

Decomposing satisfiability problems  
or  
Using graphs to get a better insight into satisfiability problems

P.R. Herwig

October 5, 2006



Decomposing satisfiability problems  
or  
Using graphs to get a better insight into satisfiability problems

**Master's Thesis**

Department of Electrical Engineering, Mathematics and  
Computer Science  
Delft University of Technology

**P.R. Herwig**  
October 2006

Graduation Committee:  
prof.dr.ir. H.J. Sips (chair)  
ir. M.J.H. Heule  
dr. H. van Maaren  
prof.dr. C. Witteveen



# Preface

During the process of writing this thesis I have had both times of joy and depression. I was introduced to the field of *satisfiability* by following the master course `Computational Logic and Satisfiability`. Satisfiability finds itself at the border of mathematics and computer science and turned out to be a very interesting topic to finish one's studies. I had especially great pleasure in doing research: Thinking of new ideas, implementing them and testing the effects. Actually writing this thesis caused somewhat more trouble. I can say I have come to better knowledge of myself.

I would like to thank dr. Hans van Maaren and Marijn Heule for guiding and directing me during the phase of research and for advice and encouragement during the writing of this thesis. Especially thanks to Marijn whom I could reflect my ideas to and who always supported me with new ones.

Finally I want to thank my girlfriend and meanwhile wife Willeke for her personal support and listening ear during the entire process of writing this thesis.



# Contents

<b>Preface</b>	<b>iv</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Graph representations</b>	<b>6</b>
2.1 Decomposable graph representations . . . . .	7
2.2 Decomposed SAT problem . . . . .	9
2.3 Decomposing possibilities . . . . .	11
<b>3 Redundancy</b>	<b>13</b>
3.1 Subsumption . . . . .	13
3.2 Blocked . . . . .	14
3.3 Blocked clauses and the effect on solving . . . . .	15
<b>4 Connectivity</b>	<b>18</b>
4.1 Connected components . . . . .	18
4.2 Articulation points . . . . .	19
4.3 Tree decomposition . . . . .	20
4.4 Diameter . . . . .	23
<b>5 Visualisation</b>	<b>27</b>
5.1 Case study: Tower of Hanoi . . . . .	29
5.2 Case study: Connamacher . . . . .	35
5.3 Conclusions . . . . .	39
<b>6 Applying splitting</b>	<b>40</b>
6.1 Balanced minimal cut . . . . .	41
6.2 Break-set variables . . . . .	44
6.3 Case study: Tower of Hanoi . . . . .	46
6.4 Splitting in practice . . . . .	48
<b>7 Conclusions &amp; Future work</b>	<b>57</b>

*CONTENTS*

vii

**A Test-set**

**63**

**B Additional visualisations**

**67**

**C Proofs**

**82**

# List of Figures

2.1	A geometric representation of $\mathbf{vcg}(S)$ . . . . .	8
2.2	A geometric representation of $\mathbf{vig}(S)$ . . . . .	8
2.3	A geometric representation of $\mathbf{cvg}(S)$ . . . . .	8
2.4	A geometric representation of $\mathbf{cg}(S)$ . . . . .	9
2.5	A geometric representation of $\mathbf{rg}(S)$ . . . . .	9
2.6	A geometric representation of $\mathbf{srg}(S)$ . . . . .	9
3.1	A geometric representation of $\mathbf{rg}(S)$ with blocked clauses in red.	15
3.2	A geometric representation of $\mathbf{rg}(S)$ after one iteration of removing blocked clauses. Both remaining clauses are blocked clauses now. . . . .	15
4.1	(1) A geometric representation of a graph and (2) one of its tree decompositions. The tree decomposition has a width of 2.	21
4.2	Test-set: Plot of the diameter and density by their fastest solver.	26
5.1	A geometric representation of a tree decomposition of the resolution graph of <b>dubois50</b> . . . . .	29
5.2	A geometric representation of a 1-reduced tree decomposition of the resolution graph of <b>dubois50</b> . . . . .	29
5.3	A geometric representation of a 1-reduced tree decomposition of the resolution graph of the preprocessed <b>hanoi4</b> . . . . .	31
5.4	Nine small geometric representation of a 1-reduced tree decomposition of $\mathbf{rg}(\mathbf{hanoi4})$ , having red bags for the variables mentioned as predicates in table 5.1. . . . .	32
5.5	Ten small geometric representations (all of the same scale) of 1-reduced tree decompositions of $\mathbf{rg}(\mathbf{hanoi4})$ with previous variables already set, having red bags for the variables mentioned as predicates in table 5.1. . . . .	34

5.6	A geometric representation of a 1–reduced tree decomposition of the resolution graph of <code>connamacher 975</code> with one bag marked green. . . . .	36
5.7	A geometric representation of the $\mathbf{rg}$ of the green bag from the 1–reduced tree decomposition of <code>connamacher 975</code> from figure 5.6 . . . . .	36
5.8	A geometric representation of the simplified graph of figure 5.7	37
5.9	A geometric representation of a 1–reduced tree decomposition of the resolution graph of <code>connamacher 975</code> with multiple bags marked green. . . . .	37
5.10	A geometric representation of the simplified graph of the subgraph of $\mathbf{rg}(\text{connamacher 975})$ formed by the green bags in figure 5.9 . . . . .	38
6.1	A geometric representation of $\mathbf{rg}(R)$ . . . . .	43
6.2	A graphical representation of the branching method for a variable break–set and bipartition. . . . .	46
6.3	A geometric representation of a 1–reduced tree decomposition of the resolution graph of the preprocessed <code>hanoi4</code> , having red bags for the break–set variables. . . . .	47
B.1	A geometric representation of a 1–reduced tree decomposition of $\mathbf{rg}(1000144.\text{col})$ . . . . .	68
B.2	A geometric representation of a 1–reduced tree decomposition of $\mathbf{rg}(\text{connamacher 975})$ . . . . .	69
B.3	A geometric representation of a 1–reduced tree decomposition of $\mathbf{rg}(\text{depots 3948})$ . . . . .	70
B.4	A geometric representation of a 1–reduced tree decomposition of $\mathbf{rg}(\text{driverlog 3963})$ . . . . .	71
B.5	A geometric representation of a 1–reduced tree decomposition of $\mathbf{rg}(\text{eulcbip 3936})$ . . . . .	72
B.6	A geometric representation of a 1–reduced tree decomposition of $\mathbf{rg}(\text{ferry 3996})$ . . . . .	72
B.7	A geometric representation of a 1–reduced tree decomposition of $\mathbf{rg}(\text{hanoi4})$ . . . . .	73
B.8	A geometric representation of a 1–reduced tree decomposition of $\mathbf{rg}(\text{stanion 1617})$ . . . . .	74
B.9	A geometric representation of a 1–reduced tree decomposition of $\mathbf{rg}(\text{linvrinv 565})$ . . . . .	75
B.10	A geometric representation of a 1–reduced tree decomposition of $\mathbf{rg}(\text{longmult8})$ . . . . .	76

B.11 A geometric representation of a 1-reduced tree decomposition of <b>rg(philips)</b> . . . . .	77
B.12 A geometric representation of a 1-reduced tree decomposition of <b>rg(pmg 3939)</b> . . . . .	78
B.13 A geometric representation of a 1-reduced tree decomposition of <b>rg(rovers 3978)</b> . . . . .	79
B.14 A geometric representation of a 1-reduced tree decomposition of <b>rg(satellite 3985)</b> . . . . .	80
B.15 A geometric representation of a 1-reduced tree decomposition of <b>rg(unsat350 20)</b> . . . . .	81
B.16 A geometric representation of a 1-reduced tree decomposition of <b>rg(vdw_5_3_50)</b> . . . . .	81



# List of Tables

2.1	Test set: Edges of the <b>vig</b> , the <b>cvg</b> , the <b>cg</b> , the <b>rg</b> and the <b>srg</b> .	10
3.1	Test set: Duplicate clauses, subsumed clauses and blocked clauses.	16
3.2	Test set: Solve times in seconds using <b>march_dl</b> for different CNF preprocessings: The original formula, after removing all blocked clauses, after removing all blocked clauses and adding all possible directly conflicting blocked binary clauses.	17
4.1	Test set: Longest shortest path over all variables, average longest shortest path per variable (of the <b>vig</b> ) and solve times in seconds using look-ahead solver <b>march_dl</b> and conflict driven solver <b>MiniSat</b> .	24
5.1	The first nine steps in solving the 4 discs Hanoi puzzle along with their predicates for the CNF translation.	32
6.1	Test set: Approximated tree width of <b>vig</b> using <b>mdfi</b> , separating vertex set of <b>vig</b> using the diameter (ds-set) and balanced approximated minimal cut of <b>chg</b> using <b>hMeTiS</b> along with its resulted balance.	42
6.2	Test set benchmarks having multiple hyperedges for variables in the <b>rhg</b> : Edges of the <b>chg</b> and the <b>rhg</b> .	44
6.3	Test set: Solve times in seconds using <b>march_dl</b> and different modifications implementing partition branching.	50
6.4	Test set: Sizes of the variable break-set ( $V_{bs}$ ), Left and Right partition, the number of partition swaps and whether a satisfying assignment could be found for the variable break set or not.	51
6.5	<b>connamacher 975</b> : Solve times in seconds using the <b>FL500</b> partition switch modification of <b>march_dl</b> for five runs of eight different balanced approximated minimal cuts using <b>hMeTiS</b> .	52

6.6	1000144.col: Solve times in seconds using the FLAps 500 modification of <code>march_dl</code> for two runs of nine different balanced approximated minimal cuts using <code>hMeTiS</code> . . . . .	54
6.7	stanion 1617: Solve times in seconds using the FLAps 500 modification of <code>march_dl</code> for five different balanced approximated minimal cuts using <code>hMeTiS</code> . . . . .	55
6.8	pmg 3939: Solve times in seconds using the FLAps 500 modification of <code>march_dl</code> for five different balanced approximated minimal cuts using <code>hMeTiS</code> . . . . .	55
6.9	philips: Solve times in seconds using the FLAps 500 modification of <code>march_dl</code> for five different balanced approximated minimal cuts using <code>hMeTiS</code> . . . . .	56
6.10	anton 930: Solve times in seconds using the FLAps 500 modification of <code>march_dl</code> for nine different balanced approximated minimal cuts using <code>hMeTiS</code> . . . . .	56
A.1	The test set . . . . .	64
C.1	Simplified incidence matrix of two disjoint subhypergraphs $H_1(V_1, E_1)$ and $H_2(V_2, E_2)$ after removing a separating hyper-edge set $E_c$ . . . . .	82
C.2	Simplified transpose of incidence matrix of two disjoint hypergraphs $H_1(V_1, E_1)$ and $H_2(V_2, E_2)$ . . . . .	83

# Chapter 1

## Introduction

Say one needs to calculate the product of 23 and 76 without the use of a calculator. It is quite likely one splits up a number in order to calculate the result more easily. First, one calculates the product of 20 and 76, and then one adds the result of 3 times 76. Of course not only 23 can be splitted by the digits, the same can be done for 76. The idea of splitting up a problem in easier subproblems to obtain a solution for the entire problem is also known as the *divide and conquer* principle.

Already in the time of Julius Ceasar *divide and conquer* was a successful military strategy. Generals observed that it was easier to defeat one army of 50,000 men, followed by another army of 50,000 men than it was to beat a single 100,000 man army. Thus the wise general would attack so as to divide the enemy army into two forces and then mop up one after the other. Caesar was the ultimate practitioner of the divide and conquer school.

Nowadays divide and conquer is an important design principle in computer science. Recursively break down a problem into two or more subproblems, until these subproblems are simple enough to be solved directly. Perhaps one of the most classical examples of a divide and conquer algorithm is *mergesort*. Mergesort is a sorting algorithm that rearranges an unordered list into a specified order. One can sort the list by simply comparing each element with all the others. However, mergesort is a much more efficient algorithm to sort a list. The algorithm was invented by John von Neumann in 1945 and works basicly as follows:

1. First divide the unsorted list into two sublists of about equal size.
2. Sort each of the two sublists.

3. Merge the two sorted sublists into one sorted list.

This can be repeated recursively, each unsorted sublist can be sorted the same way as the entire list.

Decomposing the problem into two or more subproblems of which the solutions can be found more easily and of which the solutions can also be easily merged to find a solution for the original problem is exactly what we want to achieve for the *satisfiability* problem.

Another ancient military slogan characterizing a key role of this thesis is: *Know your enemy*. Particularly for the aspect of decomposing the *satisfiability* problem we are interested in its structure. What properties can be used to capture the connectivity of the problem? Perhaps the best way to get a better insight into a *satisfiability* problem is to visualize it. For a picture can say more than a thousand words.

The satisfiability problem, or SAT *problem* in short, deals with the *satisfiability* of a propositional Boolean formula, that is: Is there a truth assignment to the Boolean variables of the formula that satisfies the formula, or not?

The standard encoding of a propositional Boolean formula in the field of SAT is in *Conjunctive Normal Form*, CNF. A CNF formula  $F$  is a conjunction of clauses  $c_1, c_2, \dots, c_m$ , with each clause being a disjunction of literals  $l_1, l_2, \dots, l_n$ , and each literal an atomic Boolean variable  $x_i$  or its negated form  $\neg x_i$ .

Assigning a Boolean variable  $x_i$  to **true** satisfies all occurrences of the literal  $x_i$ . Assigning a Boolean variable  $x_i$  to **false** satisfies all occurrences of the negated literal  $\neg x_i$ . A clause is satisfied if at least one of its literals is satisfied. Finally the CNF formula is satisfied if all its clauses are satisfied.

A lot of problems from real-world applications can be naturally formulated as a SAT problem. From bounded model checking to the combinatorics of the Van der Waerden problem [19] and from planning and scheduling to soft- and hardware verification. So, a solver capable of solving SAT problems can be used to solve a wide range of problems from real-world applications.

It is that what makes the development of fast SAT solver programs very interesting. Several SAT solving competitions have been held in the past years. Also interesting is the fact that current solving techniques seem to

defy the bad scaling character for a lot of these practical problems. Current state-of-the-art solvers can handle problem instances, as they arise in finite model-checking and planning, with up to a million variables and five million clauses [32]. Apparently practical SAT problem instances have a substantial amount of (hidden) tractable sub-structure and current techniques are able to exploit such tractable structures [38].

The most commonly used framework to solve SAT problems is the Davis-Putman-Logemann-Loveland procedure (DPLL). This is a recursive and depth-first search-tree procedure for deciding the satisfiability of a SAT problem. The basic backtracking algorithm works by selecting a literal, assigning a truth value to it, simplifying the formula and recursively checking if the simplified formula is satisfiable. The literal selected is called the *branching literal*. After assigning a truth value to a literal the formula can be simplified by removing all satisfied clauses. All complement occurrences of the literal can be removed from the clauses, so they are shortened. If this results in an empty clause it means no other literal can be set to satisfy that clause and we need to backtrack one step. The formula is restored to the previous step and the same recursive check is done assuming the opposite truth value.

At each node of the DPLL procedure *unit propagation* is applied. A clause containing only one literal is called a *unit clause*. The clause can only be satisfied by setting this literal to `true`. Assigning a truth value to such a literal may result in a new unit clause. Unit propagation iteratively fixes all unit clauses on their truth value and simplifies the formula. Also *pure literal elimination* is applied at each node of the DPLL procedure. If a variable occurs only with a single polarity in the entire formula this is called a *pure literal*. A pure literal can simply be set to `true` for all clauses it occurs in, leaving these clauses satisfied. Although pure literal elimination is part of the original DPLL procedure it is omitted in most current implementations.

Satisfiability is detected when all clauses are satisfied (an empty formula), unsatisfiability of the complete formula can only be detected after exhaustive search. A wide variety of implementations exist to select the next literal in each step of the DPLL procedure.

Although the definition of the SAT problem is quite easy, the problem itself is not. The SAT problem was the first to be proved NP-complete [10]. NP-complete problems are the most difficult within the NP class, being problems with a nondeterministic polynomial time complexity. Many real-world problems also have a NP-complete time complexity. Every NP-complete

problem can be transformed into another NP-complete problem in polynomial time.

Williams, Gomes and Selman identify in [38] two main powerful intuitions in the design of search methods. One of them is the understanding that different groups of variables in a problem encoding often play quite distinct roles. For example, one can distinguish between dependent and independent variables in a problem. The true combinatorics arises from the independent variables. The dependent variables are just needed to obtain a compact problem encoding. Branching purely on the independent variables result in a substantial speedup compared to methods that do not exploit these dependencies [17]. The other intuition is that one wants to select the variables that *simplify the problem instance as much as possible* when they are assigned truth values. For example, branching techniques will select the variable that when assigned will result in the highest number of unit and binary clauses.

There exist quite some different variable types. We already mentioned the distinguishing between independent and dependent variables. Monasson *et al.* introduced in [31] the concept of *backbone* variables. A variable is called a backbone variable, or frozen variable, if in all solutions of the SAT problem the variable is assigned the same value. Another type of variables are *backdoor* variables. Backdoors are variable subsets defined with respect to a particular algorithm; once the backdoor variables are assigned a value, the problem becomes easy (solvable in polynomial time) under that algorithm. Independent variables are a particular kind of backdoor variables. More about backdoor variables can be found in [38].

This thesis introduces the concept of a variable *break-set*. A variable break-set forms a subset that, once assigned truth values, partitions the SAT problem. The idea of setting some variables that partition the original SAT problem is not new [21, 11]. A partition of a SAT problem is a partition of the variable sets of clauses. It is desirable to partition a SAT problem since it can reduce the solution space significantly. A partitioning  $F_1, \dots, F_m$  of  $F$  for  $i \in \{1, \dots, m\}$ ,  $F$  is unsatisfiable iff one of the partitions  $F_i$  is unsatisfiable and satisfiable if all partitions  $F_i$  are satisfiable. The solution space for the  $n$  Boolean variables goes from  $2^n$  to  $2^{n_1} + \dots + 2^{n_m}$  with  $n_i$  being the number of variables of partition  $F_i$ .

As a tool in the search for partitioning possibilities we use a graph representation of the CNF problem. In order to partition a CNF problem we

are interested in the *connectivity* of the problem. How are the different variables related to each other? Does a variable only occur in a specific part of the problem? Perhaps a variable acts as a sort of spinal core throughout the entire problem, once satisfied the problem falls apart in many very small subproblems? A graph is probably the best way to capture these connectness relations. Besides using this information for partitioning possibilities we are also interested in the graph representation to get a better insight into a SAT problem. Using graph techniques to partition or to guide variable selection is not new. Durairaj and Kalla [11] use hypergraph partitioning methods to decompose a SAT problem. Bjesse *et al.* [6] use tree decompositions to guide the SAT diagnosis. Heule and Kullmann [21] present a general framework on graph splitting techniques for SAT problems.

The contribution of this thesis is twofold. On one side we use graph representations as an object to provide a better insight into satisfiability problems. On the other side we use graph representations as a means to decompose satisfiability problems.

Chapter 2 describes how a SAT problem can be represented as graph so that it might be useful for decomposition. Such graph representations should capture the connectivity of the SAT problem.

Chapter 3 discusses the use of preprocessing the SAT problem by removal of redundant information from the CNF. The possible benefit for graph representation and solving of a SAT problem, as well as the possible drawback in solving a SAT problem are described.

The connectivity of the graph representation and how to capture this is handled in chapter 4.

Chapter 5 is about visualizing SAT problems. So called *tree decompositions* of graph representations of the CNF are used to get surveyable geometric representations.

Splitting and the appliance of splitting is discussed in chapter 6. How to efficiently obtain a relative good variable break-set and how to use this set in solving a decomposed SAT problem?

In chapter 7 we bring this thesis to a close with conclusions and directions for future work.

Throughout the different chapters of this thesis a test-set of SAT problems is used to observe the properties discussed and to test different solution techniques. The test-set together with some short information on and sources of the SAT problems is given in Appendix A.

# Chapter 2

## Graph representations

A graph representation  $G(F)$  is often just a rough representation of the original SAT problem  $F$ . But losing information does not always have to be a weakness. We are interested in capturing the connectivity of the original SAT problem. A decomposed representation graph should hold that the SAT problem it represents is also decomposed. Each part of the SAT problem can be solved on its own. Before introducing the notion of *decomposing graph representations* first some basics on connectness of the graph representation are given. The rest of this chapter discusses some of the different representation graphs, their relation with decomposition in the field of SAT and their decomposing possibilities.

Two vertices  $u$  and  $v$  of a graph  $G$  are called *connected* if there is a sequence of vertices starting with  $u$  and ending with  $v$ , such that from each vertex there is an (hyper)edge to its successor. A graph  $G$  is called *connected* if every pair of vertices of  $G$  is connected. A maximal connected subgraph of  $G$  is called a *connected component*. Now, when a graph consists of more than one connected component the graph is *disconnected*.

The idea of decomposing graph representations is that for a representation graph  $G$  consisting of  $m$  multiple connected components  $G_1, \dots, G_m$ , of a SAT formula  $F$ , all connected components can be naturally associated with (non-overlapping) partitions of that formula,  $F(G_i)$  for  $i \in \{1, \dots, m\}$ , that can be solved on their own. The main requirement for decomposing a graph representation of a SAT problem is that the connected components can be naturally associated with (non-overlapping) partitions of the CNF. If  $G_1, \dots, G_m$  are the connected components of  $G(F)$  we have a partition of  $F_1, \dots, F_m$  of the CNF where  $G(F_i) = G_i$  for  $i \in \{1, \dots, m\}$ . The partitions of the CNF form the decomposition of the problem: If one of the partitions

$F_i$  is unsatisfiable, the entire SAT problem is unsatisfiable, if assignments could be found for all partitions  $F_i$ , the entire SAT problem is satisfiable as well.

