Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS

Tomáš Balyo

Solving Boolean Satisfiability Problems

Department of Theoretical Computer Science and Mathematical Logic

Supervisor: RNDr. Pavel Surynek, PhD.

Course of study: Theoretical Computer Science

2010

I wish to thank my supervisor RNDr. Pavel Surynek, PhD. He has generously given his time, talents and advice to assist me in the production of this thesis. I would also like to thank all the colleagues and friends, who supported my research and studies. Last but not least, I thank my loving family, without them nothing would be possible.

Contents

Název práce: Řešení problémů booleovské splnitelnosti

Autor: Tomáš Balyo

Katedra: Katedra teoreitické informatiky a matematické logiky

Vedoucí diplomové práce: RNDr. Pavel Surynek, PhD.

Email vedoucího: Pavel.Surynek@mff.cuni.cz

Abstrakt: V této práci studujeme možnosti rozkladu booleovských formulí do komponent souvislosti. Z tohoto důvodu zavádíme nový pojem - komponentový strom. Popisujeme některé vlastnosti komponentových stromů a možnosti jejich aplikace. Navrhli jsme třídu rozhodovacích heuristik pro SAT řešič na základě komponentových stromů a experimentálně zkoumali jejich výkon na testovacích SAT problémech. Pro tento účel jsme implementovali vlastní řešič, který využívá nejmodernější algoritmy a techniky pro řešení booleovské splnitelnosti.

Klíčová slova: Splnitelnost, rozhodovací heuristiky, komponentový strom

Title: Solving Boolean Satisfiability Problems

Author: Tomáš Balyo

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: RNDr. Pavel Surynek, PhD.

Supervisor's email: Pavel.Surynek@mff.cuni.cz

Abstract: In this thesis we study the possibilities of decomposing Boolean formulae into connected components. For this reason, we introduce a new concept - component trees. We describe some of their properties and suggest some applications. We designed a class of decision heuristics for SAT solvers based on component trees and experimentally examined their performance on benchmark problems. For this purpose we implemented our own solver, which uses the state-of-the-art SAT solving algorithms and techniques.

Keywords: Satisfiability, decision heuristics, component tree

Chapter 1

Introduction

Boolean satisfiability (SAT) is one of the most important problems of computer science. It was the first problem proven to be NP-complete[7]. The complexity class NP-complete (NPC) is a class of problems having two properties:

- A solution to the problem can be verified in polynomial time (Problems with this property are called NP problems)
- Each NP problem can be converted to the problem in polynomial time.

If any NPC problem can be solved in polynomial time then P=NP, where P is the complexity class of problems solvable in polynomial time. It is unknown whether P=NP, but many believe the answer is negative[11]. In that case it is impossible to construct a polynomial algorithm for SAT solving in the current computational model[1]. This would be most unfortunate, since we need to solve SAT problems in many practical applications. Some examples of these applications are hardware and software verification[28], planning[17] and automated reasoning[20].

Even if we settle with the idea that SAT can not be solved in polynomial time, it is not enough reason to give up. First, we must remember that the complexity estimation is for the worst case scenario. By worst case we mean, that any formula of a given size can be solved in that time. Many are possibly easier. Second, even exponential functions do not grow so rapidly if multiplied by a small constant (less than 1) or having a small exponent. Current modern SAT solvers use techniques to avoid searching unpromising

[1]Touring machine and equivalent models.

regions of the search space. This way problems of great size can be solved in reasonable time. Also they are implemented in a most effective way, so the exponential growth is as slow as possible.

In this thesis, we will focus on the first of these issues - search space pruning. The search space can be reduced by solving connected components of a formula separately. By connected components of a formula we mean subformulae corresponding to connected components of the formula's interaction graph. Solving components separately can speed up the solver exponentially. For example if a formula of n variables has two connected components of equal size, then it can be solved in $2^{n/2} + 2^{n/2} = 2^{1+n/2}$ time instead of 2^n. This approach was already used to design SAT solver decision heuristics[2, 19] or special SAT solver algorithms[4]. We will further investigate and precisely define the problem of connected components in SAT. It will be shown how this concept can be generalized to many other NPC problems.

The text will be partitioned the following way. Chapter 2 explains the problem we are solving and some of the procedures used to solve it. The third chapter deals with graphs and their connected components. Component trees are introduced here. The fourth chapter is dedicated to decision heuristics. We describe some heuristics used by state-of-the-art SAT solvers and suggest new ones. In Chapter 5 we describe the experiments we have done to measure the performance of the suggested decision heuristics.

Chapter 2

The Boolean Satisfiability Problem

2.1 Definition

In this section, we will provide the exact definitions of concepts necessary to understand the satisfiability problem.

Definition 2.1. The language of Boolean *formulae* consists of *Boolean variables*, whose values are *True* or *False*; Boolean connectives such as neg*ation* ($\neg$), *conjunction* ($\wedge$), *disjunction* ($\vee$), *implication* ($\Rightarrow$), *equivalence* ($\Leftrightarrow$); and *parentheses.*

A Boolean formula is a finite sequence of symbols from definition 2.1. The formulae are defined inductively.

Definition 2.2. Each Boolean variable is a *formula.* If A and B are *formulae* then $\neg A, (A \Rightarrow B), (A \wedge B), (A \vee B), (A \Leftrightarrow B)$ are *formulae* as well. *Formulae* are formed by a finite number of applications of these rules.

This definition enforces that every sentence constructed by Boolean connectives must be enclosed in parentheses. To improve readability, we can omit most of the parentheses, if we employ an order of precedence. The order of precedence in propositional logic is (from highest to lowest): $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$.

Definition 2.3. A *partial truth assignment* for formula F assigns a *truth value* (*True* or *False*) to *some variables* of F. A *truth assignment* for formula

F assigns a *truth value* to *every variable* of F. Both are functions from *variables* to *truth values*: $V : var(F) \rightarrow \{True, False\}$

Example 2.4. $F = x \vee (y \Rightarrow z) \Leftrightarrow (\neg x \wedge \neg y)$ is a Boolean formula. $\{v(x) = True, v(y) = False, v(z) = True\}$ is a truth assignment for F. $\{v'(z) = False\}$ is a partial truth assignment for F.

Given a truth assignment for a formula, which assigns truth values to its variables, we can consistently extend it to assign truth values to formulae with those variables. Such an extension of an assignment v will be denoted as v^*.

Definition 2.5. Let F be a formula and v a truth assignment.
if $F \equiv x$ (F is a variable) then $v^*(F) = v(x)$
if $F \equiv A \wedge B$ then $v^*(F) = \begin{cases} True & v^*(A) = True \textit{ and } v^*(B) = True \\ False & otherwise \end{cases}$
if $F \equiv A \vee B$ then $v^*(F) = \begin{cases} True & v^*(A) = True \textit{ or } v^*(B) = True \\ False & otherwise \end{cases}$
if $F \equiv A \Rightarrow B$ then $v^*(F) = \begin{cases} False & v^*(A) = True \textit{ and } v^*(B) = False \\ True & otherwise \end{cases}$
if $F \equiv A \Leftrightarrow B$ then $v^*(F) = \begin{cases} True & v^*(A) = v^*(B) \\ False & otherwise \end{cases}$

Example 2.6. If $F = (x \wedge y) \Rightarrow \neg x$, $\{a(x) = False, a(y) = True\}$ and $\{b(x) = True, b(y) = True\}$ then it is easy to verify, that $a^*(F) = True$ and $b^*(F) = False$.

Definition 2.7. A truth assignment v for F is called *satisfying* if $v^*(F) = True$. A Boolean formula F is called *satisfiable* if there exists a *satisfying assignment* for F. If F is satisfiable we write $SAT(F) = True$.

Example 2.8. In example 2.6 a is a satisfying assignment for F, b is not. F is satisfiable since a is the satisfying assignment.

Definition 2.9. A *decision problem* is a question with a yes-or-no answer. The *Boolean satisfiability problem (SAT)* is a decision problem of determining whether the given *Boolean formula* is *satisfiable*.

