{v,w}$, we explore the branch representing the elimination orderings ending with $w, v, x_i, ..., x_n$ and we prune the branch representing the orderings ending with $v, w, x_i, ..., x_n$. If $cover_{w,v} > cover_{v,w}$ we explore the branch representing the orderings ending with $v, w, x_i, ..., x_n$ and we prune the branch representing the elimination orderings ending with $w, v, x_i, ..., x_n$.

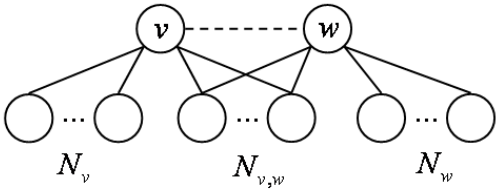

Figure 8.2: Example graph [5] for pruning rules PR 2a and P2 2b.

8.4 Algorithm BB-ghw

We introduce a branch and bound algorithm which is able to compute the generalized hypertree width of a given hypergraph $\mathcal{H}$. The branch and bound algorithm searches the branch and bound tree representing all elimination orderings of the vertices in $\mathcal{H}$. The algorithm uses the hypergraph's primal graph $G^*(\mathcal{H})$ in order to create the vertex sets that are produced according to a specific elimination ordering. We will exploit the pruning rules 1 (PR 1) and 2a (PR 2a) from section 8.3 as well as the simplicial vertex reduction rule from section 8.2 in order to shrink the search space. The resulting algorithm is named *BB-ghw*. Figure 8.3 shows algorithm BB-ghw in pseudo code notation.

Within algorithm BB-ghw we will use an array of search nodes, denoted *stack*, for storing the nodes of the current search path. A search node contains three variables *g*,*h* and f, where g is the width of the partial solution represented by the search node, h is the lower bound on the graph obtained after eliminating the vertices of the partial solution and f is a lower bound on the width of any elimination ordering ending with that partial solution. In addition, each search node s is associated a vertex, denoted *vertex*, representing the vertex which is eliminated at the search node and a list of the vertices representing the vertices which are eliminated at the child nodes of s within the search tree, denoted *children*. We say that a node is explored if all its children have been visited.

First of all Algorithm BB-ghw computes the primal graph $G^*(\mathcal{H})$ for its input hypergraph $\mathcal{H} = (V, H)$. Then a lower *lb* and an upper bound *ub* on $ghw(\mathcal{H})$ are computed from the primal graph. For computing the lower bound we use heuristic tw-ksc-width from section 8.1.2. For getting an upper bound on $ghw(\mathcal{H})$ we compute a tree decomposition for $\mathcal{H}$ by the help of the min-fill heuristic from section 4.4.2 and compute a minimum cover for the labels of the tree decomposition vertices [37]. The width of the resulting generalized hypertree decomposition acts as initial upper bound.

If the upper bound equals the lower bound we return *ub* as the generalized hypertree width. Otherwise we initialize the search node r representing the root node of the search tree by setting $r.g = 0$, $r.h = lb$ and $r.f = lb$. If there is a simplicial or vertex in $G^*(\mathcal{H})$ this vertex is the only child of the root node. If there is no such vertex there is a child for each vertex within the hypergraph.

Afterwards the branch and bound search begins and lasts until the whole search tree has been explored or we have found an upper bound which equals our initial lower bound *lb*. In each iteration we regard the last node of our current search path, denoted s. This node may be either a node representing a solution or a node which has at least one unexplored child or an explored node.

If s represents a solution we examine if the width of that solution is smaller than the upper bound we have found so far. If that is the case that width is stored in *ub* as new

upper bound. Finally the vertex that has been eliminated at this node is restored within the primal graph.

If s has at least one unexplored child, denoted t, we will visit that child node. Let v denote the vertex which will be eliminated at node t, thus $t.vertex = v$. When eliminating v from $G^*(\mathcal{H})$ we get a set of vertices X containing v and its neighbors. BB-ghw computes the minimum number k of hyperedges in $\mathcal{H}$ needed for covering the vertices in X. The width of the partial solution represented by t is the maximum of the width of the partial solution in s and k, thus $t.g = max(s.g, k)$. We compute a lower bound on the generalized hypertree width of $G^*(\mathcal{H})$ with tw-ksc-width and assign it to $t.h$. A lower bound on the width of any elimination ordering ending with the partial solution represented by t is the maximum of the width computed so far $t.g$, the lower bound on the generalized hypertree width of the remaining graph $t.h$ and any lower bound of a search state on the path from the root to t, thus $t.f = max(s.f, t.g, t.h)$. If $t.f \geq ub$ then BB-ghw won't find a solution better than ub in the subtree rooted at the current search node and thus that subtree is pruned. Otherwise the children of t are computed according to the pruning rules 1 and 2a and the simplicial vertex reduction rule.

If s has been explored the vertex $s.vertex$ that has been eliminated at s is restored in $G^*(\mathcal{H})$.