## 2.1 Decomposable graph representations

There are quite some possibilities to represent a SAT problem by a decomposable graph. Perhaps the most natural representation of a SAT problem  $F$  is the *variable hypergraph*  $\mathbf{vhg}(F)$  of  $F$ . In the  $\mathbf{vhg}(F) = (V, E)$  the vertex set  $V$  is the set of variables and the hyperedge set  $E$  is formed by the variable-sets of clauses. The dual of the  $\mathbf{vhg}(F)$  is the *clause hypergraph*  $\mathbf{chg}(F)$  in which the vertex set  $V$  corresponds to the set of clauses and the hyperedge set  $E$  is the set of variables, connecting the clauses having the same variable.

Another representation is the *bipartite variable clause graph*  $\mathbf{vcg}(F)$ . The vertex set  $V$  is formed by both sets of variables and clauses. A variable vertex  $x$  is connected to a clause vertex  $C$  iff  $x \in C$ .

Standard graphs that can be obtained from the hypergraphs:

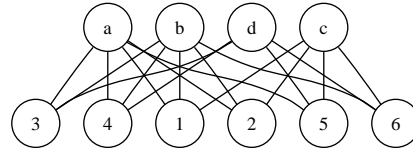
1. The *variable interaction graph*  $\mathbf{vig}(F)$ , obtained from the  $\mathbf{vhg}(F)$ , has the variables as the vertex set. Two variables are connected by an edge if they both occur in a single variable-set of a clause.
2. The *common variable graph*  $\mathbf{cvg}(F)$ , obtained from the  $\mathbf{chg}(F)$ , has the clauses as the vertex set. Two clauses are connected by an edge if they have a variable in common.

To illustrate the different graph representations we use the sample CNF:

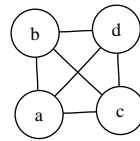
$$S = \{\{a, b, c\}, \{\neg a, \neg b, \neg c\}, \{a, b, d\}, \{a, b, \neg d\}, \{a, \neg c, d\}, \{b, \neg c, d\}\}$$

A geometric representation of  $\mathbf{vcg}(S)$  is shown in figure 2.1. Both clauses and variables are depicted by vertices. The clause vertices are numbered. An edge between a clause and variable vertex means the variable is part of the clause. A geometric representation of  $\mathbf{vig}(S)$  is given in figure 2.2. Only the variables are depicted by vertices. An edge between two variables indicates they both occur in a single clause. In a geometric representation of  $\mathbf{cvg}(S)$ , see figure 2.3, the clauses are the vertices and they are connected if they share a variable.

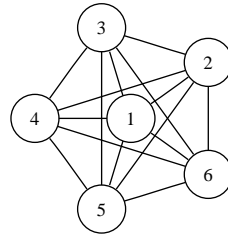
Three refinements of the  $\mathbf{cvg}(F)$ , all having clauses as the vertex set, are:



**Figure 2.1:** A geometric representation of  $\mathbf{vcg}(S)$



**Figure 2.2:** A geometric representation of  $\mathbf{vig}(S)$

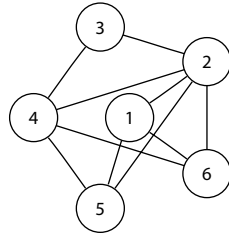


**Figure 2.3:** A geometric representation of  $\mathbf{cvg}(S)$

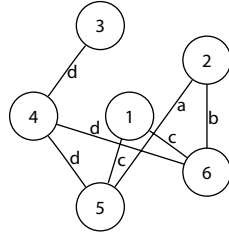
- The *conflict graph*  $\mathbf{cg}(F)$ , having two clauses connected by an edge if they have at least one clashing literal.
- The *resolution graph*  $\mathbf{rg}(F)$ , having two clauses connected by an edge if they have exactly one clashing literal.
- The *subsumption-resolution graph*  $\mathbf{srg}(F)$ , having two clauses connected by an edge if they have exactly one clashing literal and the resolvent of the clauses is not subsumed by a clause in  $F$ .

Geometric representations of these graphs for the example CNF  $S$  are given by figure 2.4, 2.5 and 2.6. Notice that a single edge in the resolution and subsumption-resolution graph represents exactly one variable (the single clashing literal between the two clauses). These variables are denoted along the edges.

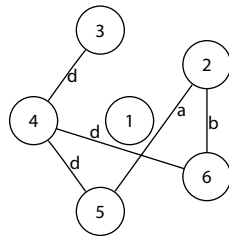
Like the refinements of the  $\mathbf{cvg}(F)$ , one can also define refinements of the  $\mathbf{chg}$ . More about these specific refinements can be found in chapter 6.



**Figure 2.4:** A geometric representation of  $\mathbf{cg}(S)$



**Figure 2.5:** A geometric representation of  $\mathbf{rg}(S)$



**Figure 2.6:** A geometric representation of  $\mathbf{srg}(S)$

By definition  $E(\mathbf{cvg}(F)) \supseteq E(\mathbf{cg}(F)) \supseteq E(\mathbf{rg}(F)) \supseteq E(\mathbf{srg}(F))$ . The number of edges of the  $\mathbf{vig}$ , the  $\mathbf{cvg}$ , the  $\mathbf{cg}$ , the  $\mathbf{rg}$  and the  $\mathbf{srg}$  of the test set are given in table 2.1.

## 2.2 Decomposed Sat problem

We should be sure that the graph representations are decomposable graph representations. We discuss the graph representations mentioned in the previous section.

A disconnected  $\mathbf{vhg}(F)$  means that the different connected components share no hyperedge. So groups of variables have no clause in common. It is trivial to see that the underlying SAT problem then is partitioned as well.

**Table 2.1:** Test set: Edges of the **vig**, the **cvg**, the **cg**, the **rg** and the **srg**.

benchmark	vars	cls	$E(\mathbf{vig})$	$E(\mathbf{cvg})$	$E(\mathbf{cg})$	$E(\mathbf{rg})$	$E(\mathbf{srg})$
1000144.col	1200	3154	6699	18120	5508	5508	5508
pyhala-braun 03	7383	24320	34362	351553	186808	162493	158022
connamacher 975	568	7150	14612	427104	21024	21024	21024
depots 3948	290	3587	7829	129830	28549	26345	21731
driverlog 3963	170	1595	3325	40288	7955	7614	7594
eulcbip 3936	135	672	804	16415	9715	7236	7236
ferry 3996	701	6661	13697	205983	15643	14646	14646
hanoi4	718	4934	6949	59060	23437	21238	20621
stanion 1617	162	774	1213	19462	10948	8836	8368
ibm SAT_dat.k15	13818	69795	130498	2125297	1094509	1025134	865566
linvrinv 565	792	2557	5089	28908	15624	12888	12888
longmult8	3810	11877	16749	312152	159857	152740	146566
philips	3642	4456	7175	47996	25827	21444	20954
pmg 3939	169	562	672	5320	3137	2353	2353
qg3-09	729	28463	12274	825017	326634	312646	159784
rovers 3978	229	3697	10304	242123	102723	82720	82720
anton 930	1000	3730	11641	65501	33970	32887	32794
satellite 3985	158	3086	7152	167174	27666	25948	25948
unsat350 20	350	1493	6425	28325	14167	14115	14114
vdw_5_3_50	250	2953	10258	159872	8688	8688	8660

Connected components of the  $\mathbf{vhg}(F)$  represent partitioned groups of clauses sharing no variables. All partitions  $\mathbf{vhg}(F_i)$  can be solved on their own. Since the groups of variables are partitioned as well, the solution of the entire SAT problem can be found by simply combining the variable settings of all partitions. The same holds for its dual graph, the  $\mathbf{chg}(F)$  and both standard graphs that can be obtained from these hypergraphs, the  $\mathbf{vig}(F)$  and  $\mathbf{cvg}(F)$ .

In the bipartite  $\mathbf{vcg}(F)$  both variables and clauses are represented as vertices. A disconnected  $\mathbf{vcg}(F)$  again means both variables and clauses are partitioned by the different connected components. Therefore, the partitions can be solved on their own and the solutions can be simply combined to find a solution of the entire SAT problem.

The refinements of the  $\mathbf{cvg}(F) - \mathbf{cg}(F)$ ,  $\mathbf{rg}(F)$  and  $\mathbf{srg}(F)$  – are partitioned for the clauses for the disconnected components since they are clause-based. One can solve the separated partitions on their own, however, efficiently finding a solution for the entire SAT problem may not be trivial.

First of all it is possible for all three refinements that edges representing a single variable do not occur in the graph at all. These are pure literals. Without the clashing literals no edges can be formed in the **cg** and its refinements. A pure literal can simply be set to **true** for all clauses it occurs in, leaving these clauses satisfied. Somewhat more complex is the occurrence of a single variable in multiple connected components. This can occur in both the **rg**( $F$ ) and the **srg**( $F$ ). Kullmann proves in his master thesis [29] that there is a solution for the entire SAT problem if there is a solution for the partial problems.

## 2.3 Decomposing possibilities

If the graph representation consists of multiple connected components, the components directly partition the graph. Unfortunately, the graph representation of practical SAT instances rarely consists of (interesting) multiple connected components. We will have to disconnect the representation graph ourselves.

A graph can be disconnected by removing a set of vertices, a so called *separating vertex set* or by removing a set of (hyper)edges, a so called *separating (hyper)edge set*. Aiming for a *well-balanced partition* is often a trade off with the size of the separating set. The theoretical gain is at its maximum when the largest part is as small as possible. A better balanced partition often has a larger separating set. Since the vertices and (hyper)edges in our graph representations correspond to either variables or clauses, the problem is equivalent to removing sets of variables or sets of clauses. One can remove a variable from  $F$  by assigning it a truth value. A clause can be removed from  $F$  by any satisfying assignment of its set of variables.

A separating vertex set in the **vhg**( $F$ ) means splitting on variables, while for its dual, the **chg**( $F$ ), it means splitting on clauses. A separating hyper-edge set is exactly the opposite for hypergraphs: For the **vhg** this means splitting on clauses and for the **chg** splitting on variables. Using a mixed separating set of vertices and edges in general seems to be a rather artificial but possible third method.

Separating vertex sets in the **vig** and **cvg** stand respectively for variable and clause splitting. Separating edge sets is not preferable for these graph since the edges do not directly correspond with either clauses or variables. For the **vig** each edge might represent multiple clauses, while for the **cvg**

each edge might represent multiple variables. For the **rg** and **srg** however, a single edge represents exactly one variable.

For splitting on clauses, the refinements of the **cvg**( $F$ ) have definitely better possibilities than the **cvg**( $F$ ) itself. Having less edges increases the likelihood of a smaller separating vertex set. Also for splitting on variables, separating edge sets in the **rg**( $F$ ) and **srg**( $F$ ) are more interesting than separating vertex sets in the **vig**( $F$ ). The separating edge cut covers all splitting possibilities given by separating vertex sets in the **vig**( $F$ ) and yet embody more structural information about  $F$  [21]. Although the **rg**( $F$ ) and **srg**( $F$ ) are larger than the **vig**( $F$ ), more *decomposition intelligence* is provided since vertices (clauses) in these graphs are only joined if they have exactly one clashing literal (**rg**) or exactly one clashing literal and the resolvent of the clauses is not subsumed by a clause in  $F$  (**srg**).

# Chapter 3

## Redundancy

In general, less edges or vertices increases the chance of the graph falling apart. Since we prefer having disconnected graph representations we would like to remove redundant variables and/or clauses from the CNF problem for this purpose. A transformation of a problem to the field of SAT does not often lead to the most compact representation in CNF. The CNF can be preprocessed to reduce the instance size significantly. For redundancy removing we are focussing only on the removal of so called *redundant clauses*. For some different preprocess techniques the reader is referred to [12]. A *redundant clause* is a clause which can be removed from a SAT problem  $F$  without affecting its satisfiability. A SAT problem can be preprocessed by removing (some of the) redundant clauses. Removing redundant clauses may also reduce the number of variables.

The effect of removing redundant clauses is that the resulting representation graph will either have less or equal edges (**vig**) or less vertices (**chg** and its refinements). The graph representation of  $F_r$ ,  $F$  without (some) redundant clauses, is always a subgraph of the graph representation of  $F$ .

Different groups of redundant clauses can be distinguished. Two forms of redundant clauses are discussed in the following sections and the last section is on the effect of removing so called *blocked* clauses on the solve time.

### 3.1 Subsumption

A trivial example of a redundant clause is a *duplicate clause*.

**Definition 3.1.1** *A clause  $C_d$  is a duplicate clause if there exists a clause  $C_i$  having exactly the same set of literals.*

Duplicate clauses can be seen as a subclass of the set of *subsumed clauses*. Let  $l(C)$  denote the set of literals of clause  $C$ .

**Definition 3.1.2** *A clause  $C_s$  is a subsumed clause if there exists a clause  $C_i$  such that  $l(C_i) \subseteq l(C_s)$ .*

A subsumed clause  $C_s$  is a duplicate clause if there exists a clause  $C_i$  such that  $l(C_i) = l(C_s)$ . Removal of subsumed clauses is always useful. These clauses only consume extra memory which slows down the reasoning process of the solver. More information about subsumed clauses can be found in [41].

## 3.2 Blocked

Less trivial is the redundancy of so called *blocked clauses*.

**Definition 3.2.1** *A clause  $C_b$  is a blocked clause if it contains a literal  $l$ , such that every clause that contains the clashing literal  $\neg l$  also contains another clashing literal of  $C_b$ .*

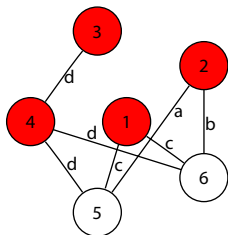
Notice that a clause having a pure literal is also a blocked clause. Since the set of clauses containing the clashing literal is the same as the set of clauses containing the clashing literal as well as another clashing literal (both empty sets). More information about blocked clauses can be found in [30].

An example: The blocked clauses of the sample CNF:

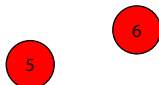
$$S = \{\{a, b, c\}, \{\neg a, \neg b, \neg c\}, \{a, b, d\}, \{a, b, \neg d\}, \{a, \neg c, d\}, \{b, \neg c, d\}\}$$

are represented as red nodes in a geometric representation of  $\mathbf{rg}(S)$  in figure 3.1. After removing the blocked clauses, other clauses can become blocked. So removing blocked clauses can be performed iterative. After removing the blocked clauses in the sample CNF, only two clauses remain, see figure 3.2. Now these clauses become blocked. After iteratively removal of blocked clauses the remaining formula is left empty: We can conclude that  $S$  is satisfiable. Notice that although we know it is satisfiable, we do not know a solution.

Removal of redundant clauses reduces the size of the CNF and therefore also the amount of memory needed by the solver. For the subsumed clauses removal is always profitable. However, this does not hold for blocked clauses. These clauses might be useful in the reasoning process of the solver and removal will slow down the solver. On the other hand, adding blocked clauses



**Figure 3.1:** A geometric representation of  $\mathbf{rg}(S)$  with blocked clauses in red.



**Figure 3.2:** A geometric representation of  $\mathbf{rg}(S)$  after one iteration of removing blocked clauses. Both remaining clauses are blocked clauses now.

could fasten the solving process. More about the consequences on the solve time of removed and added blocked clauses can be found in the next section.

Table 3.1 shows the number of duplicate clauses, subsumed clauses (without duplicate clauses) and blocked clauses in the test-set of SAT instances.

### 3.3 Blocked clauses and the effect on solving

The effect of removing blocked clauses is somewhat more complex. Depending on both SAT solver and CNF benchmark removal can be profitable. During experiments it also appeared that, again depending on both SAT solver and CNF benchmark, adding some specific blocked clauses can also be profitable. We define two sets of blocked clauses.  $B_1$  is the set of all blocked clauses that can be removed and  $B_2$  is the set of all possible directly conflicting blocked binary clauses that can be added. A directly conflicting blocked binary clause is a clause of length two being blocked for both its literals. The idea of adding these clauses is that they increase unit propagation. Table 3.2 shows the solve times for the test-set of SAT instances using the SAT solver `march_dl` [22]. The solve time is given for the original formula  $F$ , for  $F - B_1$  and for  $(F - B_1) \cup B_2$ . If a benchmark had no blocked clauses or no directly conflicting blocked binary clauses could be added the solving time is the same as for the original benchmark and a dash is shown in the table.

**Table 3.1:** Test set: Duplicate clauses, subsumed clauses and blocked clauses.

benchmark	vars	cls	dupl cls	subs cls	# iter	bl cls
1000144.col	1200	3154	0	0	1	0
pyhala-braun 03	7383	24320	45	45	5	131
connamacher 975	568	7150	0	0	1	0
depots 3948	290	3587	9	27	1	345
driverlog 3963	170	1595	0	0	1	138
eulcbip 3936	135	672	0	0	1	0
ferry 3996	701	6661	0	0	1	403
hanoi4	718	4934	889	17	14	466
stanion 1617	162	774	0	0	1	0
ibm SAT_dat.k15	13818	69795	308	1	20	1460
linvrinv 565	792	2557	0	0	1	72
longmult8	3810	11877	0	0	23	1364
philips	3642	4456	0	0	19	76
pmg 3939	169	562	0	0	1	0
qg3-09	729	28463	1514	868	1	0
rovers 3978	229	3697	0	0	8	2682
anton 930	1000	3730	0	0	3	74
satellite 3985	158	3086	0	0	1	362
unsat350 20	350	1493	0	0	1	1
vdw_5_3_50	250	2953	0	28	1	0

The effect of removing blocked clauses (and adding all possible directly conflicting blocked binary clauses) highly depends on the benchmark type. If, for example, removing all blocked clauses and afterwards adding all possible directly conflicting blocked binary clauses is profitable for one instance of a family, it is generally for all instances of the family. However what is profitable for one kind of benchmark family might have a bad impact on another type of benchmark.

**Table 3.2:** Test set: Solve times in seconds using `march_dl` for different CNF preprocessings: The original formula, after removing all blocked clauses, after removing all blocked clauses and adding all possible directly conflicting blocked binary clauses.

benchmark	$ B_1 $	$ B_2 $	$F$	$F - B_1$	$(F - B_1) \cup B_2$
1000144.col	0	2142	2.41	-	<b>2.24</b>
pyhala-braun 03	131	5	1035.67	988.79	<b>979.68</b>
connamacher 975	0	1704	<b>1626.11</b>	-	1670.82
depots 3948	345	532	0.17	<b>0.16</b>	0.20
driverlog 3963	138	275	<b>0.07</b>	0.09	0.10
eulcbip 3936	0	0	-	-	-
ferry 3996	403	1494	1.69	<b>1.15</b>	3.84
hanoi4	466	374	44.72	50.56	<b>32.31</b>
stanion 1617	0	0	952.89	-	-
ibm SAT_dat.k15	1460	1019	120.70	<b>115.41</b>	138.63
linvrinv 565	72	1260	-	-	-
longmult8	1364	234	127.49	129.34	<b>109.58</b>
philips	76 <sup>1</sup>	76 <sup>1</sup>	<b>1467.77</b>	1760.09	1653.30
pmg 3939	0	0	655.41	-	-
qg3-09	0	0	105.21	-	-
rovers 3978	2682	242	0.18	<b>0.04</b>	0.09
anton 930	74	129	93.04	<b>90.11</b>	112.88
satellite 3985	362	766	<b>0.06</b>	0.06	0.13
unsat350 20	1	14	73.20	<b>71.11</b>	73.69
vdw_5_3_50	0	0	0.18	-	-

<sup>1</sup>Only 36 Of the 76 blocked clauses that could be removed were binary clauses, so the set of added directly conflicting blocked binary clauses differs from the set of removed blocked clauses.

# Chapter 4

## Connectivity

This chapter discusses some of the connectivity properties of practical SAT instances. What connectness information can be extracted from the graph representation? Is it possible to use this information to partition the graph? Perhaps it can be used to present a surveyable visualisation of the underlying CNF.

It may be expected that the connectivity of a random generated SAT problem is different from the connectivity of an industrial (and structured) SAT problem. Probably there is also a large difference of the connectivity amongst practical SAT instances. Some connectivity structures might yield good partitioning options whilst for others partitioning is simply not feasible.

In the following sections we go through different levels of detail looking at the connectness of the graph. The last section provides a single property indicating the connectivity of the graph.

### 4.1 Connected components

As mentioned in chapter 2 a graph is partitioned when it is disconnected. A disconnected graph consists of multiple connected components. Whenever a graph representation of a CNF problem right away consists of multiple connected components the divide and conquer principle can be directly applied. The only thing needed then is to identify the different connected components and with it its variables and clauses. Efficient algorithms to identify connected components within a graph already exist. So applying this will not be a problem.

The problem however is that graph representations of practical CNF problems consisting of multiple connected components are very rare. The  $\mathbf{rg}(F)$  and its refinement the  $\mathbf{srg}(F)$  do appear to have multiple connected

components for a lot of practical SAT instances. However, in general, all but one connected component have size 1. The 1-sized connected components are by definition blocked clauses and therefore redundant. First removing the blocked clauses from the CNF will take care of this phenomenon. Except for these *blocked clause connected components* none of the benchmarks from the test-set had more than one connected component.

## 4.2 Articulation points

Looking at some more detail to the connectivity of the graph representation we find *articulation points*. An articulation point is a separating vertex set consisting of only one vertex. Removal of an articulation point will increase the number of connected components. With the concept of articulation points one can identify several *biconnected components* within a single connected component. A *biconnected component* is a maximal subset of vertices of a graph  $G$ , such that the removal of a single vertex will not disconnect the component. The biconnected components can give a sort of coarse representation of the graph, yet giving one extra level of detail compared to connected components.

The complexity of an algorithm to find the articulation points, and thus the number of biconnected components, of a graph is  $O(V + E)$ .  $V$  being the number of vertices and  $E$  being the number of edges. Algorithms to find the articulation points are based on the *Depth-First Search Algorithm*. The number of articulation points in a graph can be determined exactly and no approximation is needed.

Unfortunately graph representations of practical CNF problems consisting of multiple biconnected components are also very rare. Multiple articulation points and biconnected components occur in the  $\mathbf{rg}(F)$  and its refinement  $\mathbf{srg}(F)$  of practical SAT instances. Yet again, in general, all but one biconnected component have size 1 and the 1-sized biconnected components are blocked clauses. Despite of the small separating vertex sets, removing these vertices to create 1-sized partitions (containing only redundant clauses) is not very interesting. Another possibly interesting option to look for separating vertex sets of a graph is through its *tree decomposition* as is discussed in the next section.