Before describing how SAT is solved, we need to define a special form of Boolean formulae - the *conjunctive normal form*[21].

Definition 2.10. A *literal* is a Boolean variable or its negation. A *clause* is a disjunction (or) of *literals*. A formula is in the *conjunctive normal form (CNF)* if it is a conjunction (and) of clauses.

Example 2.11. $F = (x_1 \vee x_2 \vee \neg x_4) \wedge (x_3 \vee \neg x_1) \wedge (\neg x_1 \vee \neg x_2)$ is in CNF. $(x_1 \vee x_2 \vee \neg x_4), (x_3 \vee \neg x_1), (\neg x_1 \vee \neg x_2)$ are clauses. $x_1, x_2, \neg x_4, x_3, \neg x_1, \neg x_2$ are literals.

Formulae F and *F'* are called *equisatisfiable* if $SAT(F) \Leftrightarrow SAT(F')$. Thanks to the Tseitin transformation[25], we can construct an equisatisfiable CNF formula for any formula in linear time. From now on we will work only with CNF formulae. It is the basis of the standard DIMACS format[27] for SAT solvers. CNF is widely used, because it is simple, easy to parse and represent in a computer's memory. A CNF formula is satisfied if all clauses are satisfied. A clause is satisfied if at least one of its literals is true. Hence the goal is to find such a truth assignment, that in each clause there is a literal, which is true.

2.2 Basic SAT Solving Procedure

Most of the current state-of-the-art SAT solvers are based on the Davis Putnam Logemann Loveland (DPLL) algorithm[21]. The DPLL algorithm is basically a depth-first-search of partial truth assignments with three additional enhancements. The explanation of these enhancements for CNF formulae follows.

- *Early termination.* If all literals are false in some clause, we can backtrack since it is obvious that the current partial truth assignment can not be extended to a satisfying assignment. If all clauses are satisfied, we can stop the search - we are finished. The remaining unassigned Boolean variables can be assigned arbitrarily.

- *Pure literal elimination.* A pure literal is a literal, the negation of which does not occur in any unsatisfied clauses. Unsatisfied clauses are the clauses not satisfied by the current partial assignment - none of their literals is true. Pure literals can be assigned to make them true. This causes that some clauses become satisfied and that might result in appearance of new pure literals.

- *Unit propagation.* A clause is called unit if all but one of its literals are false and the remaining literal is unassigned. The unassigned literal of a unit clause must be assigned to be true. This can make other clauses unit and thus force new assignments. The cascade of such assignments is called unit propagation.

In the DPLL procedure the enhancements are used after each decision assignment of the depth-first-search. First, we check the termination condition. If the formula is neither satisfied nor unsatisfied by the current partial assignment, we continue by unit propagation. Finally we apply the pure literal elimination. Unit propagation is called before pure literal elimination, because it can cause the appearance of new pure literals. The other way around, pure literal elimination will never produce a new unit clause, since it does not make any literals false. A pseudocode of DPLL is presented as algorithm 2.1.

Algorithm 2.1 DPLL

```
function DPLL-SAT($F$): Boolean
  $clauses$ = clausesOf($F$)
  $vars$ = variablesOf($F$)
  $e = \emptyset$ //partial truth assignment
  return DPLL ($clauses, vars, e$)

function DPLL($clauses, vars, e$): Boolean
  if $\forall c \in clauses, e^*(c) = true$ then return true
  if $\exists c \in clauses, e^*(c) = false$ then return false
  $e = e \cup$ unitPropagation($clauses, e$)
  $e = e \cup$ pureLiteralElimination($clauses, e$)
  $x \in vars \wedge x \notin e$ //x is an unassigned variable
  return DPLL($clauses, vars, e \cup \{e(x) = true\}$) or
    DPLL($clauses, vars, e \cup \{e(x) = false\}$)
```

Theorem 2.12. *DPLL is sound and complete (always terminates and answers correctly).*

Proof. DPLL is a systematic depth-first-search of partial truth assignments. The enhancements only filter out some branches, which represent not satisfying assignments. The theorem easily follows from these properties. □

It is easy to see, that the time complexity of this procedure is exponential in the number of variables. That corresponds to the number of vertices of a binary search tree with depth n, where n is the number of variables. In practice, thanks to unit propagation and early termination, the DPLL procedure never goes as deep as n in the search tree. The maximal depth reached during search is often a fraction of n. This makes DPLL run much faster on instances of a given size, than one would expect from the formula 2^n.

2.3 Resolution Refutation

Resolution is a rule of inference, which produces a new clause from clauses containing complementary literals. Two clauses C and D are said to contain complementary literals if there is a Boolean variable x such that $x \in C \wedge \neg x \in D$. The produced clause is called the *resolvent*. Formally:

Definition 2.13. Resolution rule

$$\frac{a_1 \vee a_2 \vee \ldots \vee a_{n-1} \vee a_n,\ b_1 \vee b_2 \vee \ldots \vee b_{m-1} \vee \neg a_n}{a_1 \vee \ldots \vee a_{n-1} \vee b_1 \vee \ldots \vee b_{m-1}}$$

where $a_1, \ldots, a_n, b_1 \ldots b_m$ are literals.

Resolution is a valid inference rule. The resolvent is implied by the two clauses used to produce it. If the resolution rule is applied to clauses with two pairs of complementary literals, then the resolvent is a tautology, since it contains a pair of complementary literals.

Resolution is the base of another sound and complete algorithm for SAT solving - *resolution refutation*. The algorithm takes input in form of a CNF formula. The formula is turned into a set clauses in an obvious way. This concludes the initialization phase. After that, the resolution rule is applied to each pair of clauses (with complementary literals) in our set. Resolvents, which are not tautologous are simplified (by removing repeated literals) and added to the set of clauses. This is repeated until an empty clause is derived or no new clause can be derived. If the algorithm stopped due to an empty clause then the formula is unsatisfiable, otherwise it is satisfiable. A pseudocode for this procedure is algorithm 2.2.

The resolution refutation algorithm always terminates, because there is only a finite number of clauses on a finite number of variables. The proof of its soundness and completeness is to be found in [21].

Algorithm 2.2 Resolution refutation

```
function resolution-SAT($F$): Boolean
  $clauses$ =clausesOf($F$)
  do
    $new = \emptyset$
    foreach $C, D \in clauses$, $C_i = \neg D_j$ do
      $R$ =resolve($C, D$)
      if $R$ is an empty clause then
        return false
      simplify($R$) //remove repeated literals
      if $R$ not tautology $\wedge$ $R \notin clauses$ then
        $new = new \cup \{R\}$
    endfor
    $clauses = clauses \cup new$
  while $new \neq \emptyset$
  return true
```

The complexity of this method is exponential in space and time. Practically it is unusable for SAT solving. For some formulae it can finish quickly, but there are families of formulae proven to have exponential lower bounds on resolution refutation length[26].

The reason for including this section in the text is, that it might be useful for a better understanding of the clause learning SAT solver concept.

2.4 Conflict Driven Clause Learning

At the beginning of section 2.2 we stated, that state-of-the-art SAT solvers are based on the DPLL algorithm. To be exact, they are actually based on a special kind of it - the conflict driven clause learning (CDCL) DPLL. It combines ideas of DPLL search and resolution refutation. Each time the DPLL encounters a *conflicting clause* (a clause that has all literals false) new clauses are resolved from the current ones and added to the formula. These new clauses are called *learned clauses* or *conflict clauses*. The terms conflict clause and conflicting clause sound very similar, so the reader must be cautious not to mix them up.