If BB-ghw terminates it will return ub whose value is the generalized hypertree width of $\mathcal{H}$.

8.5 Implementation Details

We implemented algorithm BB-ghw using C++ and STL. For representing and processing the primal graph of the input hypergraph we chose the graph representation proposed in section 5.2.1. Since this representation requires $O(|V|^2)$ space and also the space needed by the states within *stack* is in $O(|V|^2)$ the space complexity of BB-ghw is $O(|V|^2)$.

For computing the solution of the LP-relaxation of the k-set cover problem of the lower bound heuristic tw-ksc-width as well as for solving the set cover problems appearing within the branch and bound search we used GNU Linear Programming Kit 4.9 (GLPK) [23].

Algorithm: BB-ghw

Input: a hypergraph $\mathcal{H} = (V, H)$
Output: the generalized hypertree width of the $\mathcal{H}$

compute $G^*(\mathcal{H})$
$lb = lower\_bound(G^*(\mathcal{H}))$
$ub = upper\_bound(G^*(\mathcal{H}))$
if $lb = ub$ **then return** ub */* ghw($\mathcal{H}$) already found */*

/ initialize the root of the search tree */*
$depth = 0$
$r = stack[depth]$
$r.h = lb, r.g = 0, r.f = lb, r.vertex = \emptyset$
if $\exists$ a simplicial vertex w in $G^*(\mathcal{H})$ **then**
  $r.children = \{w\}$
else $r.children = V$

while $(depth > -1)$ and $(ub > lb)$ **do** */* branch and bound search */*

  $s = stack[depth]$
  if $depth = |V| - 1$ **then** */* s is a leaf of the branch and bound tree - solution */*
    if $s.f < ub$ **then**
      $ub = s.f$
    $restore(s.vertex, G^*(\mathcal{H}))$
    $depth = depth - 1$

  else if $\exists\, v \in s.children$ **then** */* s has at least one unexplored child - branching*/*
    $depth = depth + 1$
    $t = stack[depth]$
    $t.vertex = v$

    determine $t.children$ according to pruning rule PR 2a
    $X = eliminate(v, G^*(\mathcal{H}))$
    k = exact set cover (X, H)

    if $\exists$ a simplicial vertex w in $G^*(\mathcal{H})$ **then** /* simplicial vertex reduction rule */
      $r.children = \{w\}$

    $t.g = max(s.g, k)$
    $t.h = lower\_bound(G^*(\mathcal{H}))$
    $t.f = max(s.f, t.g, t.h)$
    if $t.f \geq ub$ **then** $t.children = \emptyset$ */* Bounding */*

    / Pruning according PR 1*/*
    $n' = |V| - depth$ */* remaining vertices */*
    $w = max(t.g, n')$
    if $w < ub$ **then**
      $ub = w$
      if $n' \leq t.g$ **then**
        $t.children = \emptyset$ */* prune search */*

    remove v from $t.children$

  else */* all children of s have been explored - one step back in the search tree */*
    $restore(s.vertex, G^*(\mathcal{H}))$
    $depth = depth - 1$

return ub

Figure 8.3: Algorithm BB-ghw.

8.6 Computational Results

We tested algorithm BB-ghw on 95 hypergraphs from [22] on a machine with an Intel(R) Xeon(TM) 3.20 GHz processor having 4GB RAM. We implemented BB-ghw as randomized algorithm thus we performed ten runs of BB-ghw for each hypergraph. Each run was given a one hour time limit. Table 8.1 and Table 8.2 present the results BB-ghw returned for the 95 hypergraphs. The columns *Hypergraph, V* and *H* contain the name, the number of vertices and hyperedges for each hypergraph. Column *lb* presents the maximum lower bound returned by the lower bound heuristic tw-ksc-width in ten runs. Column *ub* gives the best upper bound on the generalized hypertree width of a hypergraph which has been reported in [17], "*"-entries indicate that no result was available. Column *min-fill* presents the minimum upper bound which was computed by the min-fill heuristic in ten runs for each instance. Column *BB−ghw* gives the best width reported by algorithm BB-ghw, the columns *avg*. and *std.dev*. contain the average width and the standard deviation for the ten runs. Column *time* reports the time of a run which was able to return an exact solution for an instance, "*"-entries indicate that the time limit of an hour was exceeded.

Algorithm BB-ghw was able to compute the gen. hypertree width for 23 instances. For other 21 hypergraphs BB-ghw returned an improved upper bound on the generalized hypertree width, which for 7 hypergraphs was due to an improved upper bound returned by the min-fill heuristic. Anyway for 14 hypergraphs the improved upper bound was found within the branch and bound search. For 40 BB-ghw could return an upper bound which was equal to the best upper bound known so far for that instance. Only for eight instances the best known upper bound was not reached.