### 4.3 Tree decomposition

Still a coarse representation of the graph but yet more detailed than (bi)connected components we consider the *tree decomposition* of a graph. The concept of *treewidth* and the associated graph structure *tree decomposition* was introduced by Robertson and Seymour [33] in 1986. Although originally introduced in the context of their research on graph minors, these notions have proved to be of importance in computation complexity theory. Many NP-hard problems can be solved in polynomial time when the treewidth of the input graph is bounded by some constant [27].

Practical appliance of treewidth is used by Koster, van Hoesel and Kolen [28], they use tree decompositions to obtain lower bounds and optimal solutions for a special kind of frequency assignment problems. Tree decompositions are also used to solve problems in the field of expert systems. For a short overview the reader is referred to [27].

**Definition 4.3.1** *A tree decomposition of a graph  $G = (V, E)$  is a pair  $\chi, T$ , where  $T = (I, F)$  is a tree with node set  $I$  and edge set  $F$ , and  $\chi = (X_i \mid i \in I)$  is a family of subsets, called bags, of  $V$ , one for each node of  $T$ , such that*

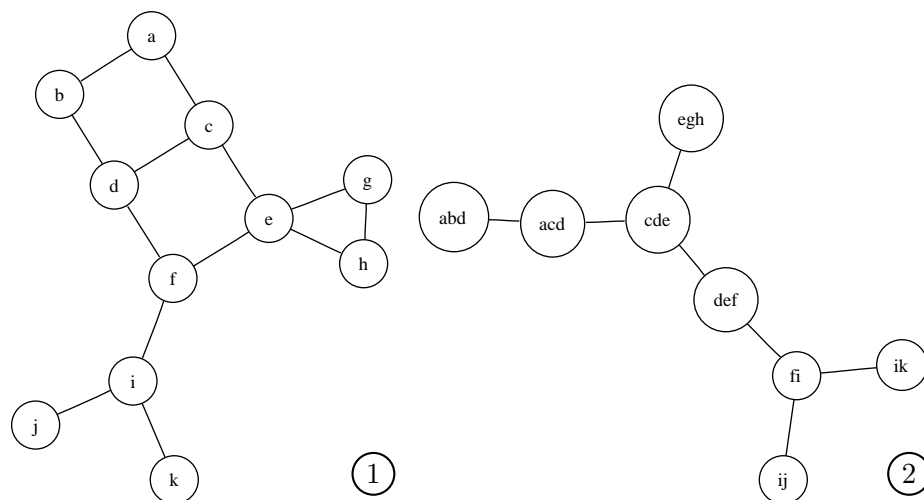
1.  $\cup_{i \in I} X_i = V$
2. For each edge  $\{v, w\} \in E$  there exists a bag with  $v, w \in X_i$
3. For each vertex  $v \in V$ ,  $T_v = \{i \in I \mid v \in X_i\}$  is a connected subtree of  $T$

The *width* of a tree decomposition  $(\{X_i \mid i \in I\}, T = (I, F))$  is  $\max_{i \in I} |X_i| - 1$ . The *treewidth*  $\tau(G)$  of a graph  $G$  is the minimum width over all tree decompositions of  $G$ .

A geometric representation of a graph and its tree decomposition is given in figure 4.1. The example is taken from Koster, Bodlaender and Hoesel [27].

Having a tree decomposition of small width in general implies that the graph has many splitting possibilities of small size [7]. The bags of the tree decomposition represent subgraphs of the graph. The edge, or *overlap*, between two adjacent bags is formed by the set of vertices that occur in both.

A tree decomposition can give good insight into the *locality* of the connectivity of a graph. In the representation graph of a randomly generated SAT problem each vertex can be connected to any other vertex. A randomly



**Figure 4.1:** (1) A geometric representation of a graph and (2) one of its tree decompositions. The tree decomposition has a width of 2.

generated SAT instance is therefore likely to have a tree decomposition with relative large bag size. The connectness of the (groups of) vertices is not very locally, but graph wide. The opposite, a tree decomposition of a graph representation of a SAT problem having a relative small bag size, means the representation graph must have some special structure.

The tree decomposition can also be useful to get a visual representation of the connectivity of a CNF problem. Directly visualizing one of the representation graphs of the CNF is only surveyable for very small problems (say less than 100 variables) and besides the connectivity structure gets lost in the high amount of detail in such representations. As we have seen in the previous sections depicting the representation graph as several (bi)connected components is not an option either. Using a visualization of a tree decomposition one can study the connectivity of a specific SAT problem in search of possible new solution strategies. Some examples and case studies are given in chapter 5.

The treewidth of a graph is, in general, NP-hard to compute. For small graphs the complete algorithm by Gogate and Dechter can be used [18]. Their algorithm has a worst time complexity of  $O(\min(V^{k+2}, V^{V-k}))$  with  $V$  being the number of vertices and  $k$  the treewidth. Several approximation algorithms for treewidth exist. In [27] four upper bound algorithms are given.

Three of them are based on determining whether a graph is *triangulated*. A *triangulated graph*, or *chordal graph*, is a graph where every cycle of at least length 4 contains a *chord*, that is, two non-consecutive vertices on the cycle are adjacent. A *triangulation* of a graph  $G$  is a triangulated graph  $H$  that contains  $G$  as a subgraph. What has this to do with the treewidth? Well:

1. The treewidth of a triangulated graph equals the maximum clique size minus one [16, 27].
2. For every graph  $G$  there exists a triangulation  $H$  with  $\tau(G) = \tau(H)$  [27].

So, finding the treewidth of a graph  $G$  is equivalent to finding a triangulation  $H$  of  $G$  with minimum clique size. The complexity of both problems is equal as well. However, any triangulation  $H$  of  $G$  will provide an upper-bound for the treewidth through its maximum clique size (minus one). The algorithms differ in the way a triangulation of the graph is created.

The fourth algorithm is based on incrementally improving a tree decomposition by decomposing the bags with the use of a minimum separating vertex set. The width of the resulting tree decomposition provides an upper-bound for the minimal treewidth [27].

Koster, Bodlaender and Hoesel also describe two lower bounds for the treewidth. The maximum clique size of a graph  $G$  minus one is the first lower bound described. The minimum degree of a graph also forms a (trivial) lower bound for the treewidth. More strict is the maximum of the minimum degree over all subgraphs of  $G$ , this is the second lower bound described.

If a tree decomposition of a graph is disconnected, the graph is disconnected as well. The disconnected parts of the tree decomposition can, by definition, only contain distinct sets of vertices. For each edge in the graph there should exist a bag containing both vertices of the edge. Since the disconnected parts contain distinct sets of vertices they cannot share edges. Thus the graph is disconnected as well.

The tree decomposition can be splitted by either removing a bag or an edge. For a tree decomposition of a graph representation of a CNF problem this means either satisfying a subproblem of the CNF (a bag) or satisfying the overlapping vertices (a set of variables for the variable oriented graphs or a set of clauses for the clause oriented graphs) between two bags. The overlap

between two bags is always smaller than the bag sizes. The best option to split a tree decomposition seems to be the removal of an edge. However, edge removal will just bipartite the tree decomposition, while bag removal might create a multipartition.

## 4.4 Diameter

As we have seen in the previous section, a tree decomposition can give good insight into the *locality* of the connectivity of a graph. Unfortunately no efficiently exact algorithm exists to create a tree decomposition with minimal width. Besides, approximation programs for tree width may come up with a higher upperbound for the  $\mathbf{rg}(F)$  than for  $\mathbf{cvg}(F)$  [21]. This while by definition  $E(\mathbf{cvg}(F)) \supseteq E(\mathbf{rg}(F))$  and thus the exact treewidth can never be larger. Another interesting property to get some insight in the locality of the graph connectivity is the *graph diameter*.

**Definition 4.4.1** *The diameter of the graph is the longest shortest path in a graph.*

The graph diameter can be computed efficiently. Algorithms exist with a time complexity of  $O(V^2E)$ . A graph having a relative large diameter is likely to have a kind of *stretched* structure. This increases the probability of the graph having a small treewidth. The opposite, a graph having a small diameter, can have either a small or large treewidth: A small graph diameter means that from every vertex there is a relative small path to all other vertices. This can be caused by just a few vertices that form ‘cornerstones’ in the CNF. If these vertices would be removed, only locally connected vertices would remain and the graph diameter would increase significantly. In a tree decomposition the locally connected vertices would be grouped in bags and the treewidth would still be low. Only the few ‘cornerstone’ vertices would form the overlap between the bags. It is also possible that there is no small group of ‘cornerstone’ vertices and a tree decomposition will have a large treewidth.

One can also use the diameter to find a separating set of the graph. If one groups all vertices by the length of the shortest path relative to one vertex, one of the middle groups of vertices having the same path length can be used (or its edges) as separating set. The distance between two vertices

having the longest shortest path is the diameter. Using one of these vertices to find a separating set will result in a set square to the length of the graph structure, this might increase the chance of finding a relative small set.

The graph diameter may also have another interesting field of appliance. It can be used to create a kind of splitting set of benchmarks for SAT solvers of different architecture.

For the **vig** of the test-set instances, table 4.1 shows the longest shortest path over all variables (diameter) and the average longest shortest path length per variable. Also both solving times on lookahead solver `march_dl` [22] and conflict driven solver MiniSat [13] are presented in the table.

**Table 4.1:** Test set: Longest shortest path over all variables, average longest shortest path per variable (of the **vig**) and solve times in seconds using look-ahead solver `march_dl` and conflict driven solver MiniSat.

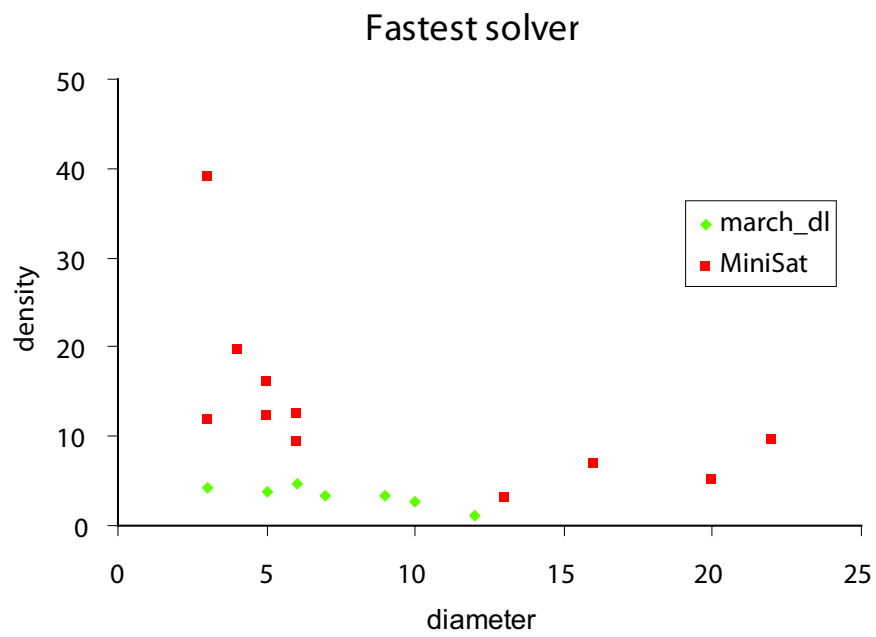
benchmark	type	vars	cls	diameter		solve time	
				max	avg	march_dl	MiniSat
1000144.col	<i>hand</i>	1200	3154	10	7.41	<b>2.41</b>	3.24
pyhala-braun 03	<i>hand</i>	7383	24320	7	6.10	<b>1035.67</b>	1257.96
connamacher 975	<i>hand</i>	568	7150	6	4.59	1626.11	<b>1520.14</b>
depots 3948	<i>indu</i>	290	3587	5	4.10	0.15	<b>0.03</b>
driverlog 3963	<i>indu</i>	170	1595	6	4.68	0.08	<b>0.02</b>
eulcbip 3936	<i>hand</i>	135	672	6	4.60	> 2000	> 2000
ferry 3996	<i>indu</i>	701	6661	22	16.11	1.69	<b>0.04</b>
hanoi4	<i>hand</i>	718	4934	16	11.98	40.61	<b>0.35</b>
stanion 1617	<i>hand</i>	162	774	6	4.92	<b>952.89</b>	1301.69
ibm SAT_dat.k15	<i>indu</i>	13818	69795	20	15.19	120.70	<b>2.39</b>
linvrinv 565	<i>hand</i>	792	2557	6	5.27	> 2000	> 2000
longmult8	<i>indu</i>	3810	11877	13	10.38	127.49	<b>64.69</b>
philips	<i>indu</i>	3642	4456	12	9.88	<b>1467.77</b>	> 2000
pmg 3939	<i>hand</i>	169	562	9	7.15	<b>655.41</b>	> 2000
qg3-09	<i>hand</i>	729	28463	3	3.00	105.21	<b>25.87</b>
rovers 3978	<i>indu</i>	229	3697	5	3.79	0.18	<b>0.04</b>
anton 930	<i>rand</i>	1000	3730	5	4.07	<b>93.04</b>	1366.88
satellite 3985	<i>indu</i>	158	3086	4	3.39	0.06	<b>0.03</b>
unsat350 20	<i>rand</i>	350	1493	3	3.00	<b>73.20</b>	> 2000
vdw_5_3_50	<i>hand</i>	250	2953	3	2.98	0.18	<b>0.03</b>

The idea is that so called *conflict driven* SAT solvers that try to learn from encountered conflicts by adding *conflict clauses* perform better on instances

for which relative short conflict clauses can be added. A large diameter means the structure of the problem is stretched and possible added conflict clauses should be short. The opposite holds for random problems. For each benchmark in table 4.1 the type of problem is provided: *indu* stands for an industrial problem, *rand* for a random problem and *hand* for a handmade problem.

The problem is that the test-set is not quite representative for this experiment. Most problems are too small and the variance of the diameter is very small. Besides, a small diameter does not necessarily mean the graph is “unstructured”. Still the diameter might be useful to form a sort of splitting set for the type of SAT problem. For instance, one can see that for industrial problems like *ferry 3996* and *ibm SAT\_dat.k15* with diameters of respectively 22 and 20 the problem is solved much faster by conflict driven solver *MiniSat*. The random problems *anton 930* and *unsat350 20* with diameters 5 and 3 were solved much faster on the architecture of *march\_dl*. This in contrast with the solve time results of the handmade problem *qg3-09* with diameter 3.

Nevertheless, when also taking the *density* of the SAT problems into account, one finds a simple linear relation that can classify the test-set by their fastest solver. The *density* of a formula is the ratio of the number of clauses to the number of variables. Figure 4.2 shows a plot of the test-set instances for their diameter and density. The instances are divided into two groups: *march\_dl* and *MiniSat* depending on their fastest solver.



**Figure 4.2:** Test-set: Plot of the diameter and density by their fastest solver.

# Chapter 5

## Visualisation

Using a visualization of a tree decomposition of one of the representation graphs one can study the connectivity of a specific SAT problem in search of possible new solution strategies. Directly visualizing one of the representation graphs of the CNF is only surveyable for very small problems and besides, the connectivity structure gets lost in the high amount of detail in such representations. As a coarse representation of the graph, the tree decomposition seems to be ideal for visualizing SAT problems.

Previous work on visualizing SAT instances is that of Slater [37] and Sinz [35]. Both Slater and Sinz use the **vig** and variants like *literal interaction graphs* and others to visualize the SAT instances. Sinz and Dieringer use a refined variant of the **vig** for their software tool **DPvis** [36] to visualize the structure of SAT instances and runs of the DPLL procedure. **DPvis** builds upon the commercially available graph layout package **yFiles** of **yWorks**<sup>2</sup> making use of implemented force-directed placement algorithms. Slater visualizes the resulting graphs with the **GraphViz** software from AT&T<sup>3</sup> and makes use of a hierarchical layouter. Sinz comments that this layouter, however, is not able to bring out the clustered structure of the instances. Although the **vig** is the smallest graph representation (not counting the hypergraph version, **vhg**) of the representations discussed in this thesis (see the number of edges for the test-set instances in table 2.1), we still wonder how much time and memory is spend to create visualizations of some larger SAT instances. Slater only provides visualizations of very small SAT instances.

The tree decomposition created are based on the refinements of the **cvg** since they embody more structural information about the SAT instance than

---

<sup>2</sup><http://www.yworks.com>

<sup>3</sup><http://www.research.att.com/sw/tools/graphviz>

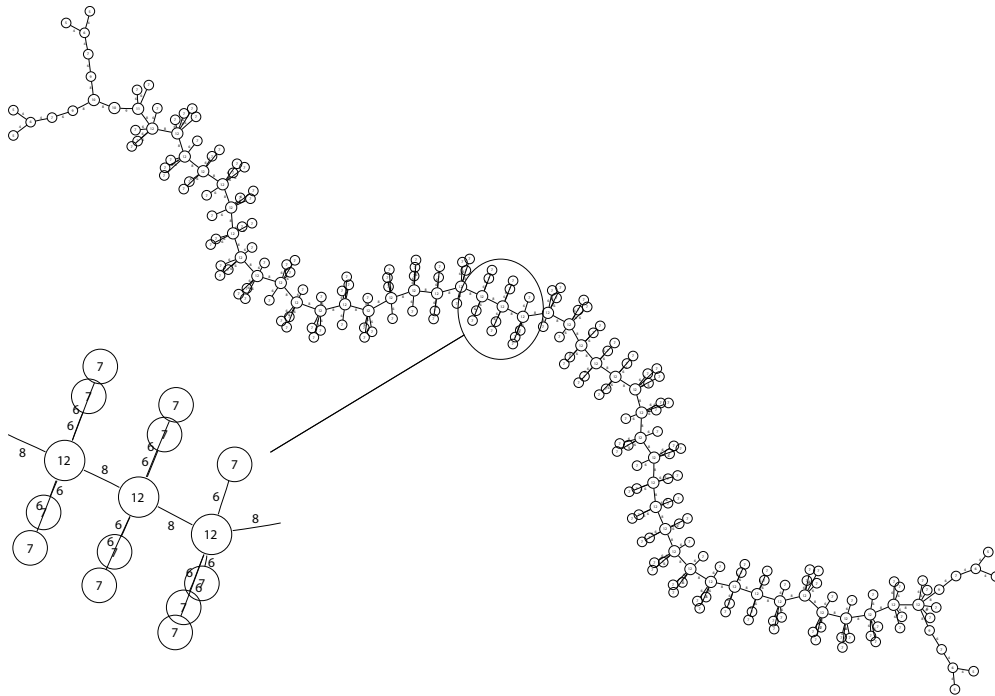
the **vig**. The tree decomposition captures the main connectivity properties of the graph and leaves a much more coarse representation. To actually create tree decompositions we used the approximation program `mdfi` from Arie Koster [23]. `mdfi` is based on the Minimum Degree Fill-in heuristic: Repeatedly a vertex of minimum degree is selected, removed from the graph and replaced by a clique of the remaining adjacent vertices to obtain a tree decomposition. Unfortunately this program uses quite an amount of memory and could only be used for the smaller SAT instances. To create tree decompositions of larger SAT instances a more efficient program is needed. For just visualization purposes the minimal tree width would be of minor importance. To visualize the resulting tree decompositions we make use of `GraphViz`. In contrast with Slater we also make use of a layouter (`neato`) that draws a graph by force-directed placement. However, for some geometric representations, like for instance the tree decomposition of `connamacher 975` a hierarchical layouter leaves the most surveyable view.

An interesting example is the tree decomposition of the resolution graph of `dubois50`<sup>4</sup>. `dubois50` is a handmade CNF instance from Olivier Dubois and consists of 150 variables and 400 clauses. The approximation program `mdfi` finds a tree decomposition of the resolution graph with width 11. This means the largest bag of the tree decomposition has size 12. The highest edge cardinality in our `dubois50` tree decomposition is 8. A geometrical representation of the tree decomposition of `rg(dubois50)` is given in figure 5.1.

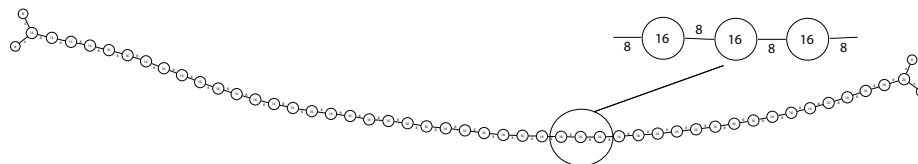
For a more clear visual representation tree decompositions can be coarsened by merging some of the nodes. An *x-reduced tree decomposition* is a tree decomposition where the smallest difference between a bag and its overlap is greater than  $x$ . A normal tree decomposition can be transformed to a  $x$ -reduced tree decomposition. One merges a node with its child node whenever the difference between the cardinality of the child node and the overlap with the parent is less or equal to  $x$ . This process starts at the largest bag, breadth-first working down all of its children. Figure 5.2 shows a 1-reduced tree decomposition of `rg(dubois50)`. Since bags may be merged during the reduce transformation the width of the tree decomposition may also increase. E.g., the 1-reduced tree decomposition of `rg(dubois50)` has a width of 15. Geometric representations of 1-reduced tree decompositions of the `rg` of (some of) the test-set instances can be found in Appendix B. In the following sections we will take a closer look at two of the test-set instances: `hanoi4` and `connamacher 975`.

---

<sup>4</sup>From the DIMACS Benchmark set, <http://www.satlib.org>



**Figure 5.1:** A geometric representation of a tree decomposition of the resolution graph of dubois50.



**Figure 5.2:** A geometric representation of a 1-reduced tree decomposition of the resolution graph of dubois50.

## 5.1 Case study: Tower of Hanoi

The *Tower of Hanoi* problem is well known. You have three pegs and a number of discs, varying in size, which can slide on to the pegs. Initial the discs are stacked in order of size, the smallest on top, on one of the pegs. The object is to move the discs to another peg, but there are two rules:

1. Only one disc may be moved at a time.

2. A disk may not be placed on a smaller one.

A simple recursive algorithm exists to solve the puzzle in the minimum number of moves. However, the minimum number of moves required is exponential in the number of discs. For  $n$  discs it takes a  $2^n - 1$  moves.