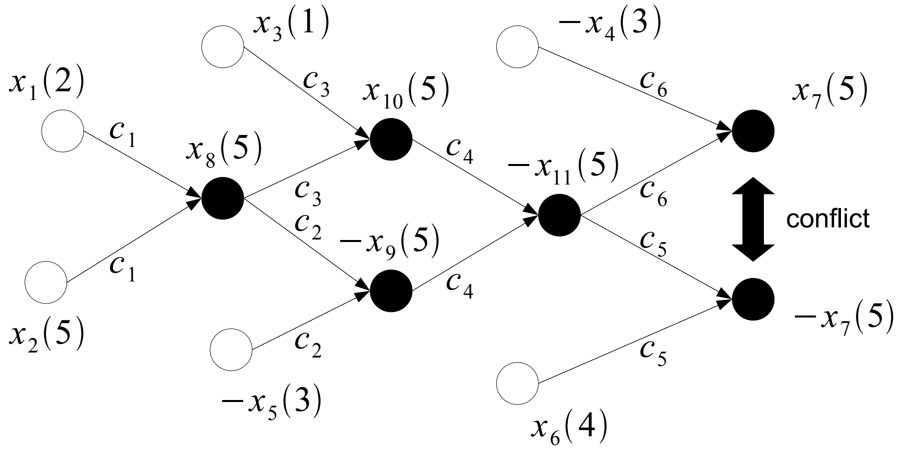
The solver GRASP in 1996[23] was among the first to implement clause learning. A simple and efficient method for learning was introduced by RelSat in 1997[16]. It was further improved by Chaff in 2001[18]. In this thesis only the basic ideas of clause learning will be described. The reader is referred to [3, 22, 30] for more information.

The recursive character of DPLL allows us to define *decision levels* of truth assignments. Decision level 0 is special. Assignments deduced from the input formula without any decision have decision level 0. Decision level n refers to the assignment implied by the decision in the n-th recursive call of DPLL and all assignments deduced from this assignment by unit propagation. When the solver backtracks to level l, all assignments with decision levels higher than l must be removed.

When the unit propagation deduces an assignment, it is due to a clause, which became unit. This clause is called the *reason* or *antecedent* of the assignment. The antecedent for literal x will be denoted as *ante(x)*. Assignments, which are due to a decision, have no antecedent. The antecedent relations between literals and clauses can be expressed in a form of an oriented graph. Such a graph is called an *implication graph*. A formal definition follows.

Definition 2.14. An *implication graph* is a directed acyclic graph $G(V, E)$ where V represents the assignments and $(x, y) \in E \Leftrightarrow x \in ante(y)$.

Figure 2.1: Implication graph



Example 2.15. Figure 2.1 is an example of a subgraph of an implication graph. Empty vertices represent decision assignments, filled vertices represent deduced assignments. The numbers in brackets denote the decision levels. The unit propagation is initiated by the assignment $x_2 = True$ on level 5. Edges are marked by reason clauses. The clauses used in this example are the following:

$c_1 = (\neg x_1 \vee \neg x_2 \vee x_8)$

$c_2 = (x_5 \vee \neg x_8 \vee \neg x_9)$

$c_3 = (\neg x_8 \vee \neg x_3 \vee x_{10})$

$c_4 = (\neg x_{10} \vee x_9 \vee \neg x_{11})$

$c_5 = (x_{11} \vee \neg x_6 \vee \neg x_7)$

$c_6 = (x_4 \vee x_{11} \vee x_7)$

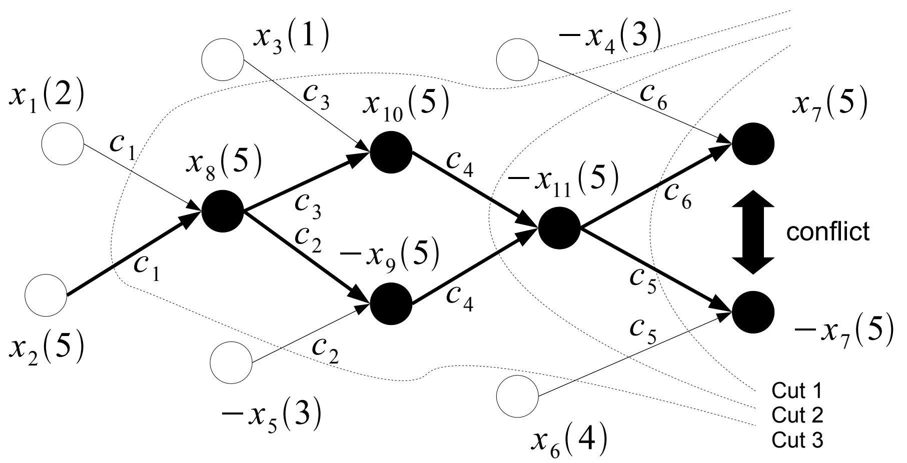
A subgraph of an implication graph containing a conflict is called a *conflict graph.* The subgraph on figure 2.1 is a conflict graph. The conflict graph describes why the conflict happened. We will use this graph to derive the clause to learn - the conflict clause.

The conflict clause is generated by partitioning the conflict graph into two parts. The partition has all the decision assignments on one side (*reason side*) and the conflicting assignments on the other side (*conflict side*). The described partitioning is called a *cut.* Vertices on the reason side contribute to the learned clause if they have at least one edge to the conflict side. We will refer to these vertices as *vertices of the cut.* The learned clause consists of the negations of the vertices of the cut, more precisely of the literals they represent. There are several ways to perform a cut of the conflict graph. Different cuts produce different conflict clauses.

Example 2.16. On figure 2.2 three different cuts of the conflict graph are displayed. The corresponding clauses are these:

Cut 1: $(x_4 \vee x_{11} \vee \neg x_6)$
Cut 2: $(x_4 \vee \neg x_{10} \vee x_9 \vee \neg x_6)$
Cut 3: $(x_4 \vee \neg x_3 \vee \neg x_1 \vee \neg x_2 \vee x_5 \vee \neg x_6)$

Figure 2.2: Conflict graph cuts



The clause constructed from a cut can also be constructed by resolving the clauses represented by the edges going to or inside the conflict side. The resolution of these clauses must be done in a proper order. Its description follows. First we resolve the clauses closest to the conflict (edges into the conflict literals). Then the resolvent is resolved with a clause that is represented by an edge into a vertex, whose negation is in the resolvent. We repeat this until all clauses from the cut have been used. See example 2.17 for a demonstration.

Example 2.17. As we can see on figure 2.2, *cut 1* intersects 4 edges, which represent 2 clauses c_5 and c_6. Resolving these clauses gives us $(x_4 \vee x_{11} \vee \neg x_6)$ which is the learned clause of *cut 1*.

The clause of *cut 2* can be derived by resolving c_5 and c_6 , then resolving their resolvent $(x_4 \vee x_{11} \vee \neg x_6)$ with c_4 and getting $(x_4 \vee \neg x_{10} \vee x_9 \vee \neg x_6)$.

The clause of *cut 3* is formed the following way. We start the same way as *cut 1*, by resolving c_5 and c_6 into $(x_4 \vee x_{11} \vee \neg x_6)$. We continue like for *cut 2* by resolving with c_4 and deriving $(x_4 \vee \neg x_{10} \vee x_9 \vee \neg x_6)$. The next clause to resolve with is c_2 or c_3. We can use them in arbitrary order. Let us, for example, select c_2 first and get $(x_4 \vee \neg x_{10} \vee \neg x_6 \vee x_5 \vee \neg x_8)$. Now we resolve with c_3 and get $(x_4 \vee \neg x_6 \vee x_5 \vee \neg x_8 \vee \neg x_3)$. Finally we resolve with c_1 and get the clause of *cut 3* which is $(x_4 \vee \neg x_3 \vee \neg x_1 \vee \neg x_2 \vee x_5 \vee \neg x_6)$.

If any clause derived from a cut of the conflict graph is also derivable by resolution, then adding these clauses to the original formula does not change the set of satisfying truth assignments. It remains to prove, the following theorem.

Theorem 2.18. *The extension of a CNF formula by a clause, which can be deduced by resolution from the original set of clauses, does not change the set of satisfying truth assignments of F.*

Proof. Let $(x_1 \vee x_2 \vee \ldots \vee x_{n-1} \vee y_1 \vee y_2 \vee \ldots \vee y_{m-1})$ be a new clause resolved from original clauses $(x_1 \vee \ldots \vee x_n)$ and $(y_1 \vee \ldots \vee y_{m-1} \vee \neg x_n)$. If a truth assignment v satisfied the original formula, then it also satisfied clauses $(x_1 \vee \ldots \vee x_n)$ and $(y_1 \vee \ldots \vee \neg x_n)$. We will show that v must satisfy the resolvent as well. $v(x_n)$ is either True or False.