Hypergraph	*V*	*H*	*lb*	*ub*	*min-fill*	*BB-ghw*	*avg.*	*std.dev.*	*time*
2bitcomp_5	95	310	3	11	13	13	14	1.41	*
2bitmax_6	192	766	5	15	13	13	25.3	6.60	*
adder_15	106	76	2	2	2	2	2	0.00	0.03
adder_25	176	126	2	2	2	2	2	0.00	0.06
adder_50	351	251	2	2	2	2	2	0.00	0.19
adder_75	526	376	2	2	3	2	2	0.00	30.31
adder_99	694	496	2	2	3	2	2	0.00	66.7
aim-50-1_6-no-3	50	80	4	9	9	9	9	0.00	*
aim-50-1_6-yes1-3	50	80	4	10	10	9	9.8	0.42	*
aim-50-2_0-no-3	50	100	5	12	12	11	11.8	0.42	*
aim-50-2_0-yes1-3	50	100	5	11	11	10	10.8	0.42	*
aim-50-3_4-yes1-3	50	170	7	13	12	12	12	0.00	*
ais6	61	581	5	10	10	9	9.6	0.70	*
ais8	113	1520	5	14	14	12	14.2	0.92	*
atv_partial_system	125	88	2	3	4	3	3.7	0.48	*
b01	47	45	2	5	5	5	5.2	0.42	*
b02	27	26	2	3	3	3	3	0.00	*
b03	156	152	2	7	7	7	7	0.00	*
b06	50	48	2	5	5	4	4.7	0.48	*
b08	179	170	3	10	10	10	10	0.00	*
b09	169	168	3	10	10	10	10.2	0.63	*
b10	200	189	3	14	13	13	14.2	0.79	*
bridge_15	137	137	2	2	3	3	3.2	0.42	*
bridge_25	227	227	2	2	3	3	3.7	0.48	*
c432	196	160	2	9	9	9	9	0.00	*
c499	243	202	3	13	13	12	15	2.16	*
clique_10	45	10	2	5	6	5	5	0.00	*
clique_15	105	15	2	8	8	8	8	0.00	*
clique_20	190	20	3	10	12	10	10	0.00	*
dubois100	300	800	2	2	2	2	2	0.00	0.34
dubois20	60	160	2	2	2	2	2	0.00	0.02
dubois21	63	168	2	2	2	2	2	0.00	0.03
dubois22	66	176	2	2	2	2	2	0.00	0.02
dubois23	69	184	2	2	2	2	2	0.00	0.03
dubois24	72	192	2	2	2	2	2	0.00	0.02
dubois25	75	200	2	2	2	2	2	0.00	0.03
dubois26	78	208	2	2	2	2	2	0.00	0.04
dubois27	81	216	2	2	2	2	2	0.00	0.04
dubois28	84	224	2	2	2	2	2	0.00	0.04
dubois29	87	232	2	2	2	2	2	0.00	0.04
dubois30	90	240	2	2	2	2	2	0.00	0.04
dubois50	150	400	2	2	2	2	2	0.00	0.1
flat30-1	90	300	5	10	12	12	14.4	1.84	*
flat30-50	90	300	5	11	11	11	11.4	0.52	*
flat30-99	90	300	5	11	14	14	15.9	1.20	*
grid10	100	180	3	*	12	9	10.2	1.03	*
grid15	225	420	3	*	20	16	18.1	1.10	*
grid20	400	760	3	*	26	21	24.9	2.18	*
grid2d_10	50	50	3	5	5	5	5	0.00	*
grid2d_15	113	112	3	8	9	8	9	0.47	*
grid2d_20	200	200	3	11	12	12	12.7	0.48	*
grid3d_4	32	32	2	6	6	5	5	0.00	*
grid3d_5	63	62	3	8	8	8	8.2	0.42	*
grid3d_6	108	108	3	12	13	12	12.9	0.32	*
grid3d_7	172	171	4	16	17	17	18.1	0.74	*
grid4d_3	41	40	3	6	8	6	6.9	0.32	*
grid4d_4	128	128	4	17	16	15	16.2	0.63	*
grid5	25	40	3	*	4	3	3.2	0.42	0.02

Table 8.1: BB-ghw results for selected benchmark hypergraphs.