Kautz and Selman [25] translated the Tower of Hanoi problem into a SAT problem: Is there a movement solution that solves a  $n$  discs Tower of Hanoi problem in  $2^n - 1$  moves? In the context of the *Blocks world* problem, they developed a scheme of axioms to describe the possible actions.

- **on**( $x, y, i$ ) means object  $x$  is on object  $y$  at time  $i$
- **clear**( $x, i$ ) means there is room to move something on object  $x$  at time  $i$
- **move**( $x, y, z, i$ ) means object  $x$  is moved from object  $y$  to object  $z$  between time  $i$  and  $i + 1$

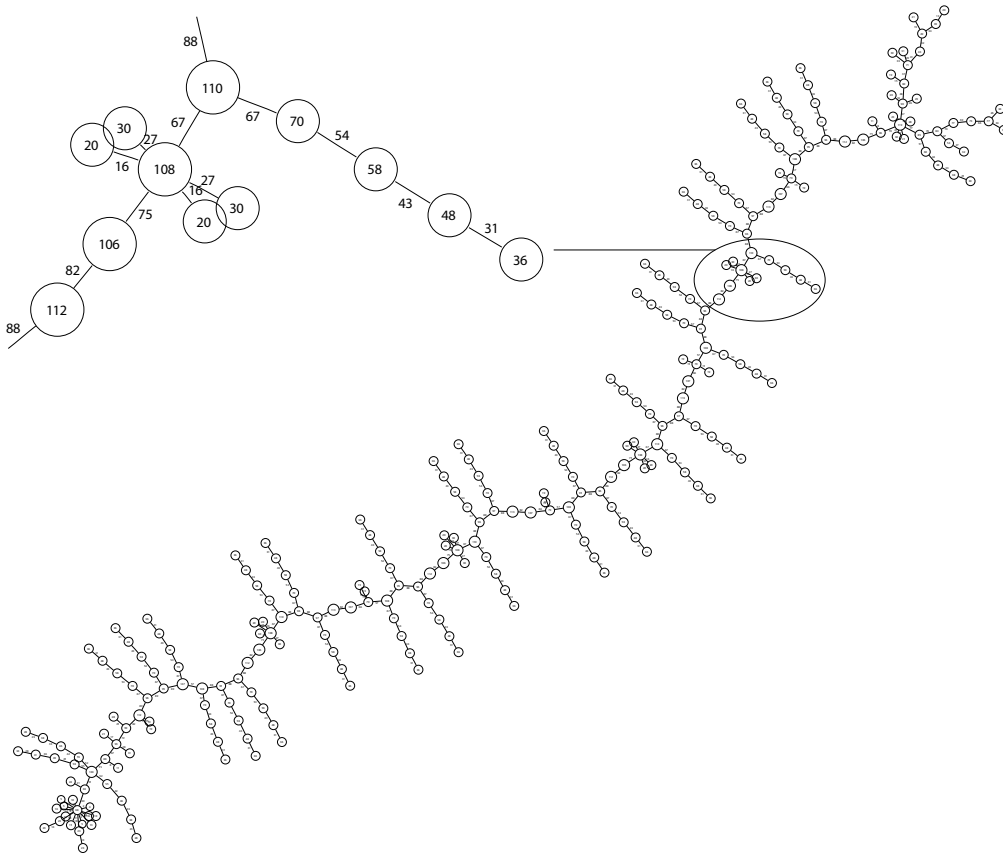
For efficiency they replaced the last predicate,  $\text{move}(x, y, z, i)$ , by three predicates with only two arguments each.

- **object**( $x, i$ ) means object  $x$  is being moved between time  $i$  and time  $i + 1$
- **source**( $y, i$ ) means something is being removed from object  $y$  between time  $i$  and time  $i + 1$
- **dest**( $z, i$ ) means something is being moved on top of object  $z$  between time  $i$  and time  $i + 1$

They used these predicates to create a SAT problem of the Tower of Hanoi. The beginning and end positions of the discs are encoded. Every timestep exactly one disc should be moved. A disc can only be moved if it is *clear* at the specific time. The 4 and 5 discs SAT versions are part of the DIMACS Benchmark Instances. A 6 discs SAT version was used in the SAT 2002 competition [34].

Now we are familiar with the Tower of Hanoi problem and its SAT translation we would like to investigate its tree decomposition. We use the resolution graph of the 4 discs version CNF, **hanoi4**, to create a tree decomposition. The **hanoi4** CNF consists of 718 variables and 4934 clauses. After preprocessing the problem by propagating unary clauses, removing duplicate clauses,

removing subsumed clauses and removing blocked clauses 2541 clauses and 541 free variables remain. The resolution graph of the preprocessed `hanoi4` problem consists of 17905 edges and, since it is clause based, 2541 vertices. This graph would be too large for a full representation. The 1-reduced tree decomposition obtained with the `mdfi` approximation of the resolution graph has 273 edges and, since it is a tree, 274 bags. Both standard and 1-reduced tree decomposition have a width of 146. A geometric representation of the 1-reduced decomposition tree is given in figure 5.3.

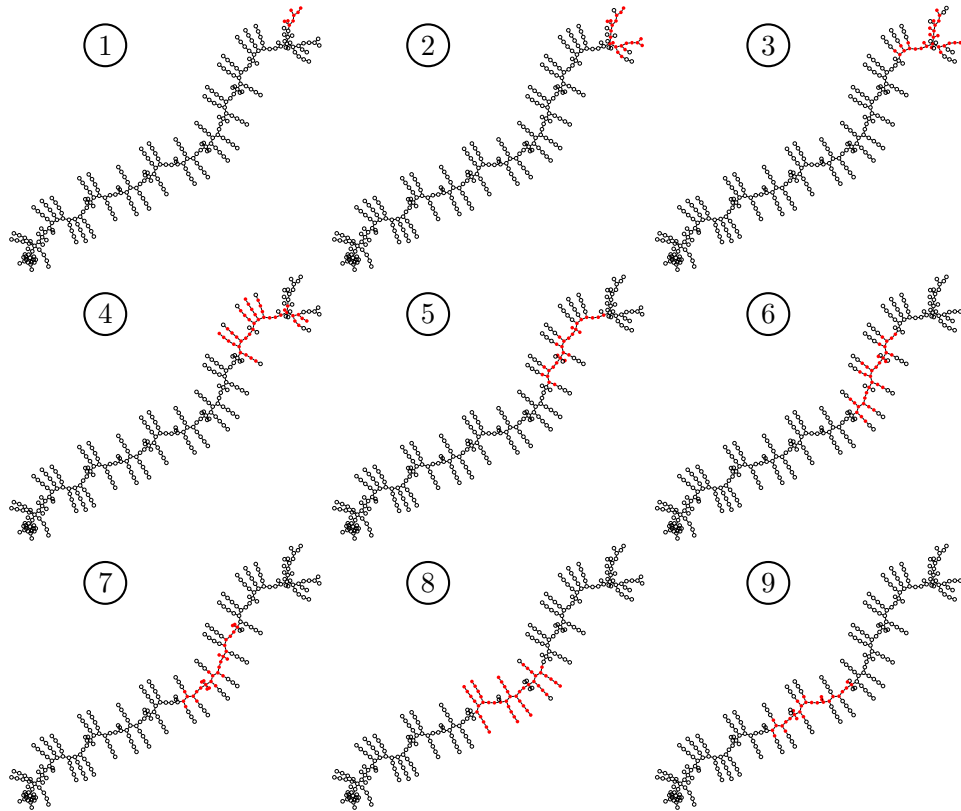


**Figure 5.3:** A geometric representation of a 1-reduced tree decomposition of the resolution graph of the preprocessed `hanoi4`.

How will the tree decomposition of `hanoi4` change if the problem is solved the ‘humanly way’. That is: Solving the puzzle starting from the beginning, working towards the goal setting. In table 5.1 the first nine steps are described together with their Kautz and Selman predicates describing the position of a specific disc at a specific time.

**Table 5.1:** The first nine steps in solving the 4 discs Hanoi puzzle along with their predicates for the CNF translation.

	human action	predicate
①	place A on PEG2	on(A,PEG2,2)
②	place B on PEG3	on(B,PEG3,3)
③	place A on B	on(A,B,4)
④	place C on PEG2	on(C,PEG2,5)
⑤	place A on D	on(A,D,6)
⑥	place B on C	on(B,C,7)
⑦	place A on B	on(A,B,8)
⑧	place D on PEG3	on(D,PEG3,9)
⑨	place A on D	on(A,D,10)



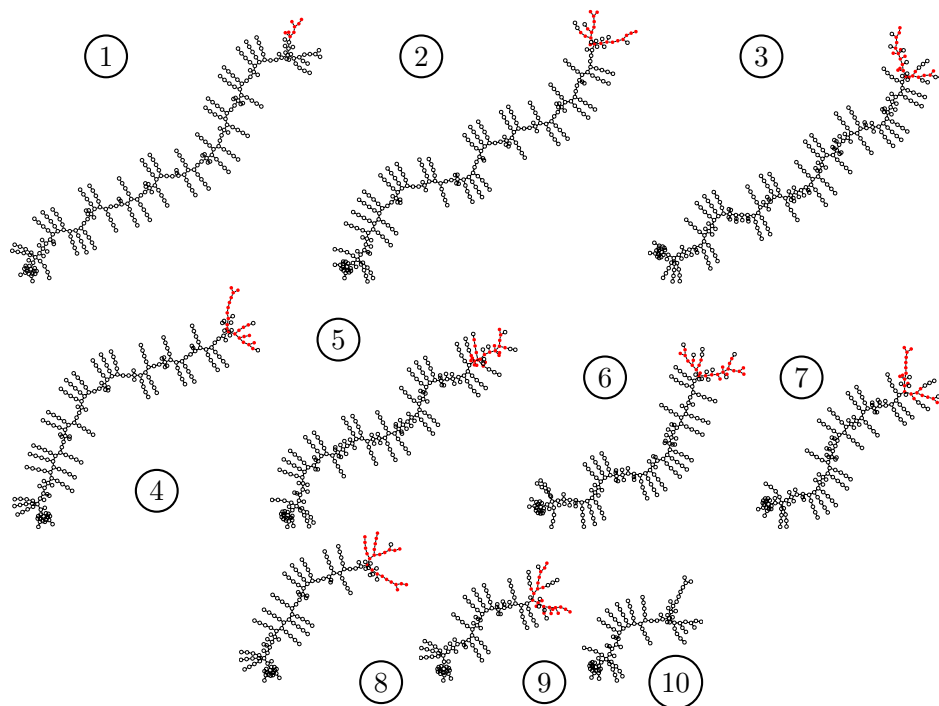
**Figure 5.4:** Nine small geometric representation of a 1-reduced tree decomposition of  $rg(hanoi4)$ , having red bags for the variables mentioned as predicates in table 5.1.

The discs are identified by the constants A, . . . , D, A being the smallest disc. The pegs are named PEG1, PEG2 and PEG3. Initial, at time 1, all discs

are placed from big to small on PEG1, the goal situation is all discs being placed on PEG3, again from big to small, at time 16.

Every predicate maps to a single variable in `hanoi4`. To visualize the location of a variable in the tree decomposition we color the bag red if one of its clauses contains the variable. Figure 5.4 shows small 1-reduced tree decompositions of the `rg` with the locations of the variables mentioned as predicates in table 5.1. One can see the locations of the different variables shift along the tree decomposition, starting at the right top going towards the ‘tail’ of the tree decomposition at the bottom left.

Figure 5.5 also shows small tree decompositions with the location of the variables of `hanoi4`, but now the variables are actually set each time. Setting a variable can easily be realized by adding it as a unary clause to the CNF. Preprocessing the CNF as mentioned earlier will propagate the unary clause throughout the rest of the CNF. Now one can see that the different locations of the variables are always located at the ‘head’ of the tree decomposition structure. Setting successive moves in time sort of ‘eats away’ the structure following the ‘spinal core’. The connectivity of the tree decomposition along the spinal core represents in certain way the element of time within the `hanoi4` problem.



**Figure 5.5:** Ten small geometric representations (all of the same scale) of 1-reduced tree decompositions of  $\text{rg}(\text{hanoi4})$  with previous variables already set, having red bags for the variables mentioned as predicates in table 5.1.

## 5.2 Case study: Connamacher

We also will take a better look at the tree decomposition of a Connamacher CSP instance [8]. The problem considered is the generic uniquely extendible constraint satisfaction problem  $(k, d)$ -UE-CSP. In  $(k, d)$ -UE-CSP, each constraint is over a  $k$ -tuple of variables, each variable must take a value from the domain  $\{0, \dots, d - 1\}$ , and every constraint is uniquely extendible.  $k$ -XOR-SAT is exactly  $(k, 2)$ -UE-CSP.

In [9] Connamacher and Molloy prove that  $(3, 4)$ -UE-CSP is NP-complete and that the satisfiability threshold for random  $(3, 4)$ -UE-SAT is  $c_k \approx .917935$ . The satisfiability threshold conjecture is that there exists a value  $c_k$  such that a random SAT formula on  $n$  variables and  $cn$  clauses, with  $n$  tending to infinity, is almost surely unsatisfiable if  $c > c_k$  and almost surely satisfiable if  $c < c_k$ .

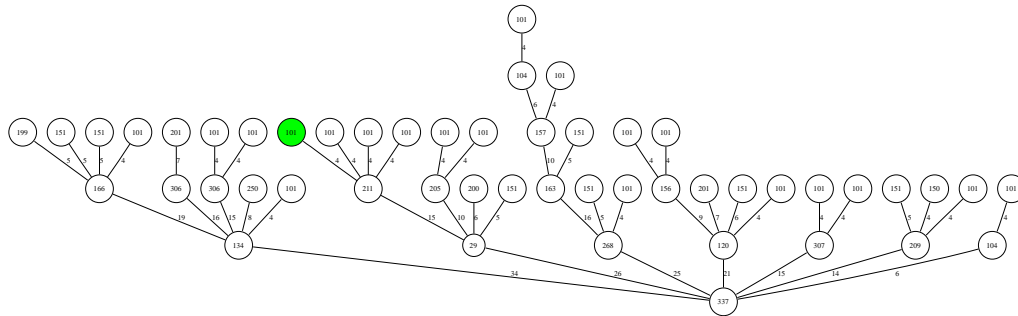
Translating a single  $(k, d)$ -UE-CSP constraint to SAT results in  $k$  clauses of length  $d$  and a number of  $k$  sized clauses having one variable from each  $d$  sized clause minimally enforcing exact one (Boolean) variable per  $d$  sized clause can be set to `true`.

We use the resolution graph of a SAT instance of a  $(3, 4)$ -UE-CSP having 600 constraints and a clause density of 4.00% above the satisfiability threshold, to create a tree decomposition. The CNF consists of 568 variables and 7150 clauses and was proven unsatisfiable during the SAT 2004 competition [3]. The instance is known as `connamacher 975` in the test set. Preprocessing this problem did not reduce the number of clauses or variables. Figure 5.6 shows a geometric representation of the 1-reduced decomposition tree obtained with `mdfi`. The 1-reduced tree decomposition has a width of 336 and has only 49 edges. The original tree decomposition had a width of 188.

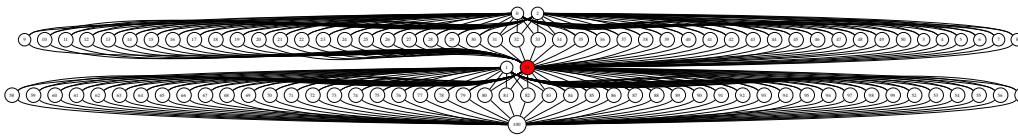
The connectivity structure of the `connamacher` problem differs clearly from the stretched structures we have seen for `dubois50` and `hanoi4`. Also interesting is the fact that the leaf bags are of (relative) large size and a lot of them are of the same size, namely 101.

We can track down the subgraph that is represented by a single bag. As an example we take the green bag from figure 5.6. And show a geometric representation of the resolution graph in figure 5.7 (Now we see that such a representation already gets quite messy for only 101 vertices and 144 edges). The CNF of this single bag consists of 101 clauses, of which five are 4-literal

clauses. Four of these clauses also appear in the parent bag (are part of the overlapping edge) All other clauses are 3-literal clauses.



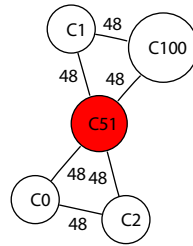
**Figure 5.6:** A geometric representation of a 1-reduced tree decomposition of the resolution graph of connamacher 975 with one bag marked green.



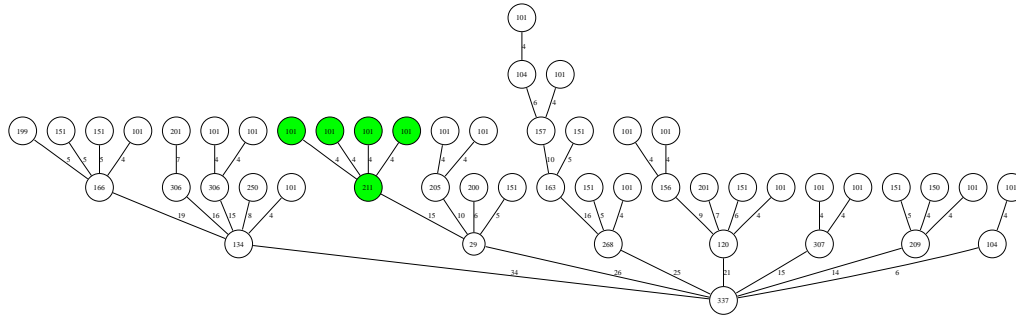
**Figure 5.7:** A geometric representation of the  $rg$  of the green bag from the 1-reduced tree decomposition of connamacher 975 from figure 5.6

The graph consists of two biconnected components, of which each component has 48 vertices (the 3-literal clauses) all having edges with 3 other vertices (the 4-literal clauses). Both biconnected components share one vertex, the articulation point. The articulation point is red in figure 5.7.

This kind of triangular form of three 4-literal clauses being connected by 48 3-literal clauses appears to be a sort of building block for the entire graph. We can create a kind of simplified resolution graph by only keeping the vertices having more than three edges. The cardinality of the (in)direct connections among the kept vertices is placed along the (new) edges between them. The nodes in these simplified resolution graphs represent single vertices having more than three edges in the  $rg$ . Such a simplified graph for the graph depicted in figure 5.7 is shown in figure 5.8. A simplified graph for the subgraph formed by the green bags of figure 5.9 is shown in figure 5.10. Articulation points are again in red. The entire resolution graph of this connamacher instance is far more connected and has no articulation points.

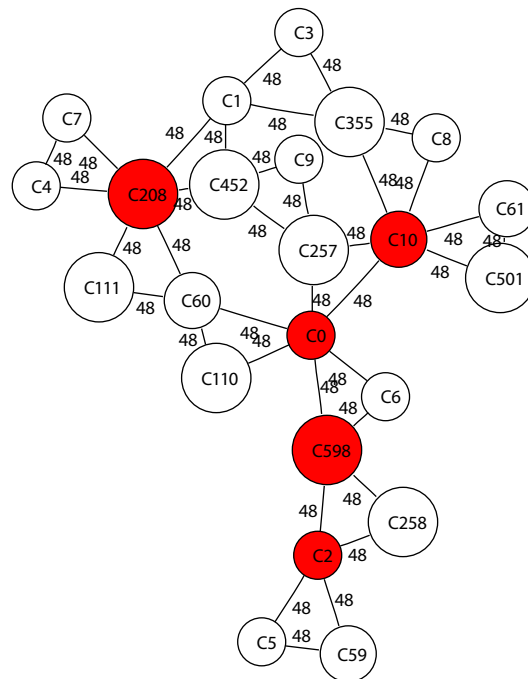


**Figure 5.8:** A geometric representation of the simplified graph of figure 5.7



**Figure 5.9:** A geometric representation of a 1-reduced tree decomposition of the resolution graph of connamacher 975 with multiple bags marked green.

The building blocks of three 4-literal clauses being connected by 48 3-literal clauses of course each represent a single constraint of the  $(3, 4)$ -UE-CSP. They were however discovered from the CNF only without knowing the actual problem being encoded by the CNF. Similar techniques could be used for detecting structures in unknown SAT instances.



**Figure 5.10:** A geometric representation of the simplified graph of the subgraph of  $\text{rg}(\text{connamacher } 975)$  formed by the green bags in figure 5.9

## 5.3 Conclusions

We have seen that tree decompositions of representation graphs can be visualized efficiently. The number of edges was reduced significantly for the tree decomposition of the graphs. For an even more coarse representation the tree decomposition can be easily transformed to a  $x$ -reduced tree decomposition.

One can study several properties like modularity and repetition of the CNF that can be observed through the tree decomposition. There is the possibility of zooming in on (some of) the bags to get a better understanding of the low scale relations of the CNF.

These techniques can be used on specific SAT problems in the search for possible new solution strategies.

# Chapter 6

## Applying splitting

When the representation graphs do not fall apart naturally, we will take care of it unnaturally. In this chapter we look for good splitting possibilities of the representation graph.

What is a *good splitting possibility*? Any separating vertex or edge set will disconnect the graph on removal. What properties makes one separating vertex or edge set favoured over another one? From the perspective of the SAT problem two things can be mentioned.

First of all one would prefer a separating set having a small set of satisfying assignments. As mentioned in chapter 2: A variable can simply be removed by assigning it a truth value and a clause can be removed by any satisfying assignment of its set of variables. For a separating set of variables this is simply the smallest set. For a separating set of clauses this is somewhat more complex. One would not directly prefer the smallest separating clause set, but the separating clause set having the smallest set of satisfying assignments. Finding the smallest set of satisfying assignments is a whole problem on itself, so the most straightforward manner to partition a SAT representation graph is to use the variables as separating set.