If $v(x_n) = True$ then at least one of $y_1 \ldots y_{m-1}$ must be True in v, because $(y_1 \vee \ldots \vee \neg x_n)$ is satisfied. The literal which is True is also present in the resolvent and so makes it satisfied in v.

If $v(x_n) = False$ then similarly at least one of $x_1 \ldots x_{n-1}$ is True in v and that literal makes the resolvent satisfied.

We have showed that extending the set of clauses by a resolvent can not decrease the number of satisfying assignments. It also can not increase it, since adding any clause to a CNF formula can not. □

When learning clauses, we will prefer a special kind clauses - *asserting clauses.* An *asserting clause* is a clause with only one literal from the current decision level. The Clauses of *cut 1* and *cut 3* from example 2.16 are asserting, while the clause of *cut 2* is not. Why are asserting clauses desirable will be explained later when discussing backtracking.

To find cuts, which lead to asserting clauses, we need to locate *unique implication points.* A *unique implication point (UIP)* is a vertex in the conflict graph, that all oriented paths from the decision vertex to the vertices in conflict go through the vertex (these paths are highlighted by using thicker lines on figure 2.2). The decision vertex itself is always a UIP. Other UIPs in our example are x_8 and $\neg x_{11}$ (see figure 2.2). For each UIP we can perform such a cut of the conflict graph, that the vertices of the cut will contain the UIP as the only vertex from the current decision level. That cut will surely correspond to an asserting clause, since only the literal added due to the UIP will have the current decision level. The described cut is done by putting all the vertices, to which there is an oriented path from the UIP, into the conflict side. Such a partitioning is a valid cut and has the desired property.

One of the learning strategies is to make the cut at the *first UIP*. By first we mean the closest to the conflict. In our example it is $\neg x_{11}$ and so the first UIP cut is *cut 1*. Another strategy is to find the cut in such a way, that the learned clause will be asserting and of minimal length. A very simple strategy is a so called *RelSat scheme*[16]. In this strategy we resolve the clause in conflict with the antecedents of its literals until the resolvent contains only one literal from the current decision level. The first UIP scheme is often considered to be the best[30].

2.5 Conflict Driven DPLL

The conflict driven DPLL is a DPLL with CDCL and non-chronological backtracking. We already described what is CDCL. Non-chronological backtracking simply means, that we do not necessarily backtrack only level at a time. These two features are tightly connected. When a conflict is encountered, the level for backtracking is determined from the learned clause. It is reasonable, since the learned clause contains information about the conflict. The level of backtrack from a clause is determined the following way. We select a literal with the second highest decision level from the clause. The decision level of this literal is the proper level to backtrack to. A special case is when the learned clause has only one literal. When this happens, we backtrack to level zero. If the learned clause is asserting, then after backtracking it becomes unit and thus forces a new assignment.

The pseudocode of the CDCL DPLL is algorithm 2.3. First we initialize the partial truth assignment to be empty and the decision level to 0. Then we check if the unit propagation (Boolean constraint propagation - BCP) itself (without decisions) can prove the formula to be unsatisfiable. Any assignments made by the BCP at this point have decision level 0. These assignment will never be removed. After the initialization stage we enter an infinite cycle. In the cycle we make a decision to select the branching literal. If no branching variable can be selected, then all must be assigned. In this case we have a satisfying assignment and we are finished. If a literal is selected, then we enter the next decision level and extend the partial truth assignment by making it true. BCP follows. If no conflict is encountered, we continue by the next decision. However if a conflict appears, we must

Algorithm 2.3 CDCL DPLL

```
function CDCL-DPLL($F$): Boolean
  $clauses$ =clausesOf($F$)
  $e = \emptyset$ //partial truth assignment
  $level = 0$ //decision level
  if BCP($clauses$,$e$)= $false$ then return false
  while true do
    $lit$ =decide($F$,$e$) //an unassigned variable
    if $lit = null$ then return true
    $level = level + 1$
    $e = e \cup \{e(lit) = true\}$
    while BCP($clauses$,$e$)= $false$ do
       if $level = 0$ return false
       $learned$ =analyzeConflict($clauses$,$e$)
       $clauses = clauses \cup \{learned\}$
       $btLevel$ =computeBTLevel($learned$)
       $e$ =removeLaterAssignments($e$,$btLevel$)
       $level = btLevel$
    endwhile
  endwhile
```

process it. First, we check whether the decision level is 0, if this is the case, we can return the answer unsatisfiable. A conflict at level 0 means, that there is a variable, which must be true and false at the same time. It is like having two unary clauses, which contain complementary literals. If we resolve them, we get the empty clause. Some unary clauses can be in the input formula, some are derived through clause learning. The rest of conflict processing is straightforward. We compute the learned clause by analyzing the conflict. We add it to our set of clauses and compute the backtrack level from it. Last, we perform the backtracking by removing assignments after the backtrack level and updating the current level indicator. Note, that immediately after the backtracking BCP is called. This BCP will derive at least one new assignment thanks to the clause we learned being asserting.

Seeing the soundness and completeness of CDCL DPLL is not as trivial as it was for DPLL. It is not too difficult either. First, we show that the algorithm always terminates.

Lemma 2.19. *The CDCL DPLL algorithm terminates for every input formula F.*

Proof. If F has n variables, then the maximum decision level of any assignment will be *n+1*. A *decision level population vector (DPV)* is a sequence $c_0, c_1, \dots, c_{n+1}$ where c_i is the number of assignments with decision level i. If P and Q are DPVs, then we write $P > Q$ if P is lexicographically bigger than Q. $(0, \dots, 0)$ is the smallest possible DPV and $(n, 0, \dots, 0)$ is the largest. The last corresponds to the state when all variables have a value assigned at level 0. We will show the invariant, that if the DPV changes from P to Q then $Q > P$. The lemma is a consequence of the invariant, since for a finite n there is a finite number of different DPVs.

Now we prove the invariant. First, elements of the DPV are increased and never decreased by new assignments. Second, when we backtrack from level *l* to level *l'*, elements *l'+1 ... l* are zeroed. Elements *0 ... l'-1* are unchanged and element *l'* is increased. Element *l'* is increased due to the asserting clause we learn before backtracking. That clause becomes unit at level *l'* and forces a new assignment. Thus the DPV is greater than it was before. □

Theorem 2.20. *The CDCL DPLL algorithm always terminates and returns a correct answer.*

Proof. We have already proven termination (lemma 2.19). If the algorithm returns the answer satisfiable, then all variables are assigned and no conflict exists in the formula. If it returns unsatisfiable, then the empty clause can be derived by resolution, which implies the formula is indeed unsatisfiable. □

When implementing CDCL DPLL we must be cautious about the learned clauses. They can be numerous and can cause, that we run out of memory. For this reason, some learned clauses are deleted during the algorithm. Clauses to be deleted are often determined by the their length, activity or some other heuristic. Clauses, which are reasons for some assignments (they appear in the implication graph), must not be deleted.

Another important feature, which is never missing in the implementations of CDCL DPLL, is restarting. Restarting means to revert to decision level 0. We erase all assignments made at decision level 1 and higher. The learned clauses are not deleted, so we are not throwing away the work we have done. Restarting is done for the sake of diversification. Diversification is very important when solving SAT. It can correct the mistakes of the decision heuristics. Restarting greatly increases the set of problems a solver can solve and decreases the time to solve them.

Chapter 3

The Component Tree Problem

In this part we will investigate the structural properties of Boolean formulae. We will try to exploit these properties to solve formulae more efficiently.