Hypergraph	*V*	*H*	*lb*	*ub*	*min-fill*	*BB-ghw*	*avg.*	*std.dev.*	*time*
hole10	110	561	3	11	11	11	11	0.00	*
hole6	42	133	3	7	7	7	7	0.00	*
hole7	56	204	3	8	8	8	8	0.00	*
hole8	72	297	3	9	9	9	9	0.00	*
hole9	90	415	3	10	10	10	10	0.00	*
par8-1-c	64	254	3	7	7	7	7.1	0.32	*
par8-2-c	68	270	4	6	6	6	6.5	0.53	*
par8-3-c	75	298	4	8	7	7	7	0.00	*
par8-4-c	67	266	3	7	7	6	7.1	0.57	*
par8-5-c	75	298	4	7	7	7	7	0.00	*
pret150_25	150	400	2	5	5	5	5	0.00	*
pret150_40	150	400	2	5	5	5	5	0.00	*
pret150_60	150	400	2	5	5	5	5	0.00	*
pret150_75	150	400	2	5	5	5	5	0.00	*
pret60_25	60	160	2	5	5	5	5	0.00	*
pret60_40	60	160	2	5	5	5	5	0.00	*
pret60_60	60	160	2	5	5	5	5	0.00	*
pret60_75	60	160	2	5	5	5	5	0.00	*
s208	115	104	2	7	7	7	7.1	0.32	*
s27	17	13	2	2	2	2	2	0.00	0
s298	136	133	3	5	5	5	5	0.00	*
s344	184	175	2	7	7	7	7	0.00	*
s349	185	176	2	7	7	7	7	0.00	*
s382	182	179	3	5	5	5	5.4	0.52	*
s386	172	165	4	8	8	7	7.9	0.32	*
s400	186	183	3	6	6	5	5.9	0.32	*
s420	231	212	2	9	9	9	9.8	0.79	*
s444	205	202	3	6	6	5	5.8	0.42	*
s510	236	217	4	23	20	20	22.3	1.64	*
s526	217	214	3	8	8	7	7.9	0.32	*
s641	433	398	3	7	8	8	8.5	0.85	*
s713	447	412	3	7	8	7	8.8	1.03	*
s820	312	294	5	13	12	12	12	0.00	*
s832	310	292	5	12	11	11	11	0.00	*
uf20-01	20	91	5	6	6	6	6	0.00	0.42
uf20-050	20	91	6	6	6	6	6	0.00	0.07
uf20-099	20	91	5	6	6	6	6	0.00	0.79

Table 8.2: BB-ghw results for selected benchmark hypergraphs.

Chapter 9

An A* Algorithm for Generalized Hypertree Width

In this chapter we present an A* algorithm for computing the generalized hypertree width of hypergraphs. The algorithm uses the lower bound heuristic, the graph reduction technique and a pruning rule presented in the previous chapter.

9.1 Algorithm A*-ghw

Basically, the A* algorithm for computing the generalized hypertree width of hypergraphs, named *A*-ghw*, has the same structure as algorithm A*-tw in section 5.1. Figure 9.1 presents algorithm A*-ghw in pseudo code notation.

The A* algorithm explores the search tree representing all elimination orderings of the vertices of its input hypergraph $\mathcal{H} = (V, H)$. In addition, A*-ghw uses the reduction rule for simplicial vertices from section 8.2 and pruning rule 2a from section 8.3 in order to shrink the search space. For getting an upper bound on $ghw(\mathcal{H})$ we compute a tree decomposition for $\mathcal{H}$ by the help of the min-fill heuristic from section 4.4.2 and compute a minimum cover for the labels of the tree decomposition vertices [37]. The width of the resulting generalized hypertree decomposition acts as initial upper bound. As lower bound heuristic we take the tw-ksc-width heuristic from section 8.1.

The algorithm uses a single priority queue, denoted *queue*, for storing search states, representing the nodes of the search tree. A search state contains the variables g, h, f, where g is the width of the partial solution represented by the search state, h is the lower bound on the graph obtained by eliminating the vertices of the partial solution and f is a lower bound on the width of all elimination orderings ending with the partial solution.

Furthermore a state contains links to its *children* within the search tree. Such a link is represented by the vertex which will be eliminated next in the child state. We say that we visit a state if we remove the state from the priority queue and by evaluating a state we mean that we assign to it the values for g, h, f and its children before inserting it into the priority queue. The queue orders the states after their value f in ascending order. Among states with the same value for f priority is given to those states which lie deeper in the search tree in the hope that we will reach the goal state earlier.

First of all the primal graph $G^*(\mathcal{H})$ is derived from the input hypergraph $\mathcal{H}$. Then an upper and a lower bound on the generalized hypertree width of the hypergraph are computed from the primal graph. If the upper bound on the instance equals the lower bound it is returned as $ghw(\mathcal{H})$. Otherwise we evaluate the initial state representing the root of the search tree by setting $g = 0$, and by assigning the value for the lower bound to h and f. If there is a simplicial vertex this vertex is the only child of the root state. If there is no such vertex there is a child for each vertex within the graph. Finally the initial state is inserted into the priority queue and the A* search begins.

During an iteration of the A* search the state s at top of the priority queue, having the lowest value for f, is visited. We create a graph G^s representing the graph that is obtained by eliminating the vertices of the partial solution represented by s. If $s.g \geq |G^s|$ we have visited a state representing a solution and thus we return $s.g$ as $ghw(\mathcal{H})$. Otherwise the children of s are evaluated and inserted into the priority queue.

For each child state t and its associated vertex v we compute its children according to pruning rule 2a. Afterwards we determine X, a set consisting of v and its neighbors in in G^s, and compute the minimum number k of hyperedges in H needed for covering the vertices in X. Then by eliminating v from G^s we obtain the graph G^s_v.