The second thing one would prefer is a well-balanced partition. For a bipartition the theoretical highest reduction of the solution space is reached with a fifty-fifty balanced partition. In general (also for splitting in multiple parts) one would want the largest part to be as small as possible. As we will see, aiming for a well-balanced partition is often a trade off with the size of the separating set. A better balanced partition often has a larger separating set.

## 6.1 Balanced minimal cut

The width of a tree decomposition is an indication of the size of a (relative small) separating set. However, additional information is needed to find a separating set leaving a well-balanced partition. The tree width is just an upperbound, perhaps a smaller bag can be used as separating set to get a well-balanced partition. Besides, using tree decompositions is far too expensive.

One could also use the graph diameter to find a separating set, but this would require quite some adaption. Firstly to find a small separating set and secondly to find one that leaves a well-balanced partition. Simply using the group of vertices having a specific distance to the vertex with the longest shortest path will probably result in not very optimal balanced minimal cut.

Graph partitioning and with it finding the minimal cut is a problem which appears in a wide range of applications such as VLSI design, efficient storage of large databases on disks, communication in parallel computing and data-mining.

Polynomial time algorithms exist to find a minimal graph cut. Finding a *balanced* minimal cut however, is NP-complete in general [15]. Simply applying a minimal cut without any constraints on the partition sizes often results in trivial cuts that leave one partition very large relative to the original graph size. Many heuristic algorithms have been developed. Alpert and Khang [1] provide a survey with descriptions and comparisons of various of these schemes. Important methods in this field are the `KernighanLin` heuristic [26] and the `FiducciaMattheyses` refinement heuristic [14]. In these methods, a vertex is moved (or a vertex pair is swapped) if it produces the greatest reduction in the edge cuts. Applying such schemes on a randomly initial section often has poor results. More intelligent initial sections will result in slow performance on larger graphs. More modern techniques are based on multilevel partitioning algorithms. First the graph is coarsened by a sequence of successively smaller graphs. An initial section is applied on the coarse graph. Step by step the graph is uncoarsened and refinements are made at each step using the standard refinement heuristics.

We use the state-of-the-art hypergraph partitioning tool `hMeTiS` [24]. Using this tool we can search for the approximated minimal cut in SAT representation graphs by separating edges representing variables. The `cvg` and its refinements might have several edges represent a single variable. This obscures the search for the smallest variable set. The `vig` has the same separating vertex set as the hypergraph version it is obtained from, the `vhg`. A separating vertex set of a hypergraph is the same as the separating edge set

of its dual (A proof is given in Appendix C). So the **chg** can be used to find a minimal cut on the variables by its separating hyperedge set.

Table 6.1 shows the approximated tree width of the **vig** obtained with **mdfi**, the separating vertex set of the **vig** using the diameter (ds-set) and the size of the separating hyperedge set of the **chg** using **hMeTiS** for a bipartition with at least a balance of 25% for the test-set. The actual resulting balance for each particular separating hyperedge set is also given. The SAT instances were first preprocessed by propagating unary clauses, removing duplicate clauses, subsumed clauses and blocked clauses.

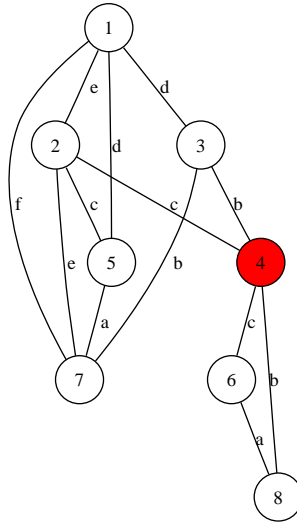
**Table 6.1:** Test set: Approximated tree width of **vig** using **mdfi**, separating vertex set of **vig** using the diameter (ds-set) and balanced approximated minimal cut of **chg** using **hMeTiS** along with its resulted balance.

benchmark	vars	cls	tree width	ds-set	cut edges	balance
1000144.col	1200	3154	353	421	<b>189</b>	0.26
pyhala-braun 03	7383	24320	<sup>5</sup> -	3114	<b>71</b>	0.28
connamacher 975	568	7150	176	275	<b>118</b>	0.26
depots 3948	290	3587	95	113	<b>58</b>	0.42
driverlog 3963	170	1595	39	54	<b>18</b>	0.30
eulcbip 3936	135	672	47	53	<b>24</b>	0.25
ferry 3996	701	6661	161	23	<b>21</b>	0.38
hanoi4	718	4934	140	40	<b>22</b>	0.45
stanion 1617	162	774	38	48	<b>23</b>	0.25
ibm SAT_dat.k15	13818	69795	<sup>5</sup> -	1485	<b>69</b>	0.50
linvrinv 565	792	2557	90	316	<b>57</b>	0.25
longmult8	3810	11877	72	428	<b>26</b>	0.40
philips	3642	4456	105	389	<b>34</b>	0.27
pmg 3939	169	562	43	48	<b>19</b>	0.26
qg3-09	729	28463	<sup>5</sup> -	<b>268</b>	296	0.50
rovers 3978	229	3697	55	67	<b>28</b>	0.44
anton 930	1000	3730	<sup>5</sup> -	490	<b>337</b>	0.25
satellite 3985	158	3086	61	74	<b>50</b>	0.46
unsat350 20	350	1493	241	272	<b>174</b>	0.25
vdw_5_3_50	250	2953	157	155	<b>85</b>	0.29

<sup>5</sup>The instance was too large for **mdfi** to compute an approximated tree width.

In the **chg** all vertices (the clauses) containing the same variable are connected by one hyperedge (representing the variable). In the **rg** (and the **srg**) it is possible that clauses connected through the same clashing literal form distinct groups. To illustrate this we use the sample CNF:

$$R = \{\{-d, e, \neg f\}, \{\neg e, \neg c\}, \{b, d\}, \{\neg b, c\}, \{a, c, d\}, \{\neg a, \neg c\}, \{\neg a, \neg b, e, f\}, \{a, b\}\}$$



**Figure 6.1:** A geometric representation of  $\mathbf{rg}(R)$ .

A geometric representation of  $\mathbf{rg}(R)$  is given in figure 6.1. Clause 4 is an articulation point and is colored red. The issue is that variable  $a$  is connecting clause 5 and 6, and connecting clause 7 and 8, but these pairs are not connected to each other by variable  $a$ . In the  $\mathbf{chg}(R)$  clause 5, 6, 7 and 8 would be connected with one hyperedge representing variable  $a$ . While in the  $\mathbf{rg}(R)$  the variables  $\{b, c\}$  would form a nice separating set, for the  $\mathbf{chg}(R)$  this should include variable  $a$  as well. So, perhaps we can obtain even smaller cuts by representing each separate clause group in the  $\mathbf{rg}$  connected by one variable by one hyperedge in a hypergraph. We call this hypergraph the *resolution hypergraph*, **rhg**. A *subsumption-resolution hypergraph*, **srhg**, can be constructed likewise. Eight of the test-set instances had one or more variables that represent multiple hyperedges in the **rhg**. All these variables had exactly two hyperedges. Table 6.2 shows the number of hyperedges of the **chg** and the **rhg** for these eight instances.

Notice that, although by definition  $E(\mathbf{chg}(F)) \subseteq E(\mathbf{rhg}(F)) \subseteq E(\mathbf{srhg}(F))$ , the refinements of the **chg** have better splitting possibilities. While for the

**Table 6.2:** Test set benchmarks having multiple hyperedges for variables in the **rhg**: Edges of the **chg** and the **rhg**.

benchmark	vars	cls	$E(\mathbf{chg})$	$E(\mathbf{rhg})$
pyhala-braun 03	7383	24320	6669	8792
hanoi4	718	4934	533	534
stanion 1617	162	774	161	210
ibm SAT_dat.k15	13818	69795	9581	13759
linvrinv 565	792	2557	756	1368
longmult8	3810	11877	2299	3676
philips	3642	4456	1391	2043
anton 930	1000	3730	946	948

normal graphs having less edges increases the likelihood of a smaller separating vertex set, the opposite holds for these hypergraphs. Each hyperedge represents a clique of vertices being connected to each other. Splitting up a single hyperedge leaves distinct smaller cliques, this increases the likelihood of having a smaller separating vertex set. Since both the **rg** and the **srh** are decomposable graph representations, so are the **rhg** and the **srhg**.

For three of the eight instances the minimal cut obtained with the approximation program **hMeTiS** resulted in more hyperedges cut than variables (with a maximum of three), so for some variables both hyperedges were cut. The number of actual variables in the separating set of **rhg** did not significantly differ from the number of variables in the separating set for the **chg** for the test-set instances. The question remains how (and whether) this theoretical additional *decomposition intelligence* can be utilized.

## 6.2 Break-set variables

As seen in the previous section, by far the most practical method to apply splitting is using the separating set of variables found by a heuristic minimal cut algorithm. If one assigns truth values to these variables the CNF is partitioned. In this section we discuss what strategy should be used to make use of this *variable break-set*.

**Definition 6.2.1** *A variable break-set is a subset of variables of a SAT problem  $F$ , such that all satisfying assignment will partition  $F$ .*

How to embody these break-set variables in the solving process? The idea is that first setting these variables will simplify the problem instance

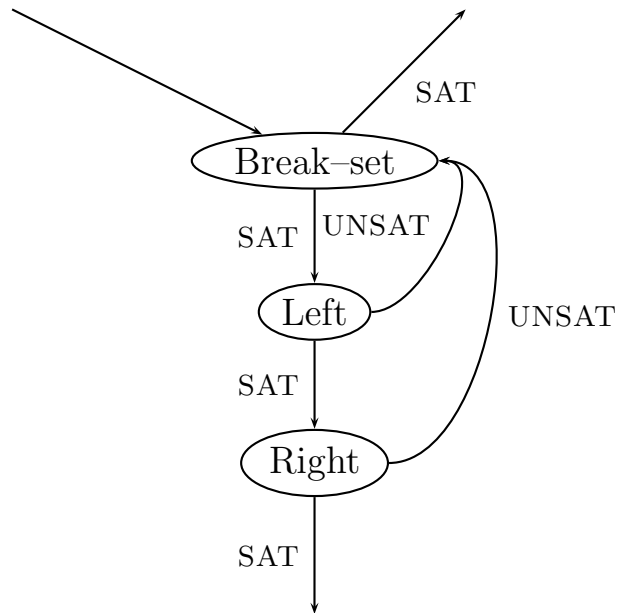
greatly. Therefore we would like to have an implementation of the DPLL procedure that gives priority to the break-set variables for the branching literal. But even more adjustments are necessary. Once a truth value is assigned to all variables of the break-set, the SAT problem is partitioned. All remaining partitions can be solved independently. However, if one of the partitions is unsatisfiable under a specific assignment of the variable break-set, backtracking all the way back to the setting of the variable break-set is needed and settings for all other partitions should be undone as well.

Each partition can again have its own break-set variables that will partition the partition, so splitting can be applied on multiple levels. For simplicity of the branching strategy we will think of a partition as a *bipartition*. Now we can name the (sets of variables of the) partitions *Left* and *Right*. The partition variable branching strategy is as follows:

1. Construct a variable break-set.
2. Branch on the variables of the break-set. If no satisfying assignment can be found for this set, the entire problem is *unsatisfiable*. If a satisfying assignment can be found then proceed with the next step.
3. Branch on the Left set variables. If no satisfying assignment can be found for this set then undo this setting and return to step 1 to find another assignment for the break-set variables. If a satisfying assignment can be found then proceed with the next step.
4. Branch on the Right set variables. If no satisfying assignment can be found for this set then undo both settings for Right and Left set variables and return to step 2 to find another assignment for the break-set variables. If a satisfying assignment can be found then the entire problem is *satisfiable*.

Figure 6.2 shows a graphical representation of the branching method.

A refinement can be made to the strategy by swapping the Left and Right set of variables if no satisfying assignment was found for the Left set variables in step 3. This prevents repeatedly working through a relative *easy* partition to end up with the relative *hard* one. Swapping the partitions will increase the chances to find a good assignment for the break-set variables early on, by directly adapting their assignment to both partitions. The selection of the break-set variables can be implemented in a variety of ways.



**Figure 6.2:** A graphical representation of the branching method for a variable break-set and bipartition.

### 6.3 Case study: Tower of Hanoi

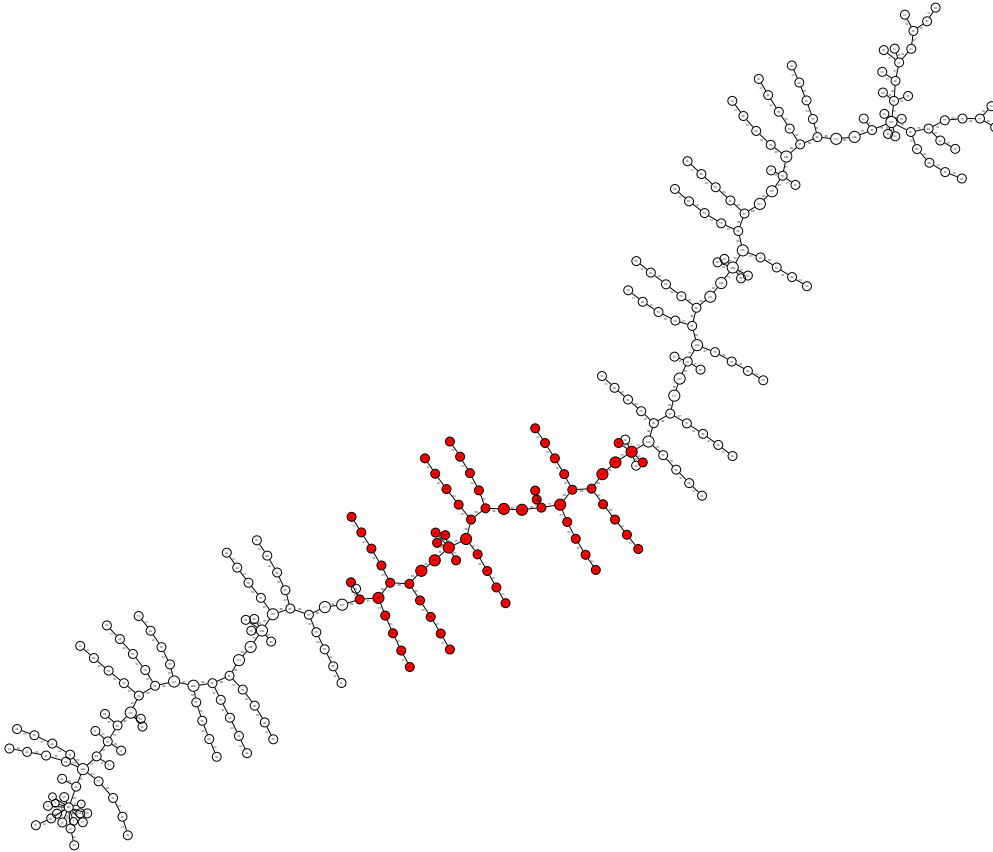
The `hanoi4` problem is already studied in chapter 5. With the use of `hMeTiS` we find an approximated balanced minimal cut of 22 hyperedges for a 0.45/0.55-bipartition of the `chg(hanoi4)`. These hyperedges represent variables, so we have a variable break-set of size 22. The set of the predicates these variables match with:

$$\{\text{on}(A,B,10), \text{on}(A,C,10), \text{on}(A,D,10), \text{on}(A,PEG1,10), \text{on}(A,PEG2,10), \text{on}(A,PEG3,10), \\ \text{on}(B,C,10), \text{on}(B,D,10), \text{on}(B,PEG1,10), \text{on}(B,PEG2,10), \text{on}(B,PEG3,10), \text{on}(C,D,10), \\ \text{on}(C,PEG1,10), \text{on}(C,PEG2,10), \text{on}(C,PEG3,10), \text{on}(D,PEG1,10), \text{on}(D,PEG2,10), \\ \text{on}(D,PEG3,10), \text{clear}(A,10), \text{clear}(B,10), \text{clear}(C,10), \text{clear}(D,10)\}$$

The variables of the break-set actually set the position of the discs at time 10. It is for this specific problem easy to see that if all break-set variables

are set (i.o.w.: the position of the discs at time 10 are fixed) the problem falls apart into two subproblems. Figure 6.3 shows a geometrical representation of a 1-reduced tree decomposition of  $\text{rg}(\text{hanoi4})$  in which the bags containing clauses having a break-set variable are colored red.

1. From starting position to the fixed position at time 10
2. From the fixed position at time 10 to the finishing position



**Figure 6.3:** A geometric representation of a 1-reduced tree decomposition of the resolution graph of the preprocessed `hanoi4`, having red bags for the break-set variables.

Knowing the hanoi problem one would expect the number of possibilities of the positions of the discs at time 10 to rapidly reduce to 81. For every disc has to be on one peg and it also can be at only one PEG at a time. Once the position of the discs is known, it directly follows if a disc is still lying free. So there are three possibilities for disc `D`, again three possibilities for disc `C`, etc.:  $3^4 = 81$ . Unfortunately these rules are not embodied this way in the CNF.

For there are no direct rules a disc has to be on a peg at each time. It is only prescribed what the starting and end position of each peg is and that each time exactly one peg should be moved. Also the rule allowing maximal one disc directly on another disc ( $\text{on}(A,D,i)$  and  $\text{on}(B,D,i)$ ) is not present (though it is for the pegs). That is hidden in the rule that a disc may only be moved when it is free and only may be moved on a disc that is free (or a free peg). And that is all depending on the time steps before and after.

Splitting the `hanoi4` problem does not seem to be quite appropriate in this context. From chapter 5, we know that the 4 discs hanoi problem can be solved in 15 moves and that each move can be mapped to a single variable in the `hanoi4` CNF. So, setting the right 15 variables to `true` will solve the problem, this while partitioning the CNF will take the setting of 22 variables.

## 6.4 Splitting in practice

The lookahead SAT solver `march_dl` [22], developed at Delft University of Technology, was modified to handle the break-set information. Additional information was added to the CNF instances so the variables could be recognized as either break-set variable, Left set variable or Right set variable.

`march_dl` is sophisticated in the way it applies lookahead on only the group of variables of the reduced clauses from the previous step [22]. Applying look-ahead on a variable is a quite expensive procedure, so we would like to have only a small group of variables to apply look-ahead on. On the other side, branching on a good variable can reduce the solve time significantly. What might be a good look-ahead mechanism for variables in the context of the break-set variables?

Three implementations were created of a modified `march_dl` solver that were able to handle the break-set information. The implementations differ in the way look-ahead is performed on the break-set variables.

- The `full look-ahead (FLA)` implementation applies look-ahead on all variables while selecting break-set variables.
- The `affected look-ahead (affFLA)` implementation applies look-ahead on all break-set variables and on the partition variables from reduced clauses from the previous step.
- The `full look-ahead 500 (FLA 500)` implementation also applies look-ahead on all variables while selecting break-set variables, but only when

the nodecount is smaller than 500. The nodecount keeps track of the number of assignments already made.

There is also a variant on the last implementation which swaps the partitions in such a way that the partition last reached in the branching process is the first to start with after the current setting of the break-set variables appears to be unsatisfiable.

- The `full look-ahead partition switch 500 (FLAps 500)` implementation is the same as the `full look-ahead 500` implementation, but now it swaps the partitions in such a way that the last reached partition in the branching process is the first to start with after the current setting of the break-set variables appears to be unsatisfiable.

Table 6.3 shows the solving times of the different implementations on the test-set instances which have a minimal break set for a balanced bipartition. All unit clauses were propagated and duplicate clauses, subsumed clauses and blocked clauses were removed for the benchmarks.

We found that for a number of unsatisfiable instances even no satisfying assignment could be found for just the break-set variables. Together with the number of partition swaps made this is presented in table 6.4.

For six of the unsatisfiable benchmarks different balanced bipartitions were tested to study the effect of finding a satisfying assignment for the break-set variables. A higher unbalance will often allow a smaller break-set. Is there a transition where on one side the break-set is so small that partition information is needed to find unsatisfiability and on the other side that the unsatisfiability can be found within the break-set variables?

Five different runs of `hMeTiS` for eight different balance categories were executed for the `connamacher 975` test set instance. One can also see the different behaviour of the modified solver on slightly different cut. Unfortunately `hMeTiS` is a non-deterministic program. Running twice with exactly the same parameters can result in different partitions. Using the unmodified `march_dl` solver `connamacher 975` is solved in 1252.61 seconds.