3.1 Interaction Graph

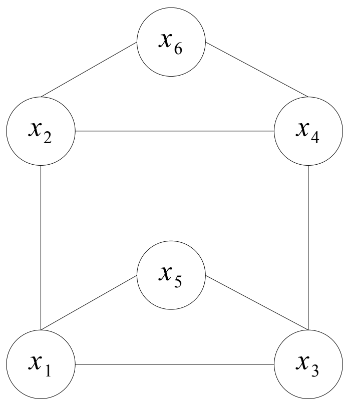
Definition 3.1. Let F be a Boolean formula in CNF. The *interaction graph* for F is the graph $G(V, E)$ where V represents the variables of F and $(x, y) \in E \Leftrightarrow \exists c \in clauses(F) : (x \in c \vee \neg x \in c) \wedge (y \in c \vee \neg y \in c)$.

In other words, the interaction graph of a formula has vertices, which represent the variables (not literals) of the formula and the vertices are connected by an edge, if they appear in a clause together. An example is given on figure 3.1. As apparent from the example, two different formulae can have identical interaction graphs.

If the interaction graph of a formula consists of more than one connected component, then the formula can be separated into subformulae corresponding to the connected components. These subformulae will have no common variable and thus they can be solved independently. The described subformulae of a formula will be called *components of the formula*. The original formula is satisfiable if all its components are satisfiable. Solving the components separately results in a significant speedup. For example if a formula of n variables has k components each with n/k variables, then the solving of the components separately will take $k \cdot 2^{n/k}$ time instead of 2^n.

Unfortunately, most of the formulae have only one component. For example formulae on figure 3.1 have only one component. If we solve a formula

Figure 3.1: Interaction graph example



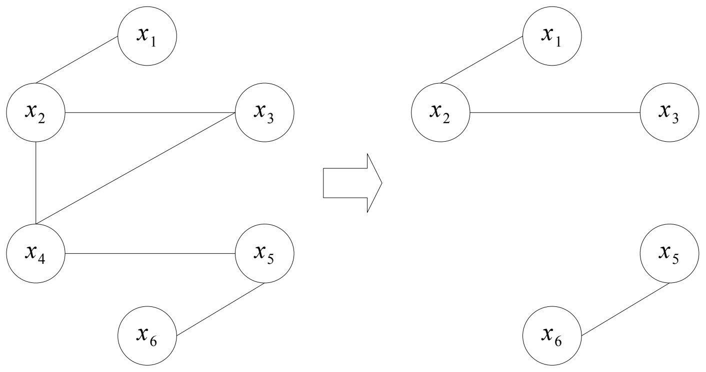
$F_1 = (x_1 \vee x_5 \vee \neg x_3) \wedge (\neg x_2 \vee \neg x_4 \vee \neg x_6) \wedge (x_1 \vee x_2) \wedge (x_4 \vee \neg x_3)$
$F_2 = (x_1 \vee x_3) \wedge (\neg x_5 \vee \neg x_3) \wedge (x_5 \vee \neg x_1) \wedge (\neg x_2 \vee \neg x_4 \vee \neg x_6) \wedge (x_1 \vee x_2) \wedge (x_4 \vee \neg x_3)$

using DPLL (or CDCL DPLL), we work with partial truth assignments. We can construct the interaction graph in regard of the partial truth assignment. In that case we ignore variables which have a value assigned and clauses which are satisfied. Formal definition follows.

Definition 3.2. Let F be a CNF formula and e a partial truth assignment for F. The *interaction graph* for F and e is the graph $G(V, E)$ where V represents the variables of F with no value defined by e and $(x, y) \in E \Leftrightarrow \exists c \in clauses(F) : (c\ not\ satisfied\ by\ e) \wedge (x \in c \vee \neg x \in c) \wedge (y \in c \vee \neg y \in c)$.

An example of an interaction graph for a formula and its partial truth assignment is on figure 3.2. We can see, that after we assign a value to x_4 the formula falls apart into two components. Now we can solve these components independently. So the plan seems to be, that we proceed as DPLL and after the unit propagation we check if the formula is still in one component. If it has been disconnected, we continue separately for the components. The problem with this plan is, that precise component detection is prohibitively expensive[19]. What we can do is some static component analysis in the phase of preprocessing or use special heuristics which do some inexpensive approximate component detection. We will discuss some heuristics of this kind in the next chapter. But first we show a method of static component analysis.

Figure 3.2: Implication graph for partial truth assignments



$F = (x_2 \vee \neg x_3) \wedge (\neg x_2 \vee \neg x_3 \vee \neg x_4) \wedge (x_6 \vee x_5) \wedge (x_2 \vee \neg x_1) \wedge (x_4 \vee \neg x_5)$
The first graph is for the empty partial truth assignment. The second is for $\{e(x_4) = false\}$.

3.2 Component Tree

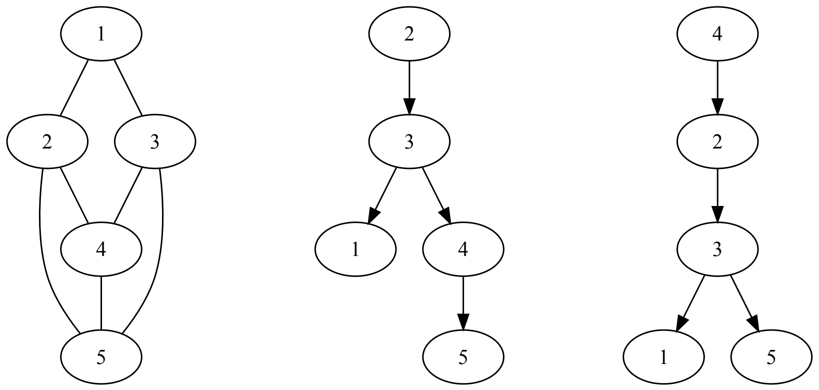
What we intend to do, is analyzing the interaction graph of a formula to determine which variables should be assigned values first to disconnect the formula in a good fashion. What we mean by good disconnection will be described soon. To explain precisely what we want, we define the *component tree*.

Definition 3.3. Let $T(V, E)$ be a tree and $v \in V$ be a vertex. The *root path* for v is the set of vertices on the path from the root of T to v including v.

Let $G(V, E)$ be a connected graph. The tree $T(V, E')$ is called a *component tree* for G if $\forall v \in V$ removing the root path of v from G causes, that the sets of vertices corresponding to the subtrees of v's sons in T become connected components in G.

The component tree shows us how the graph disconnects after removing some of its vertices. An example can be seen on figure 3.3. The first tree can be interpreted the following way. If vertices *2* and *3* are removed from the graph, then sons of *3* in the tree: *{1}* and *{4,5}* become connected components in the graph.

Figure 3.3: A graph and two of its component trees



Apparently there are several different component trees for a given graph. In the example on figure 3.3 we have two component trees. Which one is better? Component tree value will be defined. We will consider the tree with a lower value better.

Definition 3.4. Let $T(V,E)$ be a tree. The *component value* of a vertex $v \in V$ is defined as $val(v) = \begin{cases} 1 & v\ is\ a\ leaf \\ 2 * \sum_{s \in sons(v)} val(s) & otherwise \end{cases}$

The *component value of a tree* is the component value of its root.

The component values of component trees in the example on figure 3.3 are 12 and 16 respectively. It means, that the left component tree is better, because it has a lower component value. Our goal will be to find the best possible component tree for a given graph. By best we mean such a component tree, that there is no other component tree for the given graph with a lower component value. A tree with the given properties will be called an optimal component tree.

In the context of SAT, what we want is an optimal component tree for the formula's interaction graph. We will then use the component tree in our decision heuristic for the DPLL. The component value of a formula's interaction graph is an upper bound on the number of decisions a solver requires to solve the formula. The bound holds for a solver that somehow uses the component tree when making decisions. The details will be explained later. Now we only focus on the construction of component trees for general graphs.

For a better understanding of the concept of component trees we provide the following example.

Example 3.5. Let G be a clique on n vertices. The following statements hold:

1. Any component tree for G is a path of n vertices.
2. The order of the vertices in this path is arbitrary.
3. There are $n!$ component trees for G.
4. The component value of each component tree for G is 2^{n-1}.
5. Every possible component tree for G is optimal.