The width of the partial solution represented by t is the maximum of the width of the partial solution represented by s and d, thus $t.g = max(s.g, d)$. $t.h$ is assigned a lower bound on the treewidth of graph G^s_v. Both $t.g$ as well as $t.h$ represent a lower bound on the width of all elimination orderings ending with the partial solution represented by t as well as any lower bound of a search state on the path from the root to t, thus we set $t.f = max(t.g, t.h, s.f)$. If there is a simplicial vertex within G^s_v this vertex is the only child of state t.

Finally we insert t into the priority queue if $t.f$ is less than the upper bound ub on $ghw(\mathcal{H})$. States with $t.f \geq ub$ won't lead to solutions which are better than the upper bound solution we have already computed therefore they are excluded from the search in order to decrease the memory needed by the A* algorithm.

If all search states with $f < ub$ have been visited but none of them represented a solution it must be that ub is the treewidth of the graph and thus ub is returned by the algorithm in that case.

Algorithm: A*-ghw

Input: a hypergraph $\mathcal{H} = (V, H)$
Output: the generalized hypertree width of the $\mathcal{H}$

compute $G = G^*(\mathcal{H})$
$lb = lower\_bound(G)$
$ub = upper\_bound(G)$
if $lb = ub$ **then return** ub /* *treewidth already found* */

/* *evaluate the root of the search tree* */
$r = new\ State()$
$r.h = lb, r.g = 0, r.f = lb$

if $\exists$ a simplicial vertex w in G **then**
  $r.children = \{w\}$
  $r.reduced = true$
else $r.children = V$
$queue.push(r)$

/* *A* search - visit next state in queue* */
while *queue* is not empty **do**

  $s = queue.pop()$
  create G^s

  /* *new lower bound is found* */
  if $s.f > lb$ **then** $lb = s.f$

  /* *optimal solution is found* */
  if $s.g \geq |G^s|$ **then return** $s.g$

  /* *evaluate the children of current search state* */
  for each $v \in s.children$ **do**

    $t = new\ State()$
    $t.children = V(G^s) - \{v\}$
    if not $s.reduced$ **then** prune $t.children$ according to PR 2a

    $X = v \cup N_{G^s}(v)$
    $k =$ exact set cover (X, H)
    $G^s_v = eliminate(v, G^s)$
    $t.g = max(s.g, k)$
    $t.h = lower\_bound(G^s_v)$
    $t.f = max(t.g, t.h, s.f)$

    if $\exists$ a simplicial vertex w in G^s_v **then**
      $t.children = \{w\}$
      $t.reduced = true$

    if $t.f < ub$ **then** $queue.insert(t)$

return ub

Figure 9.1: Algorithm A*-ghw.

9.2 Implementation Details

We implemented algorithm A*-ghw using C++ and STL. For representing and processing the primal graph of the input hypergraph we chose the graph representation proposed in section 5.2.1.

For computing the solution of the LP-relaxion of the k-set cover problem of the lower bound heuristic tw-ksc-width as well as for solving the set cover problems appearing within the A* search we used GNU Linear Programming Kit 4.9 (GLPK) [23].

If we restrict the running time of the A* algorithm by a time limit then the f-value of the last state visited before the limit was exceeded may act as a lower bound on the treewidth of a graph for the same reasons as mentioned in 5.3.

9.3 Computational Results

We tested algorithm A*-ghw on 87 hypergraphs from [22]. All experiments were run on a machine with an Intel(R) Pentium(R)-4 3.40 GHz processor having 1 GB RAM. Since the lower and upper bound heuristics of A*-ghw are implemented in randomized fashion we performed ten runs of A*-ghw for each hypergraph. Each run was given a one hour time limit. Table 9.1 and Table 9.2 present the results which were returned by A*-ghw for the 87 hypergraphs.

The columns *Hypergraph*, *V* and *H* contain the name, the number of vertices and hyperedges for each hypergraph. Column *lb* presents the maximum lower bound returned by the lower bound heuristic tw-ksc-width in ten runs for the initial hypergraph. Column A^**-ghw* gives the maximum value returned by algorithm A*-ghw in ten runs. Column *time* reports the time of a run which was able to return an exact solution for an instance, "*"-entries indicate that the time limit of an hour was exceeded.

Algorithm A*-ghw was able to compute the generalized hypertree width for 19 instances whereas for other 9 hypergraphs A*-ghw was able to improve the initial lower bound. For the remaining 59 instances A*-ghw was not able to improve the quality of the initial lower bound.