**Table 6.3:** Test set: Solve times in seconds using `march_dl` and different modifications implementing partition branching.

benchmark	sat	original	FLA	affFLA	FLA 500	FLAps 500
1000144.col	U	<b>1.75</b>	2.75	1.79	2.67	2.68
pyhala-braun 03	U	<b>907.32</b>	> 2000	> 2000	> 2000	> 2000
connamacher 975	U	1252.61	1016.72	> 2000	517.43	<b>508.73</b>
depots 3948	S	0.17	<b>0.15</b>	0.16	0.17	0.19
driverlog 3963	S	0.09	0.08	0.10	0.08	0.09
eulcbip 3936	-	> 2000	> 2000	> 2000	> 2000	> 2000
ferry 3996	S	<b>1.22</b>	1.57	2.56	1.58	1.57
hanoi4	S	39.44	4.37	25.94	4.41	<b>3.98</b>
stanion 1617	U	<b>936.82</b>	> 2000	> 2000	> 2000	> 2000
ibm SAT_dat.k15	S	138.63	> 2000	<b>85.03</b>	292.91	292.97
linvrv 565	-	> 2000	> 2000	> 2000	> 2000	> 2000
longmult8	U	137.07	65.85	<b>55.38</b>	64.35	91.58
philips	U	1438.46	> 2000	> 2000	1412.49	<b>1403.53</b>
pmg 3939	U	<b>502.37</b>	577.82	617.63	610.39	615.49
qg3-09	U	<b>39.13</b>	50.21	63.62	51.31	50.27
rovers 3978	S	0.09	0.09	0.10	0.09	<b>0.08</b>
anton 930	U	<b>90.62</b>	340.95	164.53	183.20	179.05
satellite 3985	S	<b>0.07</b>	0.10	0.10	0.12	0.10
unsat350 20	U	<b>68.77</b>	119.66	75.06	75.74	74.41
vdw_5_3_50	S	0.17	<b>0.16</b>	0.19	0.39	0.33

**Table 6.4:** Test set: Sizes of the variable break-set ( $V_{bs}$ ), Left and Right partition, the number of partition swaps and whether a satisfying assignment could be found for the variable break set or not.

benchmark	sat	balance	$ V_{bs} $	L	R	# ps	sat $V_{bs}$
1000144.col	U	0.26	189	771	231	0	no
pyhala-braun 03	U	0.28	71	4788	1810	$\geq 0$	yes
connamacher 975	U	0.26	118	76	374	0	no
depots 3948	S	0.42	58	141	65	1	yes
driverlog 3963	S	0.30	18	92	48	1	yes
eulcbip 3936	-	0.25	24	22	88	$\geq 173$	yes
ferry 3996	S	0.38	21	405	255	1	yes
hanoi4	S	0.45	22	278	233	2	yes
stanion 1617	U	0.25	23	113	25	$\geq 1$	yes
ibm SAT_dat.k15	S	0.50	69	4869	4643	0	yes
linvrinv 565	-	0.25	57	538	161	$\geq 0$	yes
longmult8	U	0.40	26	1341	932	1	yes
philips	U	0.27	34	1022	335	1	yes
pmg 3939	U	0.26	19	115	34	97797	yes
qg3-09	U	0.50	296	79	0	0	no
rovers 3978	S	0.44	28	38	66	0	yes
anton 930	U	0.25	337	74	535	0	no
satellite 3985	S	0.46	50	21	87	0	yes
unsat350 20	U	0.25	174	2	172	0	no
vdw_5_3_50	S	0.29	85	0	160	0	yes

**Table 6.5:** connamacher 975: Solve times in seconds using the FL500 partition switch modification of march\_dl for five runs of eight different balanced approximated minimal cuts using hMeTiS. The actual balance, the size of the break set, the number of partition swaps and whether a satisfying assignment for the variable break set could be found or not are given as well. connamacher 975 is solved in 1252.61 seconds using the unmodified march\_dl solver.

balance category	actual balance	$ V_{bs} $	FLAps 500	# ps	sat $V_{bs}$
0.10 – 0.90	0.10	68	> 2000	$\geq 0$	yes
	0.10	72	> 2000	$\geq 0$	yes
	0.10	68	> 2000	$\geq 0$	yes
	0.10	68	1713.26	0	yes
	0.10	66	> 2000	$\geq 0$	yes
0.15 – 0.85	0.15	88	1360.07	0	no
	0.15	88	> 2000	$\geq 0$	no
	0.15	88	1492.19	0	no
	0.15	92	867.23	0	no
	0.15	88	> 2000	$\geq 0$	yes
0.20 – 0.80	0.21	104	1088.59	0	no
	0.21	108	929.78	0	no
	0.20	100	1065.95	0	no
	0.22	108	> 2000	$\geq 0$	no
	0.23	112	1648.36	0	no
0.25 – 0.75	0.27	120	1266.69	0	no
	0.26	124	875.90	0	no
	0.25	118	445.90	0	no
	0.26	120	1387.18	0	no
	0.26	120	656.04	0	no
0.30 – 0.70	0.34	128	726.72	0	no
	0.32	128	> 2000	$\geq 0$	no
	0.31	128	> 2000	$\geq 0$	no
	0.30	126	954.75	0	no
	0.31	132	1795.23	0	no
0.35 – 0.65	0.36	136	> 2000	$\geq 0$	no
	0.36	136	1053.35	0	no
	0.39	136	730.31	0	no
	0.36	133	> 2000	$\geq 0$	no
	0.39	136	> 2000	$\geq 0$	no

Table 6.5: Continued

balance category	actual balance	$ V_{bs} $	FLAps 500	# ps	sat $V_{bs}$
0.40 – 0.60	0.41	140	> 2000	$\geq 0$	no
	0.50	140	> 2000	$\geq 0$	no
	0.41	138	> 2000	$\geq 0$	no
	0.41	140	393.96	0	no
	0.41	136	> 2000	$\geq 0$	no
0.45 – 0.55	0.50	140	> 2000	$\geq 0$	no
	0.48	140	> 2000	$\geq 0$	no
	0.50	140	> 2000	$\geq 0$	no
	0.47	144	1724.01	0	no
	0.50	140	> 2000	$\geq 0$	no

**Table 6.6:** 1000144.col: Solve times in seconds using the FLAps 500 modification of `march_dl` for two runs of nine different balanced approximated minimal cuts using `hMeTiS`. The actual balance, the size of the break set, the number of partition swaps and whether a satisfying assignment for the variable break set could be found or not are given as well. 1000144.col is solved in 1.75 seconds using the unmodified `march_dl` solver.

balance category	actual balance	$ V_{bs} $	FLAps 500	# ps	sat $V_{bs}$
0.05 – 0.95	0.05	51	9.18	0	no
	0.05	49	11.80	0	no
0.10 – 0.90	0.10	96	6.54	0	no
	0.10	93	7.28	0	no
0.15 – 0.85	0.16	132	6.61	0	no
	0.15	129	5.27	0	no
0.20 – 0.80	0.20	155	12.81	0	no
	0.21	162	9.65	0	no
0.25 – 0.75	0.26	184	5.80	0	no
	0.26	191	3.80	0	no
0.30 – 0.70	0.31	210	3.61	0	no
	0.32	207	4.19	0	no
0.35 – 0.65	0.38	228	4.44	0	no
	0.38	231	4.44	0	no
0.40 – 0.60	0.47	240	6.36	0	no
	0.46	235	6.60	0	no
0.45 – 0.55	0.50	243	5.80	0	no
	0.50	250	4.16	0	no

Three of the five balance categories for `chg(stanion 1617)` for the minimal cut for a bipartition with `hMeTiS` resulted in a fifty-fifty bipartition. Using the unmodified `march_dl` solver `stanion 1617` is solved in 936.82 seconds. Executing the CNF with the additional partition information resulted in:

**Table 6.7:** `stanion 1617`: Solve times in seconds using the FLAps 500 modification of `march_dl` for five different balanced approximated minimal cuts using `hMeTiS`. The actual balance, the size of the break set, the number of partition swaps and whether a satisfying assignment for the variable break set could be found or not are given as well. `stanion 1617` is solved in 936.82 seconds using the unmodified `march_dl` solver.

balance category	actual balance	$ V_{bs} $	FLAps 500	# ps	sat $V_{bs}$
0.25 – 0.75	0.26	23	43.62	0	yes
0.30 – 0.70	0.30	25	255.85	0	yes
0.35 – 0.65	0.50	25	421.44	0	yes
0.40 – 0.60	0.50	25	425.51	0	yes
0.45 – 0.55	0.50	25	409.64	0	yes

**Table 6.8:** `pmg 3939`: Solve times in seconds using the FLAps 500 modification of `march_dl` for five different balanced approximated minimal cuts using `hMeTiS`. The actual balance, the size of the break set, the number of partition swaps and whether a satisfying assignment for the variable break set could be found or not are given as well. `pmg 3939` is solved in 502.37 seconds using the unmodified `march_dl` solver.

balance category	actual balance	$ V_{bs} $	FLAps 500	# ps	sat $V_{bs}$
0.25 – 0.75	0.26	19	586.65	0	yes
0.30 – 0.70	0.33	21	98.59	0	yes
0.35 – 0.65	0.36	22	354.30	0	yes
0.40 – 0.60	0.40	23	436.64	0	yes
0.45 – 0.55	0.46	23	610.00	0	yes

**Table 6.9:** philips: Solve times in seconds using the FLAps 500 modification of `march_dl` for five different balanced approximated minimal cuts using `hMeTiS`. The actual balance, the size of the break set, the number of partition swaps and whether a satisfying assignment for the variable break set could be found or not are given as well. `philips` is solved in 1438.46 seconds using the unmodified `march_dl` solver.

balance category	actual balance	$ V_{bs} $	FLAps 500	# ps	sat $V_{bs}$
0.25 – 0.75	0.27	34	> 2000	$\geq 0$	yes
0.30 – 0.70	0.31	36	> 2000	$\geq 0$	yes
0.35 – 0.65	0.35	38	> 2000	$\geq 0$	yes
0.40 – 0.60	0.41	39	1093.19	0	yes
0.45 – 0.55	0.46	45	1498.73	0	yes

**Table 6.10:** anton 930: Solve times in seconds using the FLAps 500 modification of `march_dl` for nine different balanced approximated minimal cuts using `hMeTiS`. The actual balance, the size of the break set, the number of partition swaps and whether a satisfying assignment for the variable break set could be found or not are given as well. `anton 930` is solved in 90.62 seconds using the unmodified `march_dl` solver.

balance category	actual balance	$ V_{bs} $	FLAps 500	# ps	sat $V_{bs}$
0.05 – 0.95	0.05	110	> 2000	$\geq 0$	yes
0.10 – 0.90	0.10	190	344.18	0	no
0.15 – 0.85	0.15	248	212.78	0	no
0.20 – 0.80	0.20	295	138.45	0	no
0.25 – 0.75	0.25	330	> 131.70	0	no
0.30 – 0.70	0.30	360	152.26	0	no
0.35 – 0.65	0.36	389	128.15	0	no
0.40 – 0.60	0.46	412	122.58	0	no
0.45 – 0.55	0.50	428	116.57	0	no

# Chapter 7

## Conclusions & Future work

We have seen how to represent a SAT problem as a graph and which requirements it needs to fulfil to be a decomposable graph representation. Multiple connected components in the representation graph should match with a decomposed SAT problem.

In order to reduce the graph size and increase the chance of the graph falling appart redundant clauses can be removed from the CNF. Although obtaining a subgraph is preferred in the context of the representation graph, removing blocked clauses is not always desirable in the context of the direct solve time.

Practical SAT problems rarely naturally fall apart in connected components in the representation graph, the same holds for biconnected components. Using the concept of tree decomposition one can get a coarse representation of the graph and capture the locality of the internal connectivity. The diameter can also give a good indication of the locality of the representation graph and besides, in contrast with the treewidth, it can be calculated efficiently.

More research on the use of the diameter property to create a kind of splitting set should be performed. Is the linear classification of SAT instances on their fastest solver by their diameter and density just a coincidence for the test-set used, or will it stil hold for a more representative set of larger instances with a wider range in diameter sizes?

Chapter 5 showed that tree decompositions of representation graphs can be used to efficiently visualize specific SAT problems. The resulting graphs are relative small and therefore good surveyable. Visualizing the SAT instances can be used as a study tool to get a better insight into the problem.

Representation graphs rarely naturally consist of multiple connected com-

ponets. Using heuristic minimal cut algorithms one can quite easily find a balanced approximated minimal cut of the representation graph. A problem of the hypergraph partitioning tool **hMeTiS** is, that it is non-deterministic. Running the program twice with exactly the same arguments might result in different outputs. This made testing very difficult.

Some of the test-set instances had one or more variables being represented by multiple hyperedges in the **rhg**. By definition the **rhg** and **srhg** have better splitting possibilities than the **chg**. What type of variable forms distinct cliques of clauses in such hypergraphs? Can they be assigned a specific role in the problem encoding?

The branching strategy made it possible to use a variable break-set to actually decompose the SAT problem. This strategy can be easily extended to be used for multiple partitions and even for multilevel partitioning. In multilevel partitioning a variable break-set can be constructed again for a single partition to decompose it into multiple partitions.

In the first instance the results of using a variable break-set to solve the test-set instances, like presented in table 6.3, did not result in a very consistent whole. Also the overall improvement was not significant. A problem like **stanion 1617** could not be solved within 2000 seconds using partitioning branching. For **hanoi4** all different implementations using partitioning branching decreased the solve time. Of course the effect of decomposing the problem by a break-set may very well depend on the type of the problem. The effect of the different implementations for applying look-ahead on the variable break-set also varied for the different instances of the test-set.

Further research, however, raised hope again. Running multiple tests, also for different balanced partitions for the same instance showed quite some variation in the solve time. Although the solving time on some runs even exceeded the boundary of 2000 seconds, there were also runs having way shorter solving times. For the random or random like problems (as could be expected) no gain in solving time could be achieved in decomposing the problem (e.g., the results of **anton 930** in 6.10, **unsat350 20,1000144.col**). Yet, **connamacher 975**, normally solved with **march\_dl** in 1252.61 seconds, had a run with a solving time of 445.90 seconds. **stanion 1617**, normally solved with **march\_dl** in 936.82 seconds, even had a solving time of 43.62 seconds. Obviously the solve time depends very much on the exact variable break-set chosen. It is not only size that matters. The question now remains: What makes one break-set, except from just size, so much better than other ones? It is to be expected that the answer lies in the different roles of the variables. Further research is needed on finding *effective* variable break-sets to decompose SAT problems.

# Bibliography

- [1] C.J. Alpert and A.B. Khang. *Recent directions in netlist partitioning*. Integration, the VLSI Journal, vol. 19(1-2), 1-81, 1995.
- [2] D. Le Berre and L. Simon. *The essentials of the SAT'03 Competition*. In Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing, SAT 2003, Lecture Notes in Comp. Sci., Springer, vol. 2919, 452–467, 2004.
- [3] D. Le Berre and L. Simon. *Fifty-five solvers in Vancouver: The sat 2004 competition*. In Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing, SAT 2004, Lecture Notes in Comp. Sci., Springer, vol. 3542, 321–344, 2005.
- [4] D. Le Berre and L. Simon. *Special Volume on the SAT 2005 Competitions and Evaluations*. Journal on Satisfiability, Boolean Modeling and Computation, vol. 2, 2006.
- [5] A. Biere, A. Cimatti, E.M. Clarke and Y. Zhu. *Symbolic model checking without BDDs*. In Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Lecture Notes in Comp. Sci., Springer Verlag, vol. 1579, 193–207, 1999.
- [6] Per Bjesse, James Kukula, Robert Damiano, Ted Stanion and Yunshan Zhu. *Guiding SAT Diagnosis with Tree Decompositions*. In Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing, SAT 2003, Lecture Notes in Comp. Sci., Springer, vol. 2919, 315–329, 2004.
- [7] Hans Bodlaender. *Discovering Treewidth*. SOFSEM 2005: Theory and Practice of Computer Science, Lecture Notes in Comp. Sci., Springer Verlag, vol. 3381, 1–16, 2005.

- [8] Harold Connamacher. *A Random Constraint Satisfaction Problem That Seems Hard for DPLL*. The Seventh International Conference on Theory and Applications of Satisfiability Testing, Online Proceedings, 2004.
- [9] H. Connamacher and M. Molloy. *The exact satisfiability threshold for a potentially intractable random constraint satisfaction problem*. In Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science, 590–599, 2004.
- [10] S. Cook. *The Complexity of Theorem Proving Procedures*. In Proceedings of Third Annual ACM Symposium on Theory of Computing, 151–158, 1971.
- [11] Vijay Durairaj and Priyank Kalla. *Exploiting Hypergraph Partitioning for Efficient Boolean Satisfiability*. In High-Level Design Validation and Test Workshop, 2004. Ninth IEEE International, 141–146, 2004.
- [12] Niklas Eén and Armin Biere. *Effective Preprocessing in SAT Through Variable and Clause Elimination*. In Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing, SAT 2005, Lecture Notes in Comp. Sci., Springer, vol. 3569, 61–75, 2005.
- [13] Niklas Eén and Niklas Sörensson. *Translating Pseudo-Boolean Constraints into SAT*. Journal on Satisfiability, Boolean Modeling and Computation, vol. 2, 1–26, 2006.
- [14] C.M. Fiduccia and R.M. Mattheyses. *A linear time heuristic for improving network partitions*. In Proceedings of the 19th IEEE Design Automation Conf., 175–181, 1982.
- [15] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: W.H. Freeman and Company, 1979.
- [16] Fanica Gavril. *The intersection graphs of subtrees in trees are exactly the chordal graphs*. Journal of Combinatorial Theory, Ser. B, 16:47–56, 1974.
- [17] Alfonso Gerevini and Ivan Serina. *Planning as Propositional CSP: From Walksat to Local Search Techniques for Action Graphs*. Journal on Constraints, Springer, vol. 8(4), 389–413, 2003.
- [18] V. Gogate and R. Dechter. *A complete anytime algorithm for treewidth*. In proceedings of the 20th Annual Conference on Uncertainty in Artificial Intelligence, 201–208, 2004.

- [19] P.R. Herwig, M.J.H. Heule, P.M. Van Lambalgen and H. Van Maaren. *A new method to construct lower bounds for Van der Waerden numbers*. Submitted to The Electronic Journal of Combinatorics, 2005.
- [20] M.J.H. Heule. *March: Towards a lookahead Sat solver for general purposes*. Master's thesis, Delft University of Technology, 2003.
- [21] Marijn Heule and Oliver Kullmann. *Decomposing clause-sets: Integrating DLL algorithms, tree decompositions and hypergraph cuts for variable- and clause-based graph representations of CNF's*. Computer Science Report Series, University of Wales Swansea, CSR 2-2006, 2006.
- [22] Marijn J.H. Heule and Hans van Maaren. *March dl: Adding Adaptive Heuristics and a New Branching Strategy*. Journal on Satisfiability, Boolean Modeling and Computation, 2:47–59, 2006.
- [23] Ilya V. Hicks, Arie M.C.A. Koster and Elif Kolotoğlu. *Branch and tree decomposition techniques for discrete optimization*. In TutORials 2005, INFORMS TutORials in operations Research, Series B, 1–29, 2005.
- [24] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. *Multilevel hypergraph partitioning: Application in vlsi domain*. In Proceedings of the 34th Conference on Design Automation, ACM, 526–259, 1997.
- [25] Henry Kautz and Bart Selman. *Planning as Satisfiability*. In Proceedings of the European Conference on Artificial Intelligence (ECAI), 359–363, 1992.
- [26] B.W. Kernighan and S. Lin. *An efficient heuristic procedure for partitioning graphs*. Bell Syst. Tech. J., vol. 49(2), 291-307, 1970.
- [27] Arie M. C. A. Koster, Hans L. Bodlaender and Stan P. M. van Hoesel. *Treewidth: Computational Experiments*. Tech. rep., KonradZuse - Zentrum für Informationstechnik Berlin, 2001.
- [28] A.M.C.A. Koster, S.P.M. van Hoesel, and A.W.J. Kolen. *Solving partial constraint satisfaction problems with tree decomposition*. Networks, 40:170–180, 2002.
- [29] Oliver Kullman. *Obere und untere Schranken für die Komplexität von aussagenlogischen Resolutionsbeweisen und Klassen von SAT-Algorithmen*. Master's thesis, Johann Wolfgang Goethe-Universität Frankfurt am Main, 1992.

- [30] Oliver Kullmann. *On a generalization of extended resolution*. Discrete Applied Mathematics, 96-97(1-3):149–176, 1999.
- [31] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman and L. Troyansky. *Determining Computational Complexity from Characteristic ‘Phase Transitions’*. Nature, 400:133–137, 1999.
- [32] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang and S. Malik. *Chaff: Engineering an Efficient SAT Solver*. In Proceedings of the 38th Conference on Design Automation, 530–535, 2001.
- [33] N. Robertson and P. D. Seymour. Graph minors II. *algorithmic aspects of tree-width*. Journal of Algorithms, 7:309–322, 1986.
- [34] L. Simon, D. Le Berre, and E. Hirsch. *The SAT 2002 competition*. Annals of Mathematics and Artificial Intelligence (AMAI) 43:343-378, 2005.
- [35] Carsten Sinz. *Visualizing the Internal Structure of SAT Instances*. The Seventh International Conference on Theory and Applications of Satisfiability Testing, Online Proceedings, 2004.
- [36] Carsten Sinz and Edda-Maria Dieringer. *DPvis - A Tool to Visualize the Structure of SAT Instances*. In Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing, SAT 2005, Lecture Notes in Comp. Sci., Springer, vol. 3569, 257–268, 2005.
- [37] A. Slater. *Visualisation of satisfiability problems using connected graphs*. [http://rsise.anu.edu.au/~sim\\$andrews/problem2graph](http://rsise.anu.edu.au/~sim$andrews/problem2graph), 2004.
- [38] Ryan Williams, Carla P. Gomes and Bart Selman. *Backdoors To Typical Case Complexity*. In Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), 1173–1178, 2003.
- [39] E. Zarpas. *Simple yet efficient improvements of SAT based Bounded Model Checking*. In Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD), Lecture Notes in Comp. Sci., Springer, vol. 3312, 174–185, 2004.
- [40] H. Zhang and M.E. Stickel. *Implementing the Davis-Putnam Method*. Journal of Automated Reasoning, vol. 24(1-2), 277–296, 2000.
- [41] Lintao Zhang. *On Subsumption Removal and On-the-Fly CNF Simplification*. In Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing, SAT 2005, Lecture Notes in Comp. Sci., Springer, vol. 3569, 482–489, 2005.

# Appendix A

## Test-set

The CNF benchmarks used for testing are given in table [A.1](#). The set consists of handmade, industrial and random SAT problems.

① 1000144.col is part of the 3color family. Instances of the 3color family are SAT translations of the graph 3 coloring problem. Is there a coloring for a graph using 3 colors such that connected vertices always have different colors?

② pyhala-braun instances were part of the SAT 2002 competition [34]. The pyhala family was submitted by Tuomo Pyhala as a generator. Depending on arguments, it can generate a SAT encoding of factoring of primes (unsatisfiable instances) or products of two primes (satisfiable instances). The benchmarks encode multiplication circuits with predefined output. Two circuits are available (braun or adder-tree multipliers). The instance used is pyhala-braun-unsat-35-4-03, an unsatisfiable instance using braun circuits.

③ connamacher 975 was contributed by Connamacher [8] to the SAT 2004 competition [3]. This family consists of encodings of the generic uniquely extendible constraint satisfaction problem. The instance used is connm-ue-csp-sat-n600-d0.04-s1793042357.cnf.