From the example we can see, that sometimes there are several optimal component trees for a graph. Also we can see that the number of component trees or even optimal component trees can be very large ($n!$ for n vertices).

Component trees are usually made of long *linear segments*. By *linear segment* we mean an oriented path, where the last vertex has zero or at least two sons and all other vertices have exactly one son. For example component trees for cliques have always only one linear segment. Component trees on figure 3.3 have both three linear segments. The linear segments of the first tree are *{(2,3),(1),(4,5)}* and of the second tree are *{(4,2,3),(1),(5)}*. An interesting property of the component trees is expressed by the following lemma.

Lemma 3.6. *For each component tree, the vertices in its linear segments can be arbitrarily permuted, and the component tree remains valid and also preserves its component value.*

Proof. We will prove the lemma by induction on the height of the component tree.

For component trees of height 1, the claim obviously holds. Let us have a component tree of height h.

First, let us assume, that its root has at least two sons. The subtrees of the sons have smaller heights, so the claim is true for them due to the induction hypothesis. The root is a one element linear segment and for those there is nothing to prove.

The second and final case is, that the root has exactly one son. Then it extends a linear segment S, where the son belongs. The definition of the component tree dictates, that removing the root path of a vertex disconnects the graph into connected components corresponding to the subtrees of the sons of the vertex. For vertices in the component tree, which have only one son, the definition requires nothing. Disconnection of a graph into 1 component is not a disconnection. Thus the only important vertex of each linear segment is the last vertex. In our case, the root path of the last vertex of linear segment S is equal to the set of the elements of S. Since removing a set of vertices from a graph results in the same graph, regardless of the ordering of those vertices, the order of vertices in a linear segment is not important. Thus the vertices can be permuted arbitrarily. The subtrees of our linear segment's sons are smaller component trees and the lemma holds for them by the induction hypothesis.

The preservation of the component value is obvious, since the shape of the tree does not change at all. Only some of the vertices are "renamed". □

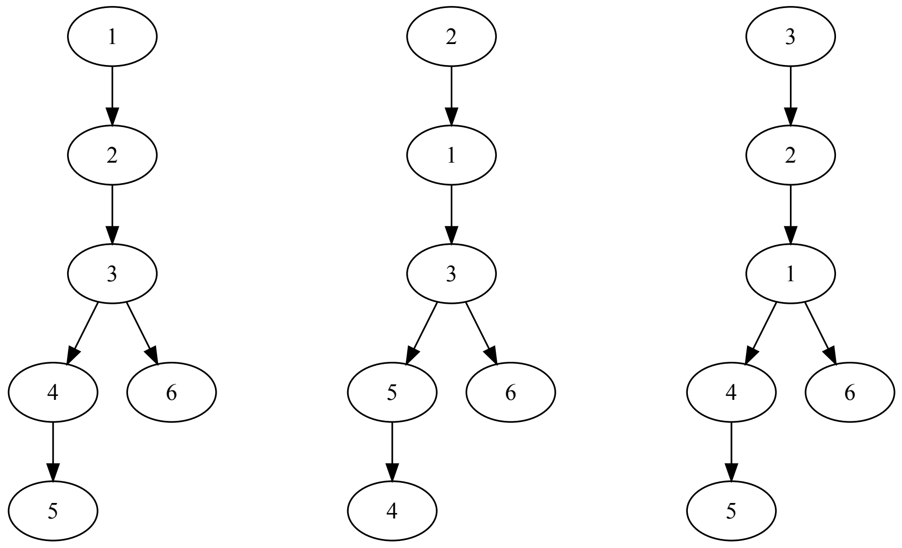
The lemma says, that the component trees on figure 3.4 are equivalent. If one of them is a valid component tree for a graph, then the other two are also valid for that graph. This property is very useful. We will use it when designing decision heuristics for a SAT solver.

The problem of finding an optimal component tree for a given graph will be called *the component tree problem (CTP)*. The *decision version of the component tree problem* is the yes-or-no question: Is there a component tree for a graph G with a component value less than or equal to v? We will show, that the decision version of CTP is in NP. It remains open, whether it is NPC.

Lemma 3.7. *The decision version of CTP is in NP.*

Proof. The certificate is the component tree itself. First, its size is clearly polynomial. Second, we can surely verify the validity of a component tree for a given graph in polynomial time. It can be done, for example, by verifying the requirement from the component tree definition for every vertex. Third, we can also verify, that the component tree has the required component value. This value can be exponential in the number of vertices, but if we use binary encoding, it can be done in polynomial time. □

Figure 3.4: Permuted linear segments



Theorem 3.8. *The CTP can be solved in polynomial time on a non-deterministic Touring machine.*

Proof. The component tree value for a tree with n vertices is a positive number less than 2^n. We can use binary search to find the best (lowest) component value component tree for a given graph. We use the decision version of CTP to check if a solution of a given quality exists. Binary search on 2^n possible values takes $\log_2(2^n) = n$ time. The described algorithm calls n times the decision version of CTP, which takes polynomial time on a non-deterministic Touring machine (lemma 3.7). $\square$

3.3 Component Tree Construction

Now we will present two algorithms for component tree construction. The first one is a depth-first-search with a few additional instructions. We will call it *DFS Find*. This algorithm is very fast, but it often yields a solution very distant from being optimal. Its Pseudocode is presented as algorithm 3.1.

Algorithm 3.1 DFS Find component tree construction

```
find($v, G, T(V, E)$)
  $V = V \cup \{v\}$
  for $u \in Neighbours(G, v)$ do
    if $u \notin V$ then
      find($u, G, T(V, E)$)
      $E = E \cup \{(v \rightarrow u)\}$
    endif
  endfor
```

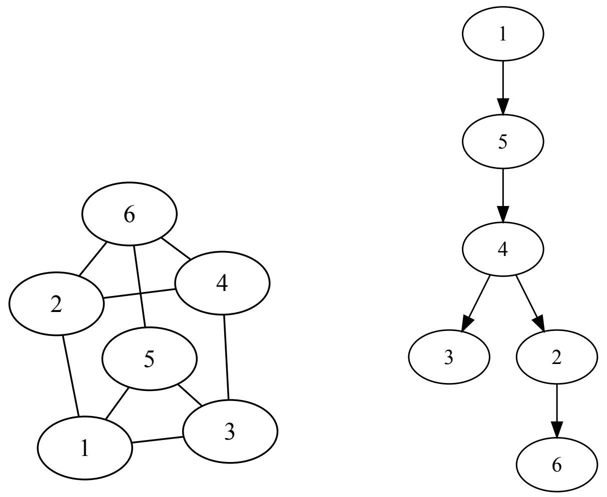
This algorithm creates a special kind of component trees. If two vertices x and y are connected in the component tree then they are also connected in the graph. This kind of a tree is called a *depth-first-search tree (DFS Tree).* A component tree for a graph is not necessarily a DFS tree. On figure 3.5 we see a graph and its optimal non DFS component tree. If there is no optimal component tree for a graph which is DFS, then this algorithm can not find an optimal solution. Such graphs exist, an example is on figure 3.5. DFS Find would create only a path. Starting from any vertex and taking the neighbors in any order always results in a path. The reason is, that DFS Find must always proceed to a neighbor.

Now we will describe a better algorithm. DFS Find built the component tree from its root to the leaves. The second algorithm does it the opposite way. Starting from the leaves and connecting them into small trees. Connecting small trees into bigger ones and finally to one component tree. This algorithm will be called the *component tree builder (CTB)* algorithm. Its pseudocode is algorithm 3.2.

The algorithm works with a forest of component trees. When processing a new vertex it checks its neighbors for vertices which are already in the forest. The algorithm saves the roots of the trees where those neighboring vertices belong. The new vertex then becomes the parent of these roots. This way the new vertex either enlarges one of the trees in the forest (by becoming its new root) or connects two or more trees into one (by becoming a common root).

The main for cycle enumerates through the edges of the input graph. The order of the vertices is significant for the component value of the resulting component tree. Any permutation of vertices is good, the component tree built using that ordering will be a valid component tree. This statement

Figure 3.5: No DFS optimal tree.



also holds the other way around.