Hypergraph	*V*	*H*	*lb*	*A*-ghw*	*time*
2bitcomp_5	95	310	4	4	*
2bitmax_6	192	766	5	5	*
adder_15	106	76	2	2	4.450
adder_25	176	126	2	2	0.060
aim-50-1_6-no-3	50	80	4	4	*
aim-50-1_6-yes1-3	50	80	4	4	*
aim-50-2_0-no-3	50	100	5	6	*
aim-50-2_0-yes1-3	50	100	5	5	*
aim-50-3_4-yes1-3	50	170	7	8	*
ais6	61	581	5	5	*
ais8	113	1520	5	6	*
atv_partial_system	125	88	2	2	*
b01	47	45	2	2	*
b02	27	26	2	2	*
b03	156	152	2	2	*
b06	50	48	2	3	*
b08	179	170	3	3	*
b09	169	168	3	3	*
b10	200	189	3	3	*
bridge_15	137	137	2	2	*
bridge_25	227	227	2	2	*
c432	196	160	2	2	*
c499	243	202	3	3	*
clique_10	45	10	2	3	*
clique_15	105	15	2	3	*
clique_20	190	20	3	3	*
dubois100	300	800	2	2	0.330
dubois20	60	160	2	2	0.020
dubois21	63	168	2	2	0.020
dubois22	66	176	2	2	0.020
dubois23	69	184	2	2	0.020
dubois24	72	192	2	2	0.030
dubois25	75	200	2	2	0.020
dubois26	78	208	2	2	0.030
dubois27	81	216	2	2	0.030
dubois28	84	224	2	2	0.040
dubois29	87	232	2	2	0.040
dubois30	90	240	2	2	0.040
dubois50	150	400	2	2	0.100
flat30-1	90	300	5	5	*
flat30-50	90	300	5	5	*
flat30-99	90	300	5	5	*
grid10	100	180	3	3	*
grid15	225	420	3	3	*
grid2d_10	50	50	3	3	*
grid2d_15	113	112	3	3	*
grid2d_20	200	200	3	3	*
grid3d_4	32	32	2	4	*
grid3d_5	63	62	3	4	*
grid3d_6	108	108	3	3	*
grid3d_7	172	171	4	4	*
grid4d_3	41	40	3	4	*
grid4d_4	128	128	4	4	*
grid5	25	40	3	3	538.7

Table 9.1: A*-ghw results for selected benchmark hypergraphs.

Hypergraph	*V*	*H*	*lb*	*A*-ghw*	*time*
hole10	110	561	3	6	*
hole6	42	133	3	4	*
hole7	56	204	3	5	*
hole8	72	297	3	5	*
hole9	90	415	3	6	*
par8-1-c	64	254	3	3	*
par8-2-c	68	270	4	4	*
par8-3-c	75	298	4	4	*
par8-4-c	67	266	3	3	*
par8-5-c	75	298	4	4	*
pret150_25	150	400	2	2	*
pret150_40	150	400	2	2	*
pret150_60	150	400	2	2	*
pret150_75	150	400	2	2	*
pret60_25	60	160	2	2	*
pret60_40	60	160	2	2	*
pret60_60	60	160	2	2	*
pret60_75	60	160	2	2	*
s208	115	104	2	2	*
s27	17	13	2	2	0.00
s298	136	133	3	3	*
s344	184	175	2	2	*
s349	185	176	2	2	*
s382	182	179	3	3	*
s386	172	165	4	4	*
s400	186	183	3	3	*
s420	231	212	2	2	*
s444	205	202	3	3	*
s510	236	217	4	4	*
s526	217	214	3	3	*
uf20-01	20	91	5	6	0.380
uf20-050	20	91	6	6	0.080
uf20-099	20	91	5	6	0.760

Table 9.2: A*-ghw results for selected benchmark hypergraphs.

Chapter 10

Conclusions

In this master thesis we presented new heuristic methods for tree decompositions and generalized hypertree decompositions.

In chapter 5 we proposed an A* algorithm, named A*-tw, for computing the treewidth of graphs which additionally applies reduction and pruning methods presented in [5], [8] and [24] in order to narrow the search space which has to be explored. Computational experiments revealed that A*-tw was able to compute the exact treewidth for all but two benchmark instances [18] for which the branch and bound bound algorithms in [5] and [24] could determine the treewidth. A*-tw could fix the treewidth for an additional instance.

In chapter 6 we presented a genetic algorithm, named GA-tw, for computing upper bounds on the treewidth of graphs based on a genetic algorithm for triangulations of the moral graph of Bayesian networks [36]. Computational experiments showed that the position-based crossover operator (POS) and the insertion mutation operator (ISM) are suitable operators for achieving small upper bounds on the treewidth of graphs. Compared with the best upper bounds for 62 benchmark graphs [18] known from [4], [5], [13] and [24], GA-tw found an improved upper bound on the treewidth for 22 graphs, GA-tw was able to return the same upper bound for 31 graphs, and for only 9 graphs the results delivered by GA-tw were worse.

In chapter 3 we proved that at least one elimination ordering of a hypergraph corresponds to a generalized hypertree decomposition of optimal width for that hypergraph. This result implies that the set of elimination orderings may be regarded as a search space for the generalized hypertree width of hypergraphs and that heuristic methods based on elimination orderings may find an optimal generalized hypertree decomposition.