④ depots 3948 is an industrial planning problem contributed by Frédéric Maris to the SAT 2005 competition [4]. The original name of the instance was depots2\_v01a.cnf.

⑤ driverlog 3963 is an industrial planning problem contributed by Frédéric Maris to the SAT 2005 competition [4]. The original name of the instance was driverlog3\_v01a.cnf.

**Table A.1:** The test set

	<b>benchmark</b>	<b>sat</b>	<b>vars</b>	<b>cls</b>
①	1000144.col	U	1200	3154
②	pyhala-braun 03	U	7383	24320
③	connamacher 975	U	568	7150
④	depots 3948	S	290	3587
⑤	driverlog 3963	S	170	1595
⑥	eulcbip 3936	-	135	672
⑦	ferry 3996	S	701	6661
⑧	hanoi4	S	718	4934
⑨	stanion 1617	U	162	774
⑩	ibm SAT_dat.k15	S	13818	69795
⑪	linvrvn 565	-	792	2557
⑫	longmult8	U	3810	11877
⑬	philips	U	3642	4456
⑭	pmg 3939	U	169	562
⑮	qg3-09	U	729	28463
⑯	rovers 3987	S	229	3697
⑰	anton 930	U	1000	3730
⑱	satellite 3985	S	158	3086
⑲	unsat350 20	U	350	1493
⑳	vdw_5_3_50	S	250	2953

⑥ eulcbip 3936 was contributed by Klas Markström to the SAT 2005 competition [4]. eulcbip is a family based on Eulerian graphs which are aimed at being hard for resolution based SAT-solvers. The original name of the instance was eulcbip-7-UNSAT.cnf.

⑦ ferry 3996 is an industrial planning problem contributed by Frédéric Maris to the SAT 2005 competition [4]. The original name of the instance was ferry6\_ks99a.cnff.

- ⑧ **hanoi4** is a SAT translation of the 4 discs version of the Tower of Hanoi problem by Kautz and Selman[25]. The 4 and 5 discs SAT versions are part of the DIMACS Benchmark Instances which can be found at <http://www.satlib.org>.
- ⑨ **stanion 1617** is a crafted problem used in the SAT 2003 competition [2] and was submitted by Ted Stanion (Synopsis Inc.). The original name of the instance was `hwb-n24-01-S1048418025.cnf`.
- ⑩ **ibm SAT\_dat.k15** is an instance generated through bounded model checking from the IBM Formal Verification Benchmark Library [39] and was submitted by Emmanuel Zarpas to the SAT 2002 competition [34]. The instance is part of the `IBM_FV_2002_01_rule` set.
- ⑪ **linvrinv 565** is a crafted benchmark and was submitted by Armin Biere to the SAT 2005 competition [4]. The original name of the instance was `linvrinv6.cnf`.
- ⑫ **longmult8** is contributed by Armin Biere. Instances from the `longmult` family arise from bounded model checking [5].
- ⑬ **philips** contributed by Heule to the SAT 2004 competition [3]. Encoding of a multiplier circuit provided by Philips.
- ⑭ **pmg 3939** is a crafted benchmark contributed by Klas Markström to the SAT 2005 competition [4]. The original name of the instance was `pmg-11-UNSAT.cnf`.
- ⑮ **qg3-09** is a SAT translation of the Quasigroup, or Latin Square, problem. Zhang and Stickel used these instances in [40]. The instance was part of the SAT 2002 competition [34] in the category of handmade problems. More information can be found at <http://www.satlib.org>.
- ⑯ **rovers 3987** is an industrial planning problem contributed by Frédéric Maris to the SAT 2005 competition [4]. The original name of the instance was `rovers2_v01i.cnf`.
- ⑰ **anton 930** was contributed by Anton to the SAT 2004 competition [3].

This is a random  $l$ -clustered  $k$ -sat instance. The original name of the instance was `lksat-n1000-m3730-k3-l5-s1682954997.cnf`.

⑱ `satellite 3985` is an industrial planning problem contributed by Frédéric Maris to the SAT 2005 competition [4]. The original name of the instance was `satellite1_v01a.cnf`.

⑲ `unsat350 20` is an unsatisfiable uniform random 3-sat formula generated by `mkcfnf` which can be obtained from <http://www.satlib.org>.

⑳ `vdw_5_3_50` is an encoding of the Van der Waerden certificate  $W(5, 4, 50)$  [19].

# Appendix B

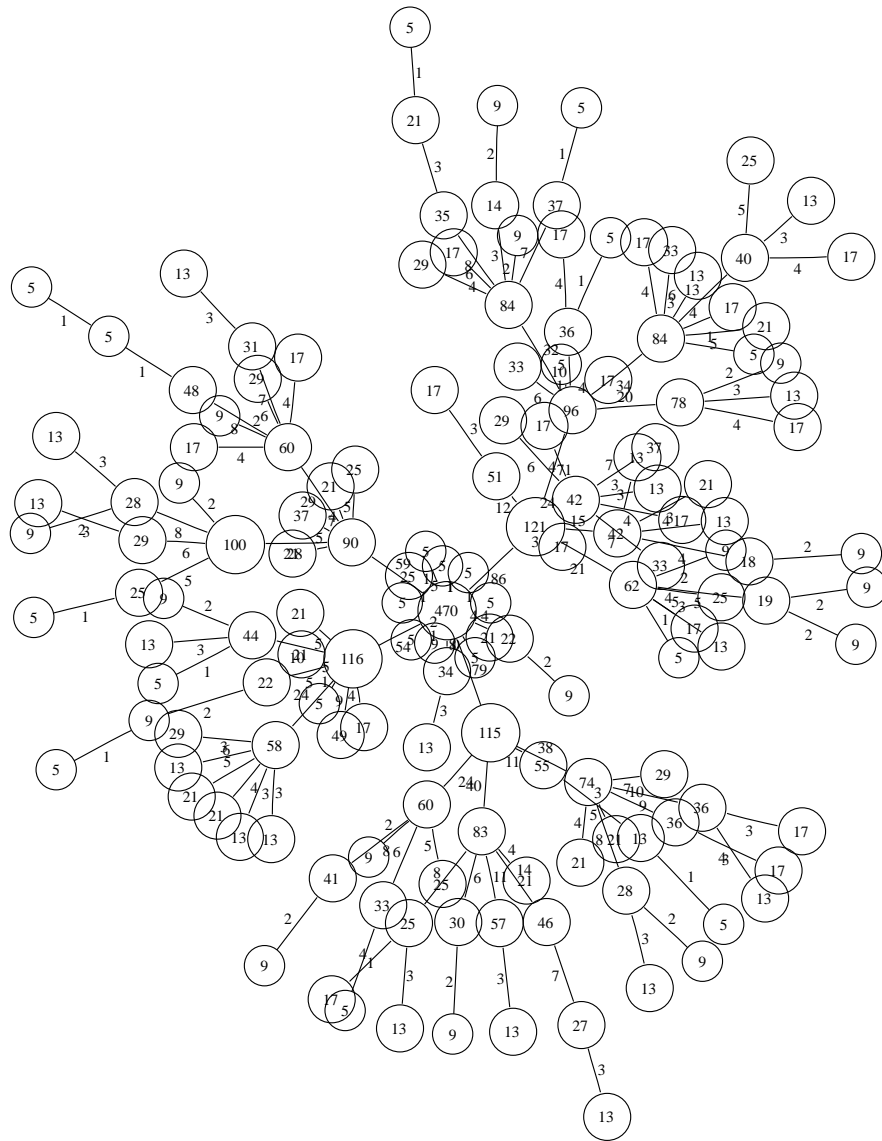
## Additional visualisations

This appendix shows geometric representations of 1-reduced tree decompositions of the `rg` of the test-set instances. The basic tree decompositions were created with the approximation program `mdfi` from Arie Koster [23]. `GraphViz`<sup>6</sup> is used to visualize the resulting tree decompositions. The following representations were created using a *by force-directed placement* layouter, `neato`.

- ② The `pyhala-braun 03` instance was too large for `mdfi` to generate a tree decomposition.
- ⑩ The `ibm SAT_dat.k15` instance was too large for `mdfi` to generate a tree decomposition.
- ⑮ The `qg3-09` instance was too large for `mdfi` to generate a tree decomposition.
- ⑰ The `anton 930` instance was too large for `mdfi` to generate a tree decomposition.

---

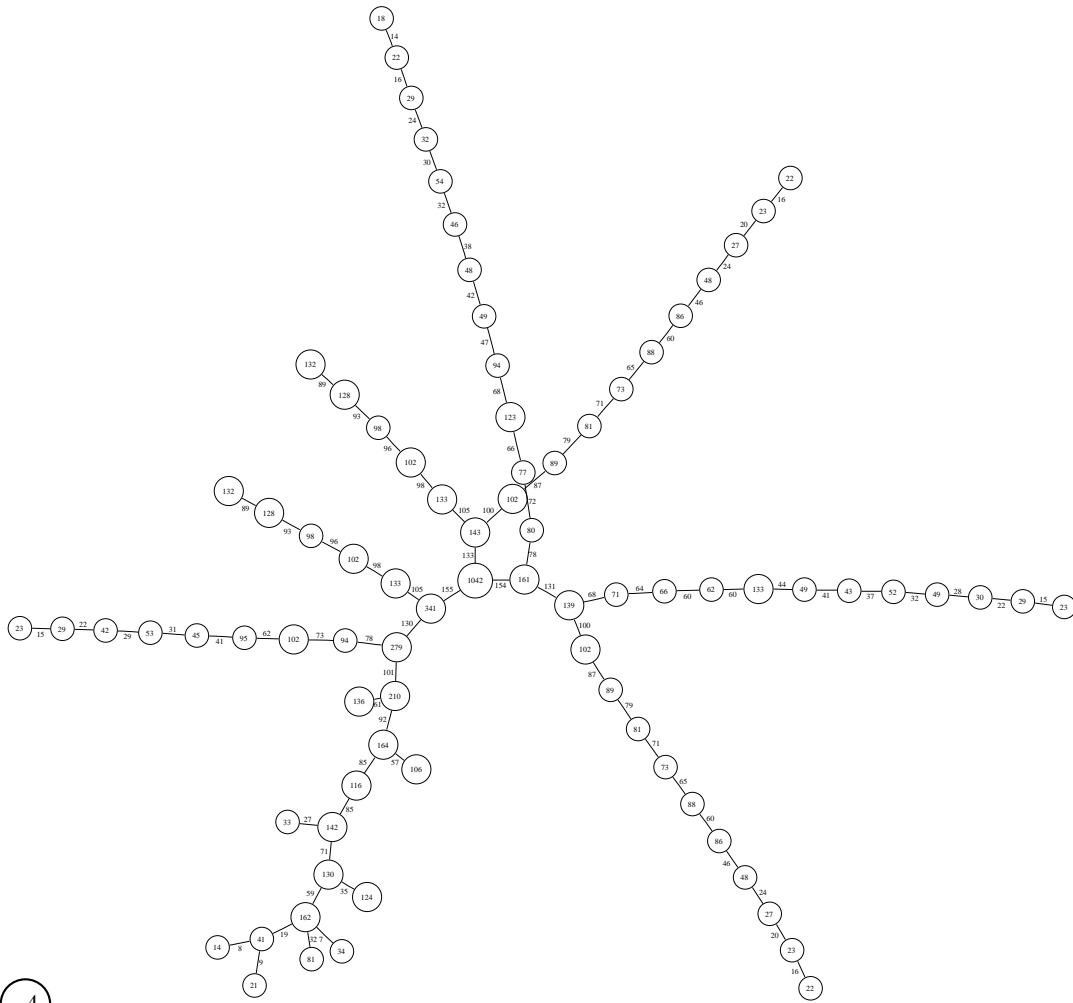
<sup>6</sup><http://www.research.att.com/sw/tools/graphviz>