Lemma 3.9. *Any valid component tree of a graph can be constructed by the CTB algorithm given the proper ordering of vertices.*

Proof. Let T be a valid component tree. The proper ordering of vertices for constructing T can be acquired the following way. Run DFS on T and output the vertex when you visit it the last time (when you are returning to its parent). In other words, the proper order is gained by DFS postordering. □

Lemma 3.9 implies, that the CTB algorithm has the potential of finding an optimal component tree, since it can find any valid component tree. The only problem is to guess a good ordering. This gives us a trivial algorithm for finding an optimal component tree: run the CTB algorithm for each permutation of vertices and return the best result. The complexity is $n!$, since we must test each possible permutation. This makes it impossible to use in practice. Instead of trying out all possible orderings, we will guess a good one and build the tree according to it. To guess a good ordering we will use the following heuristic.

Algorithm 3.2 Component tree builder

```
ComponentTreeBuild($G(V, E)$)
  $V' = \varnothing, E' = \varnothing$
  for $v \in V$ do
    $R = \varnothing$
    for $s \in Neighbours(G, v)$ do
      if $s \in V'$ then $R = R \cup \{$rootOf($s$)$\}$
    endfor
    $V' = V' \cup \{v\}$
    for $r \in R$ do
      $rep(r) = v$
      $E' = E' \cup (v \to r)$
    endfor
  endfor
  return $T(V', E')$

rootOf($v$)
  while $rep(v)$ defined do $v = rep(v)$ endwhile
  return $v$
```

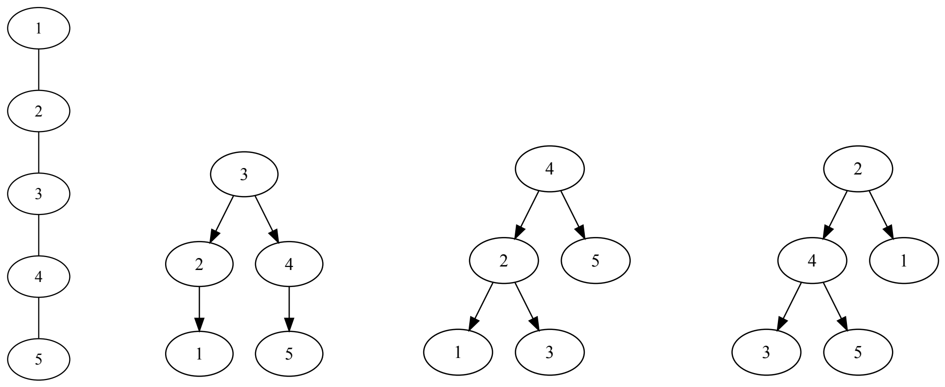
Definition 3.10. *Greedy heuristic*: Compute the score of the vertices, which have not yet been used. The score of a vertex is a sum of the number of its neighbors and its potential component value. The potential component value is the value of the component tree, that would be formed, if this vertex was used in the current step. Select a vertex with the lowest score.

Experiments have showed, that the CTB algorithm with the greedy heuristic (Greedy CTB) produces much better component trees than the DFS Find algorithm or the CTB algorithm with a random heuristic (random order of vertices). But still, Greedy CTB is not optimal. A counter-example for optimality is displayed on figure 3.6.

3.4 Compressed Component Tree

According to lemma 3.6 the order of vertices in the linear segments of component trees is unimportant. This allows us to look at those linear segments as sets of vertices instead of sequences. To emphasize this, we will contract

Figure 3.6: Counter-example of Greedy CTB optimality



The first component tree is optimal for the graph with the value 8. Greedy CTB would never produce it, since after selecting vertices 1 and 5 first (their score is 1), it would select 3 (with score 2). Greedy CTB would produce the second or the third tree, both non-optimal with value 10.

the linear segments into single vertices. A tree with contracted linear segments will be referred to as a *compressed component tree*. An example of a tree and its compressed equivalent is on figure 3.7.

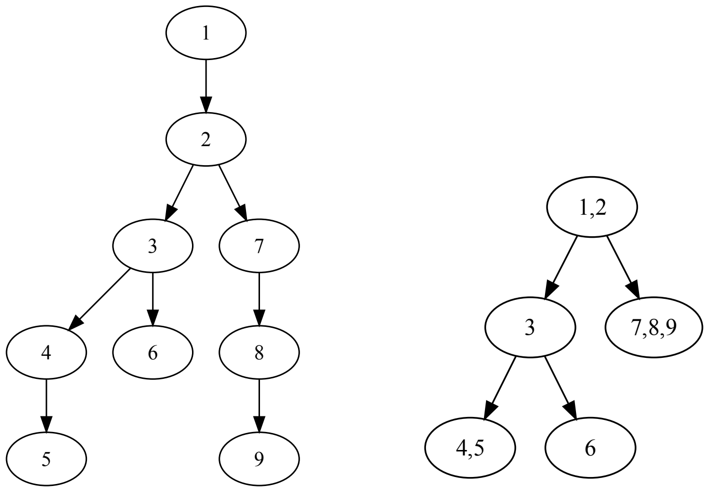
The component value of a compressed tree is the component value of the component tree, that was compressed to obtain it. If we need a compressed component tree for a graph, we can make it by constructing a regular component tree and the contracting its linear segments. Also the algorithms described in section 3.3 can be easily modified to construct compressed component trees directly.

3.5 Applications

The concept of component trees was created for the purpose of SAT solving. It was designed for analyzing interaction graphs of Boolean formulae. However, it can be used for many other NP problems. All problems, which are solved by searching the universe of possible values for some variables, could potentially benefit from this idea.

The most straightforward application could be the coloring of a graph

Figure 3.7: Compressed component trees



by 3 colors. In this case we would create a component tree for the input graph itself. If we disconnect the graph by coloring some of its vertices, the components can be colored independently. If we used the number 3 in the definition of the component tree value (instead of 2), it would represent the maximum number of trials for solving the 3-coloring of a given graph. The situation is, of course, analogous for coloring by any number of colors (higher than 2).

An application for solving constraint satisfaction problems (CSP)[21] is also possible. We can create a graph similar to the interaction graph. The vertices represent the variables. Two vertices are connected by an edge, if there is a constraint, that contains both the variables assigned to those vertices. The component value can be defined the following way. The value of a vertex is the sum of the values of its sons multiplied by the size of its variable's domain. If a vertex has no sons, we can define its value as the domain size of its variable.

In this thesis, we will experimentally investigate the usefulness of component trees for SAT solving. It would be interesting to do a similar research for CSP. That might be a promising subject for future work.

Chapter 4

Decision Heuristics

In this chapter we return to SAT solving. We stated, that the way of solving SAT is CDCL DPLL. If we take a look at its pseudocode (algorithm 2.3), there is a function *DECIDE*. This function is expected to return an unassigned variable or its negation - an unassigned literal. If there are many unassigned variables, the function has many possibilities for literal selection. Selecting a good literal, a literal that will cause the algorithm to finish quickly, is a hard task. For satisfiable formulae an ideal variable selection procedure would render the DPLL a linear time algorithm. Unfortunately we do not have such a procedure yet. Solvers are using heuristics instead. These heuristics are called *decision heuristics*.

In this chapter we will describe some of the well known decision heuristics. Then we introduce a new heuristic based on the component tree concept. We will show, how it can be combined with other heuristics.

4.1 Jeroslow-Wang

The Jeroslow-Wang (JW)[15] is a score based decision heuristic. Score based means, that we compute a numeral score for each literal and we select a literal with the highest or lowest score. In the case of JW we select an unassigned literal with the highest score. The score for a formula F is defined by the following equation.

$$s(lit) = \sum_{lit \in c, c \in F} 2^{-|c|}$$

In the equation above c represents a clause and $|c|$ represents its size (number of literals). The preferred literals are those, which appear in many short clauses. The scores of the literals are computed once at the beginning of the solving. This makes JW a static heuristic. It means, that the process of solving does not influence the decision making.