In chapter 7 we proposed a genetic algorithm, named GA-ghw, for computing upper bounds on the generalized hypertree width of hypergraphs. When applied to 19

benchmark hypergraphs from [22], GA-ghw found an improved upper bound on the generalized hypertree width for 12 graphs, GA-ghw was able to return the same upper bound for 2 graphs, and for 5 graphs the width returned by GA-ghw was worse than the best upper bound known from [17].

We implemented a self-adaptive island genetic algorithm for generalized hypertree width upper bounds based on [19]. Although the self-adaptive genetic algorithm did not give so good results as the the previously mentioned genetic algorithm, its main advantage is that it is able to adjust its control parameters itself and doesn't require time-consuming experiments in order to obtain suitable values for those control parameters.

In chapter 8 we proved that we may obtain a lower bound on the generalized hypertree width of a hypergraph from a lower bound on its treewidth and from an exact solution or a lower bound for a k-set cover problem, arising from the treewidth lower bound. Based on that results, we proposed a lower bound heuristic for generalized hypertree width, named tw-ksc-width. Moreover we showed how known reduction [8] and pruning techniques [5], for shrinking the search space for treewidth, may also be applied for reducing the search space for the generalized hypertree width. We proposed a branch and bound algorithm, named BB-ghw, in chapter 8 and an A* algorithm, named A*-ghw, in chapter 9, which use the lower bound heuristic tw-ksc-width and some of the derived reduction and pruning techniques.

We tested algorithm BB-ghw with 95 benchmark instances in [22] and compared the results delivered by BB-ghw with known upper bounds on their generalized hypertree width from [17]. Algorithm BB-ghw was able to compute the generalized hypertree width for 23 instances. For other 21 hypergraphs BB-ghw returned improved upper bounds on the generalized hypertree width, which for 7 hypergraphs was due to an improved upper bound returned by the min-fill heuristic. Anyway, for 14 hypergraphs the improved upper bound was found within the branch and bound search. For 40 BB-ghw could return an upper bound which was equal to the best upper bound for that instance. Only for 8 instances the best known upper bound was not reached.

Algorithm A*-ghw was applied to 87 benchmark instances in [22] and it was able to compute the generalized hypertree width for 19 instances whereas for other 9 hypergraphs A*-ghw was able to improve the initial lower bound.

One interesting point for future research is the development of new lower bound heuristics for the generalized hypertree width of hypergraphs. With improved lower bounds on the generalized hypertree width of hypergraphs branch and bound algorithms would be able to discard more regions from their search space. Also the behavior of A* algorithms depends on the quality of their underlying lower bound heuristics.

Another task for future research is the development of new reduction and pruning rules which again would reduce the search space of A* and branch and bound algorithms.

Bibliography

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

[2] Fahiem Bacchus. Lecture notes of CSC2512. http://www.cs.toronto.edu/~fbacchus/csc2512/Lectures/lecture2.pdf.

[3] Fahiem Bacchus. Tree width and elimination width. http://www.cs.toronto.edu/~fbacchus/csc2512/Lectures/treeWidthEliminationWidth.pdf, 2005.

[4] Emgad H. Bachoore and Hans L. Bodlaender. New upper bound heuristics for treewidth. In *WEA*, pages 216–227, 2005.

[5] Emgad H. Bachoore and Hans L. Bodlaender. A branch and bound algorithm for exact, upper, and lower bounds on treewidth. In *AAIM*, pages 255–266, 2006.

[6] Hans L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. In *STOC*, pages 226–234, 1993.

[7] Hans L. Bodlaender. Discovering treewidth. In *SOFSEM*, pages 1–16, 2005.

[8] Hans L. Bodlaender, Arie M. C. A. Koster, and Frank van den Eijkhof. Preprocessing for triangulation of probabilistic networks. Technical report, Institute of Information and Computing Science, Utrecht University, 2003. UUCS-2003-001.

[9] Hans L. Bodlaender, Arie M. C. A. Koster, and Thomas Wolle. Contraction and treewidth lower bounds. In *ESA*, pages 628–639, 2004.

[10] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, pages 77–90, 1977.

[11] V. Chvatal. A greedy heuristic for the set covering problem. *Math. of Operations Research*, 4:233–235, 1979.

[12] F. Clautiaux, A. Moukrim, S. Nègre, and J. Carlier. Heuristic and metaheuristic methods for computing graph treewidth. *RAIRO Operations Research*, 38:13–26, 2004.

[13] François Clautiaux, Jacques Carlier, Aziz Moukrim, and Stéphane Nègre. New lower and upper bounds for graph treewidth. In *WEA*, pages 70–80, 2003.

[14] Rina Dechter. Constraint networks. In *Enceclopedia of Artificial Intelligence, 2nd edn.*, pages 276–285. Wiley, New York, 1992.