1

**Figure B.1:** A geometric representation of a 1-reduced tree decomposition of  $rg(1000144.col)$ .

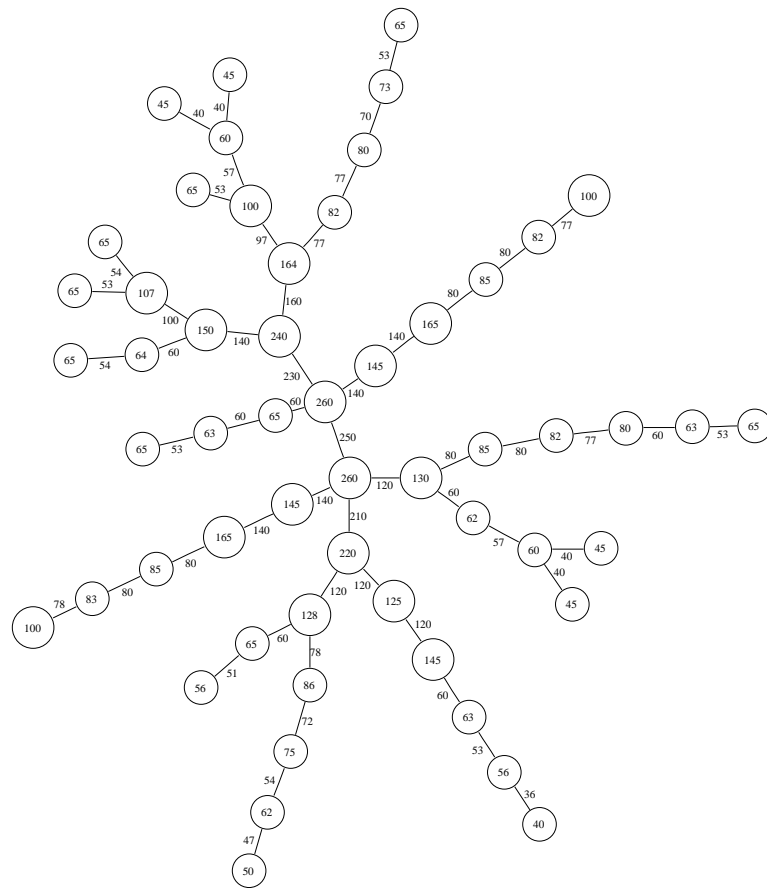




4

**Figure B.3:** A geometric representation of a 1-reduced tree decomposition of  $rg(\text{depots } 3948)$ .

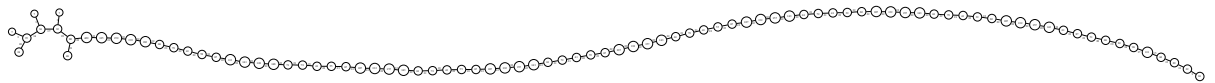




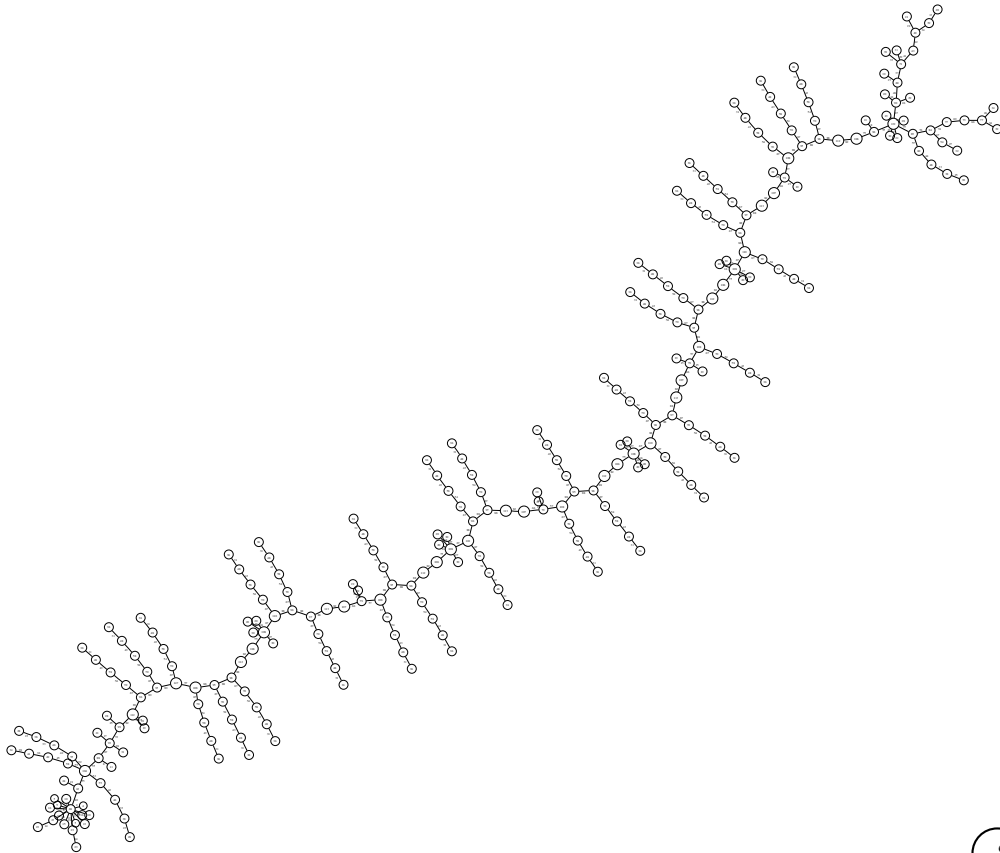
6

**Figure B.5:** A geometric representation of a 1-reduced tree decomposition of  $\text{rg}(\text{eulcbip } 3936)$ .

7

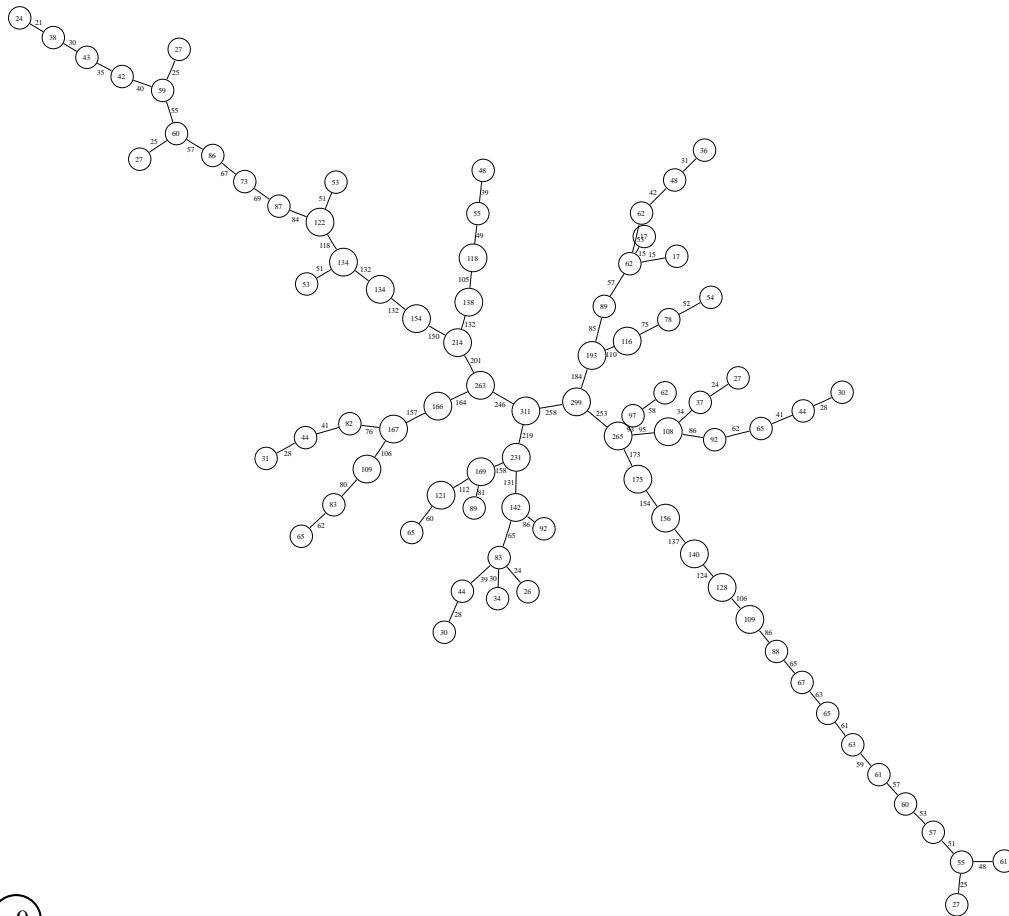


**Figure B.6:** A geometric representation of a 1-reduced tree decomposition of  $\text{rg}(\text{ferry } 3996)$ .



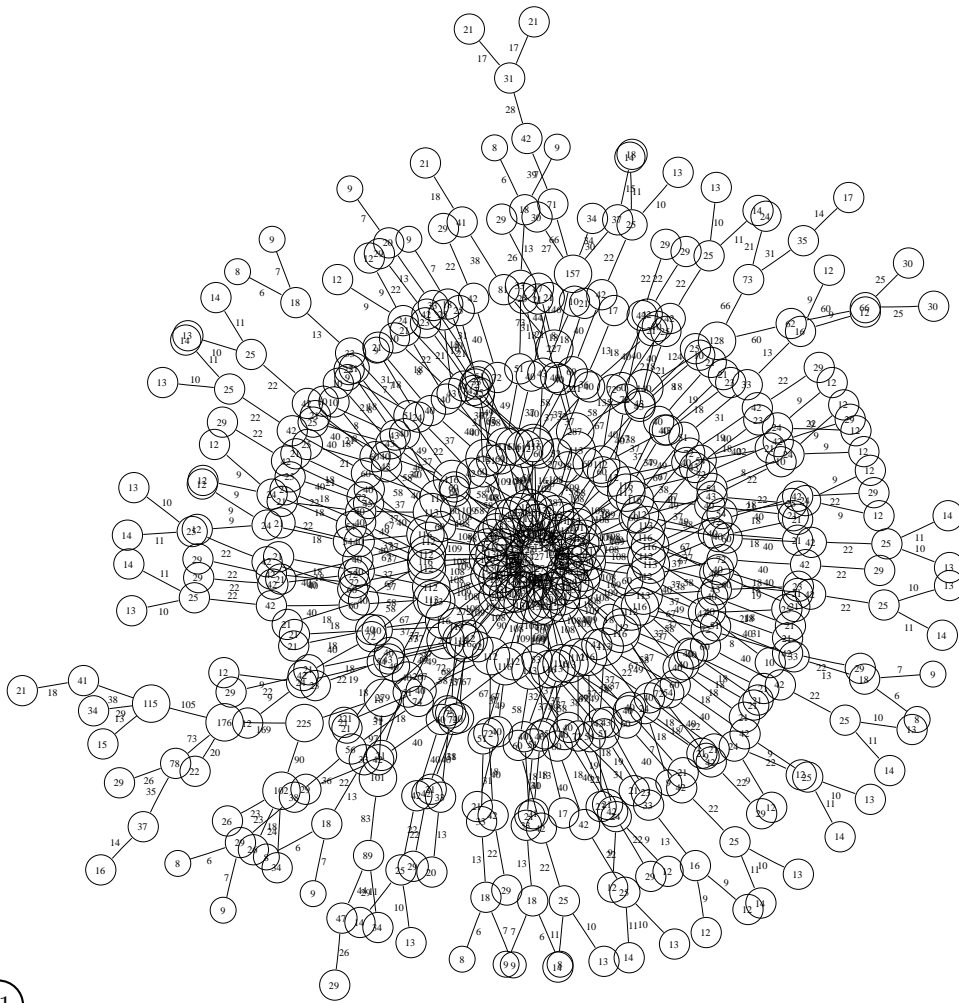
8

**Figure B.7:** A geometric representation of a 1-reduced tree decomposition of  $\text{rg}(\text{hanoi4})$ .



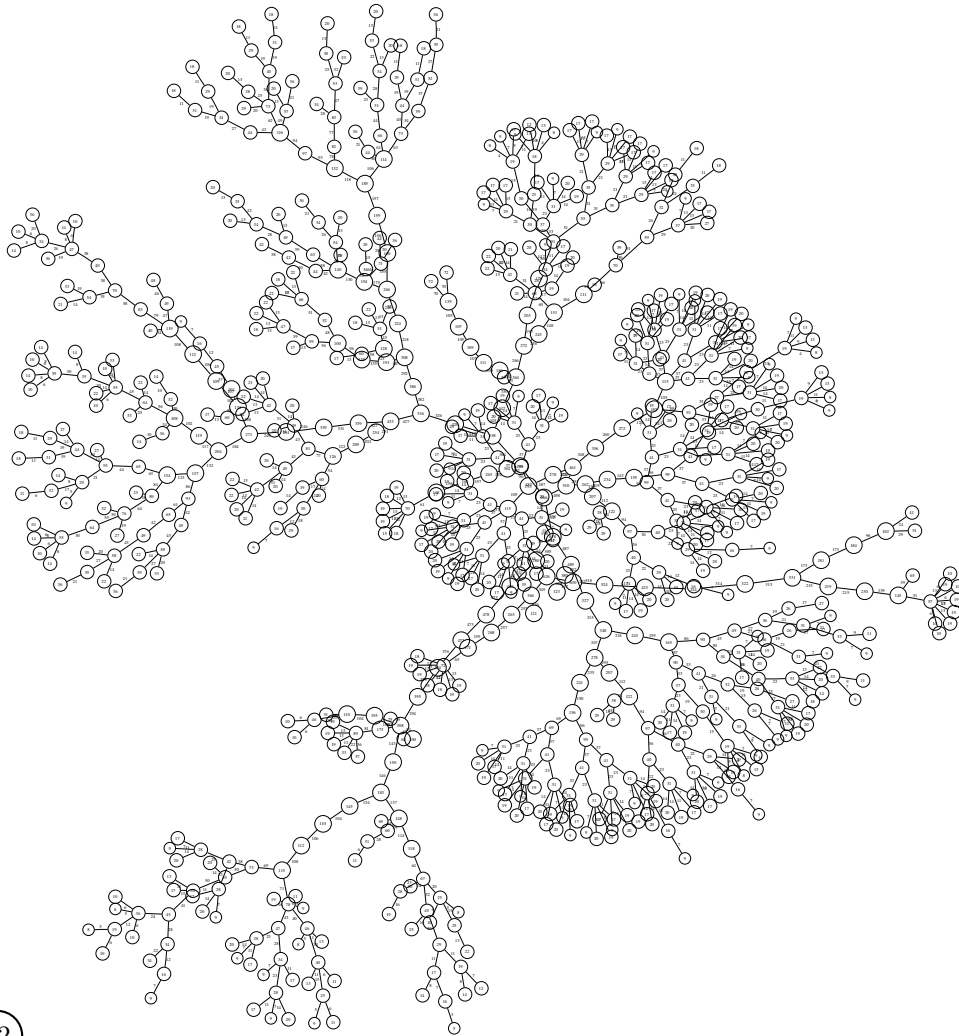
9

**Figure B.8:** A geometric representation of a 1-reduced tree decomposition of  $\text{rg}(\text{stanion } 1617)$ .



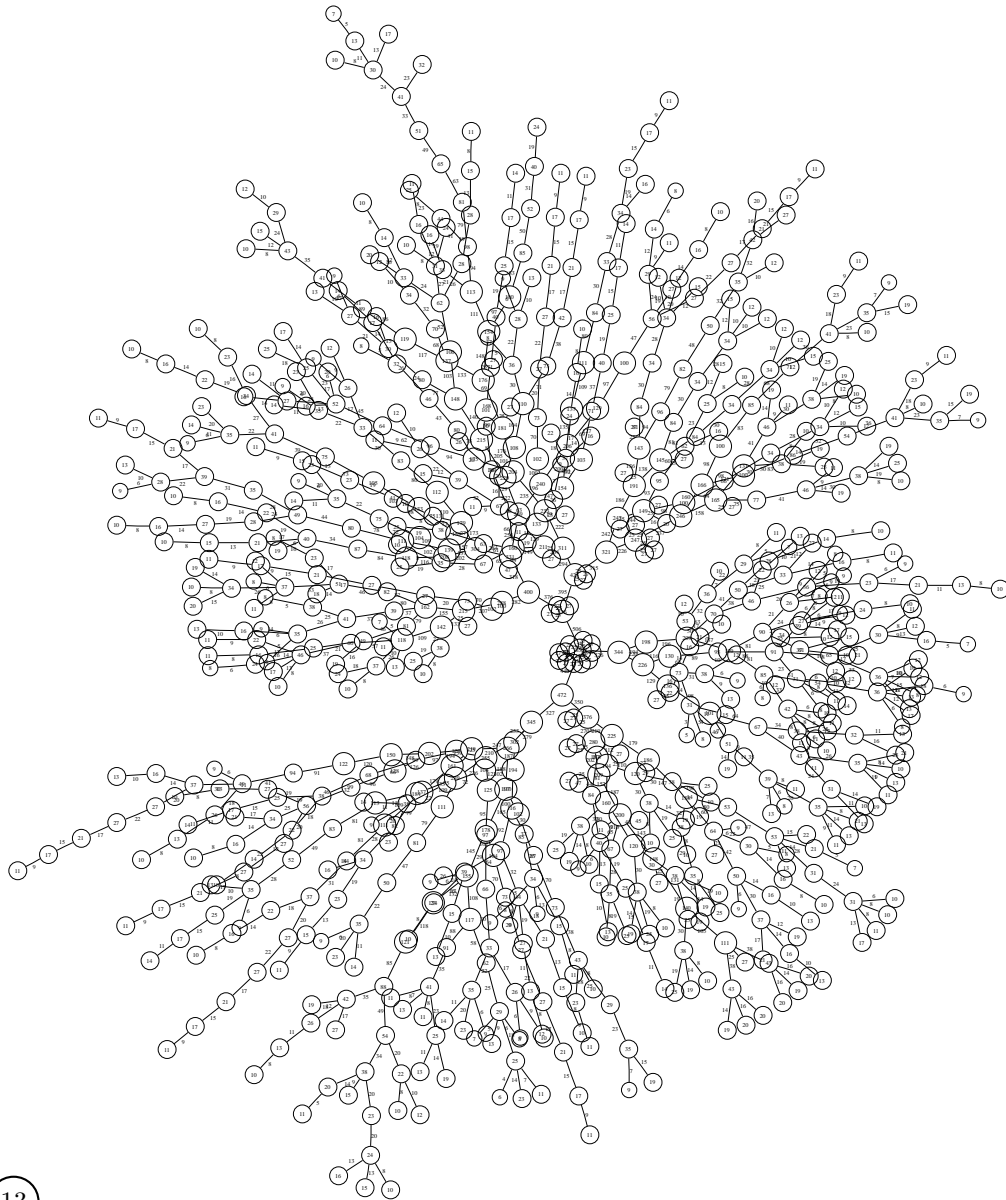
11

**Figure B.9:** A geometric representation of a 1-reduced tree decomposition of  $rg(\text{linrvinr } 565)$ .



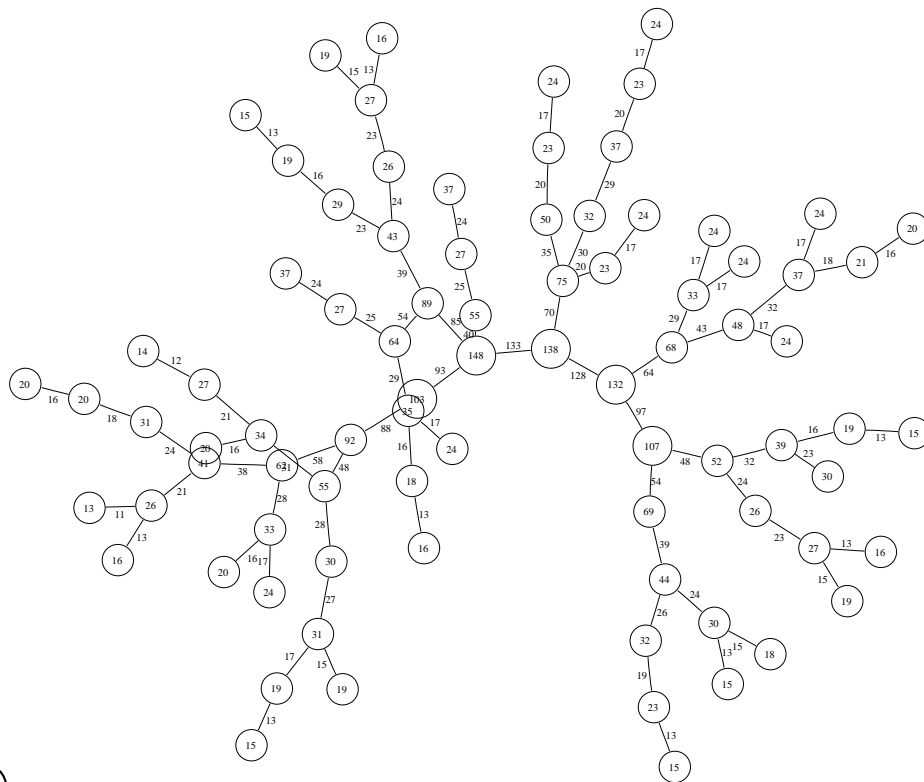
12

**Figure B.10:** A geometric representation of a 1-reduced tree decomposition of  $\text{rg}(\text{longmult8})$ .



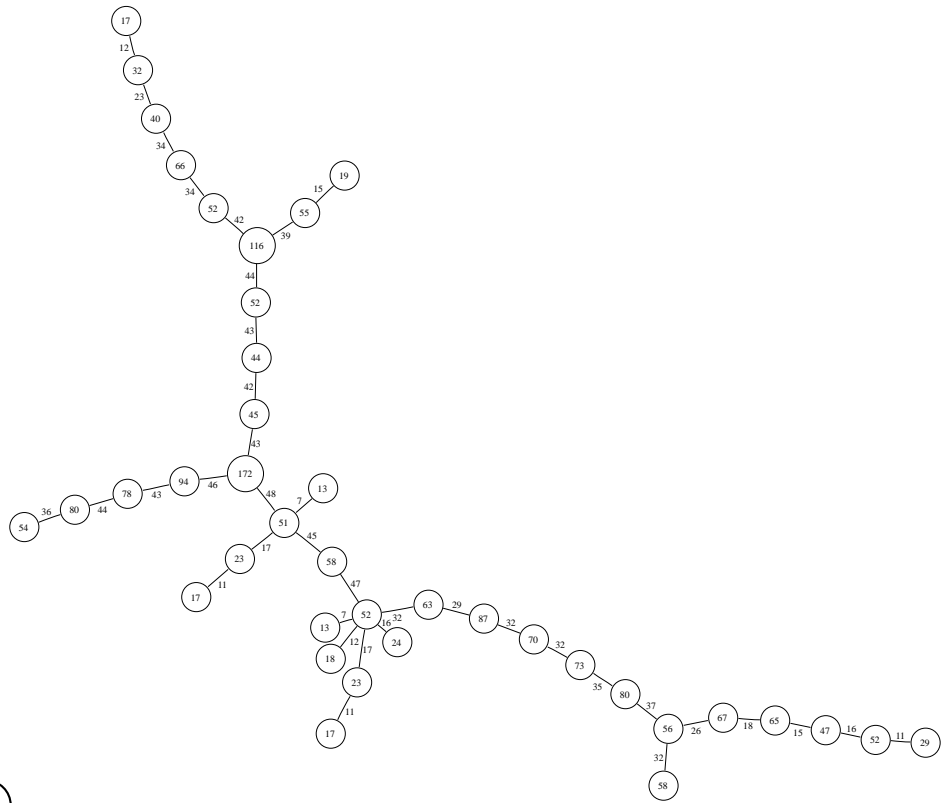
13

**Figure B.11:** A geometric representation of a 1-reduced tree decomposition of  $\text{rg}(\text{philips})$ .



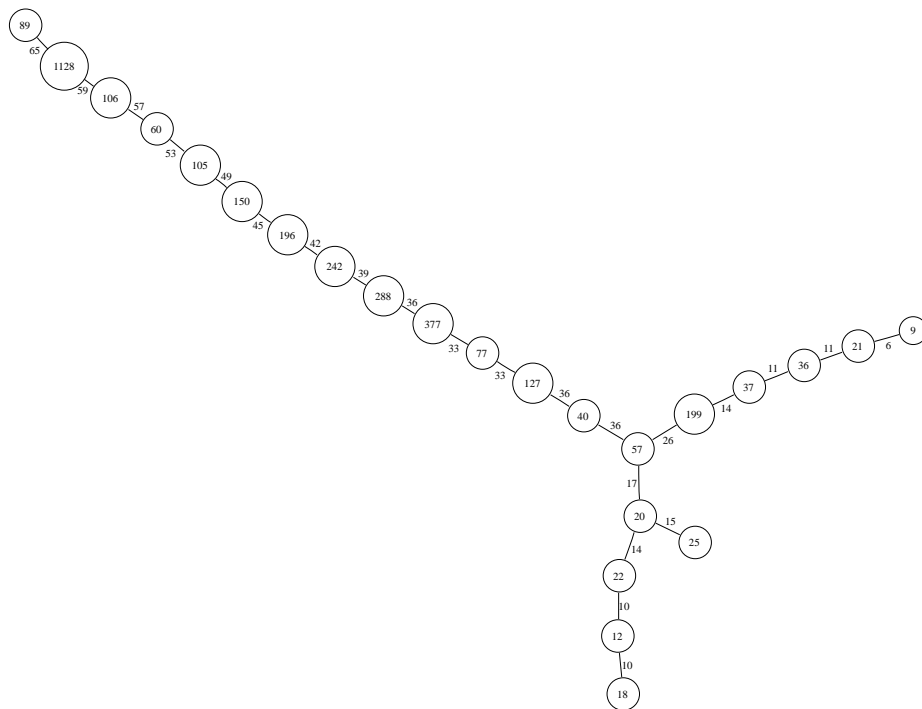
14

**Figure B.12:** A geometric representation of a 1-reduced tree decomposition of  $\text{rg}(\text{pmg } 3939)$ .



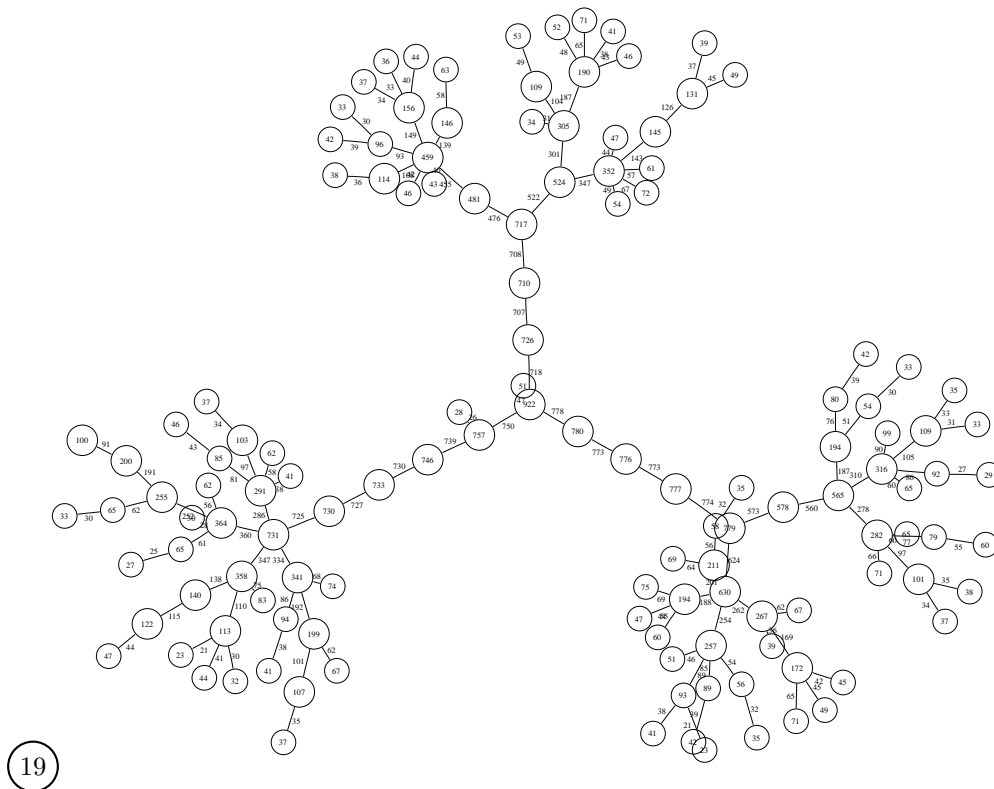
16

Figure B.13: A geometric representation of a 1-reduced tree decomposition of  $\text{rg}(\text{rovers } 3978)$ .



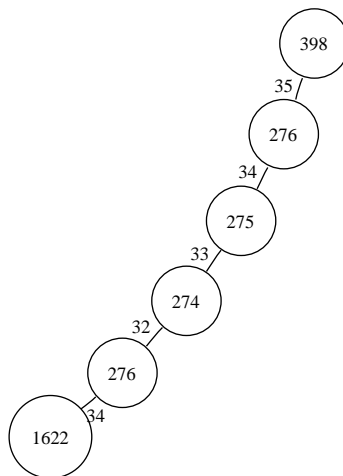
18

**Figure B.14:** A geometric representation of a 1-reduced tree decomposition of  $\text{rg}(\text{satellite } 3985)$ .



19

Figure B.15: A geometric representation of a 1-reduced tree decomposition of  $rg(\text{unsat350 } 20)$ .



20

Figure B.16: A geometric representation of a 1-reduced tree decomposition of  $rg(\text{vdw}_5_3_50)$ .

# Appendix C

## Proofs

A hypergraph  $H$  is a pair  $(V, E)$  where  $V$  is a set of elements, called vertices, and  $E$  is a set of subsets of elements of  $V$ , called hyperedges. Let  $V_1 = \{v_1, v_2, \dots, v_n\}$  and  $E = \{e_1, e_2, \dots, e_m\}$ . Every hypergraph has an  $n \times m$  incidence matrix  $A = a_{ij}$ , where

$$a_{ij} = \begin{cases} 1 & \text{if } v_i \in e_j \\ 0 & \text{otherwise} \end{cases}$$

The transpose  $A^t$  of the incidence matrix defines a hypergraph  $H^* = (V^*, E^*)$  called the dual of  $H$ , where  $V^*$  is an  $m$ -element set and  $E^*$  is an  $n$ -element set of subsets of  $V^*$  for  $v_j^* \in V^*$  and  $e_j^* \in E^*$ ,  $v_j^* \in e_j^*$  if and only if  $a_{ij} = 1$ .

A set of hyperedges  $E_c$  whose removal partitions the hypergraph  $H(V, E)$  into two disjoint subhypergraphs  $H_1(V_1, E_1)$  and  $H_2(V_2, E_2)$  is called a hyperedge cut. This can be represented in the incidence matrix as depicted in table C.1.

**Table C.1:** Simplified incidence matrix of two disjoint subhypergraphs  $H_1(V_1, E_1)$  and  $H_2(V_2, E_2)$  after removing a separating hyperedge set  $E_c$ .

	$E_1$	$E_c$	$E_2$
$V_1$	$H_1$		0
$V_2$	0		$H_2$

The subhypergraphs are disjoint, so for each vertex  $v_1 \in V_1$ ,  $v_1 \notin E_2$  and for each vertex  $v_2 \in V_2$ ,  $v_2 \notin E_1$  holds. This is represented by 0-matrices. The transpose of this matrix is showed in table C.2

**Table C.2:** Simplified transpose of incidence matrix of two disjoint hypergraphs  $H_1(V_1, E_1)$  and  $H_2(V_2, E_2)$ .

	$E_1^*$	$E_2^*$
$V_1^*$	$H_1^*$	0
$V_c^*$		
$V_2^*$	0	$H_2^*$

The subhypergraphs  $H_1^*(V_1^*, E_1^*)$  and  $H_2^*(V_2^*, E_2^*)$  are still disjoint, no hyperedge from  $E_1^*$  contains a vertex from  $V_2^*$  and visa versa.  $V_c^*$  represents a vertex-cut: removal of this set of vertices will partition the hypergraph.