[15] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[16] Rina Dechter and Judea Pearl. Network-based heuristics for constraint-satisfaction problems. *Artif. Intell.*, 34(1):1–38, 1987.

[17] Artan Dermaku, Tobias Ganzow, Georg Gottlob, Ben McMahan, Nysret Musliu, and Marko Samer. Heursitic methods for hypertree decompositions. Technical report, Technische Universitt Wien, 2005. DBAI-TR-2005-53.

[18] Graph coloring instances. http://mat.gsia.cmu.edu/COLOR/instances.html.

[19] N. Shibata E. Takashima, Y. Murata and M. Ito. Self adaptive island GA. *Congress on Evolutionary Computation*, pages pp. 1072–1079 (2003), 2003.

[20] A. E. Eiben, Robert Hinterding, and Zbigniew Michalewicz. Parameter control in evolutionary algorithms. *IEEE Trans. Evolutionary Computation*, 3(2):124–141, 1999.

[21] Rajiv Gandhi, Samir Khuller, and Aravind Srinivasan. Approximation algorithms for partial covering problems. In *ICALP*, pages 225–236, 2001.

[22] Tobias Ganzow, Georg Gottlob, Nysret Musliu, and Marko Samer. A CSP hypergraph library. Technical report, Technische Universitt Wien, 2005. DBAI-TR-2005-50.

[23] GLPK (GNU linear programming kit). http://www.gnu.org/software/glpk/.

[24] V. Gogate and R. Dechter. A complete anytime algorithm for treewidth. In *Proceedings of the 20th Annual Conference on Uncertainty in Artificial Intelligence UAI-04*, pages 201–208, AUAI Press, Arlington, Virginia, USA, 2004.

[25] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York, 1980.

[26] Georg Gottlob, Martin Grohe, Nysret Musliu, Marko Samer, and Francesco Scarcello. Hypertree decompositions: Structure, algorithms, and applications. In *WG*, pages 1–15, 2005.

[27] Georg Gottlob, Nicola Leone, and Francesco Scarcello. A comparison of structural CSP decomposition methods. *Artif. Intell.*, 124(2):243–282, 2000.

[28] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions: A survey. In *MFCS*, pages 37–57, 2001.

[29] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. *J. Comput. Syst. Sci.*, 64(3):579–627, 2002.

[30] I. V. Hicks, A. M. C. A Koster, and E. Kolotoğlu. Branch and tree decomposition techniques for discrete optimization. *Tutorials in Operations Research, INFORMS*, 2005.

[31] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975.

[32] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations, Proc. Sympos. IBM Thomas J. Watson Res. Center, Yorktown Heights*, pages 85–103, 1972.

[33] Phokion G. Kolaitis and Moshe Y. Vardi. Conjunctive-query containment and constraint satisfaction. In *PODS*, pages 205–213, 1998.

[34] A. M. C. A. Koster, H. L. Bodlaender, and S. P. M. van Hoesel. Treewidth: Computational experiments. Technical report, Institute of Information and Computing Science, Utrecht University, 2001. UU-CS-2001-49.

[35] Arie M. C. A. Koster, Thomas Wolle, and Hans L. Bodlaender. Degree-based treewidth lower bounds. In *WEA*, pages 101–112, 2005.

[36] P. Larrañaga, C. M. H. Kuijpers, M. Poza, and R. H. Murga. Decomposing bayesian networks: triangulation of the moral graph with genetic algorithms. *Statistics and Computing (UK)*, 7(1):19–34, 1997.

[37] Ben McMahan. Bucket elimination and hypertree decompositions. Implementation Report, Institute of Information Systems (DBAI), TU Vienna, 2004.

[38] Zbigniew Michalewicz and David B. Fogel. *How to Solve It: Modern Heuristics*. Springer, 2004.

[39] Ugo Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Inf. Sci.*, 7:95–132, 1974.

[40] J. Pearson and P. Jeavons. A survey of tractable constraint satisfaction problems. Technical report, Royal Holloway, University of London, 1997. CSD-TR-97-15.

[41] Siddharthan Ramachandramurthi. The structure and number of obstructions to treewidth. *SIAM J. Discrete Math.*, 10(1):146–157, 1997.

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

[43] D. J. Rose. *A Graph Theoretic Study of the Numerical Solution of Sparse Positive Definite Systems of Linear Equations (R. C. Reed ed.)*, pages 184–218. Academic Press, 1972.

[44] Donald J. Rose and Robert Endre Tarjan. Algorithmic aspects of vertex elimination. In *STOC*, pages 245–254, 1975.

[45] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach (Second Edition)*. Prentice Hall Series in Artificial Intelligence, 2002.

[46] Alexander Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1996.

[47] Thomas Schwentick, Georg Gottlob, and Zoltan Miklos. private communications, 2005.

[48] Reiko Tanese. Distributed genetic algorithms. In *ICGA*, pages 434–439, 1989.