When learning clauses, we can update the scores of their literals by adding $2^{-|c|}$, where c is the learned clause. This is in accordance with the definition. Updating scores using learned clauses makes JW a dynamic heuristic. In opposition to static heuristics, the literal selection of dynamic heuristics is influenced by the going of the solving algorithm. We will use this dynamic version of JW for our experiments.

4.2 Dynamic Largest Individual Sum

The dynamic largest individual sum (DLIS)[23] heuristic is also score based like JW. The score of a literal in this case is the number of clauses containing it. Only clauses, which are not satisfied at the current decision point are considered. This heuristic is dynamic because of this property. An unassigned literal with the highest score is selected as a decision literal.

The aim of this heuristic apparently is to satisfy as many clauses as possible. DLIS has a significant disadvantage, its computation is very time consuming. The reason is, that only the unsatisfied clauses are counted. We can implement it by recomputing the scores of literals at each decision, which is obviously very slow. Another way is to keep updating the scores when clauses are becoming satisfied or not satisfied (when backtracking). The second way appears to be more efficient, but the updating slows down the solver too much.

We would forgive the slowness of DLIS, if it yielded good decision literals. It, unfortunately, does not. The are many other heuristics, which are fast to compute and solve most of the formulae using fewer decisions.

4.3 Last Encountered Free Variable

The last encountered free variable (LEFV)[2] is different from the majority of heuristics, since it is not score based. LEFV uses the propagation of the DPLL procedure to find a literal for the decision. The propagation always starts with a literal. We check the clauses, where the negation of this literal appears. Those clauses are candidates for unit clauses. Some of them are indeed unit, but many are not. We keep a pointer to the last not unit and not satisfied candidate clause we encounter during the propagation. When a decision is required, we select an unassigned literal from this clause.

This heuristic is very easy to implement and its time and memory complexity are minimal. It has a special property, which could be called component friendliness. This heuristic tends to solve the components independently. Since the propagation for a literal does not leave the component of the formula where the literal is, the next decision variable is surely selected from this component again. A more precise formulation and proof of this property is to be found in [2].

4.4 Variable State Independent Decaying Sum

One of the most important solvers in the history of SAT solving was Chaff[18]. One of its many contributions is the variable state independent decaying sum (VSIDS) heuristic. VSIDS is again score based. The literal with the highest score is selected as the decision literal. The scoring system description follows.

Each literal l has a score *s(l)* and an occurrence count *r(l)*. Before the search begins *s(l)* is initialized as the number of clauses, where l appears. When a clause is learned, we increment the occurrence count *r(l)* for each literal it contains. Every 255 decisions the score of each literal is updated, *s(l)* becomes $s(l)/2 + r(l)$ and *r(l)* is zeroed.

The score is similar to the score of DLIS, but here we do not care if the clause is satisfied or not. Another difference is, that we periodically half the scores to increase the impact of recently learned clauses and the literals they contain.

A little disadvantage is, that the reaction of this heuristic to the most recent solution state is delayed. It is because of the scores are updated only every 255 decisions. Nevertheless, this heuristic performs very well and is also fast to compute.

4.5 BerkMin

The BerkMin heuristic[12] is a heuristic, which also uses scores of literals. However it does not select the highest scoring literal for the decision as the other score based heuristics. The decision literal is selected from the most recent not satisfied learned clause. The literal with the highest score from the literals in the clause is selected. Now we describe, how the score of the literals is computed.

Similarly to VSIDS or DLIS the score of a literal is initialized as the number of clauses containing the literal. When a conflict happens, all the clauses participating in it are registered in the scoring system. By registering a clause in the scoring system we mean, that we increase by one the score of the literals in it. Clauses participating in the conflict are all clauses, that are used to produce the learned clause. In example 2.17 we resolved clauses from the implication graph to create the conflict clause. All these clauses used in the resolution are participating in the conflict. To be concrete, using example 2.17, if we learned the clause of *cut 2*, the conflict participants would be clauses c_5, c_6 and c_4. We would increase the scores of literals $\neg x_{10}, x_9, \neg x_{11}, \neg x_6, \neg x_7, x_4, x_7$ by one and the score of literal x_{11} by two.

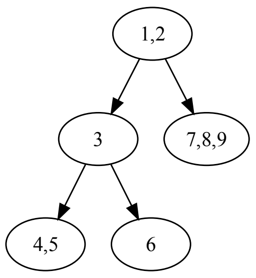
Unlike VSIDS the scores are not decaying. Another difference is, that the scores are increased immediately, not only every 255 decisions. This addresses the issue of delayed reaction. By registering all the clauses participating in the conflict, we extract more information from the conflict. The VSIDS registered the learned clause only. There could be an important literal, which played a significant part in the conflict, but did not get into the learned clause. VSIDS would not credit it, but BerkMin does according to its frequency in the participating clauses. These differences are probably the reasons for BerkMin being a better heuristic than VSIDS. BerkMin's computational complexity is low and its performance is spectacular. Almost all of the best current SAT solvers use this decision heuristic.

4.6 Component Tree Heuristic

In this section, we introduce the component tree heuristic (CTH). The idea is very straightforward and foreseeable. Let us have a compressed component tree for the formula we are solving. We start with the root node. We keep selecting a free variable from the current node until possible. If there are no free variables, we continue to the next node like in a regular DFS of a tree. We present an example on figure 4.1.

The idea is to select the variables which disconnect the formula first. When moving to the next node in the compressed component tree, there can be several possibilities to continue. Concretely when moving to one

Figure 4.1: Component tree heuristic



The order of nodes is {1,2}, {3}, {4,5}, {6}, {7,8,9} or {1,2}, {7,8,9}, {3}, {4,5}, {6}. Another two possibilities are the described ones with {4,5} and {6} swapped.

of the sons, we can choose which son will be visited first. Three simple strategies for son selection are:

- *Random* son selection.
- *BigFirst* son selection. We select the son with the highest component value first.
- *SmallFirst* son selection. The lowest component value son is selected.

We will investigate experimentally, in the next chapter, which is the best. But without experiment, one could expect the SmallFirst strategy to be the best. Since this represents the fail-first idea. The small component can be proved to be unsatisfiable faster, so the solver can backtrack sooner to correct its previous wrong decisions. On the other hand, Random represents diversification. The other two strategies select the same ordering of sons each time the search goes around.

This heuristic has an interesting property, which is formulated in the next theorem. To prove it, we will need the following lemma.

Lemma 4.1. *A CDCL DPLL solver using the CTH always backtracks to a decision level corresponding to a variable from a compressed component tree node, which is a predecessor of the current node.*

Proof. We backtrack to the second highest level among the levels of literals in the learned clause. We will do the proof by contradiction. Let us assume, that the level we should backtrack to corresponds to a decision variable v located in the node B which is not a predecessor of the current node C. When v was selected as the decision variable, all the variables in the nodes, which are common predecessors of B and C had already been assigned. This means, that the variables of B and C were in separate components. Thus the variable v can not be in any connection with the current conflict and is not present in the learned clause. □

What the lemma says is, that we will never backtrack to a node of the compressed component tree, from which there is no path to the node, where the conflict appeared. We will not backtrack for example to our brother node or any of his successors. Now we are ready to prove the property of the CTH, which we advertised before the lemma.

Theorem 4.2. *Let us have a CDCL DPLL solver with the CTH. Let us have a Boolean formula and a component tree with value V for its interaction graph. Then the solver solves the formula using at most V decisions.*

Proof. We do the proof by induction on the size of the component tree. If the tree has one node, then the formula has one variable. Such a formula is surely solvable using 1 decision.

Let us have a tree of size n and let its root have one son. The subtree defined by the son is of size *n-1* and the induction hypothesis says, that the formula represented by the subtree can be solved using at most V_{son} decisions. V_{son} is the component value of the son. Now we add a new variable to the formula and since variables in a Boolean formula have 2 possible values, the new formula can be solved using at most $2 * V_{son}$ decisions.

Now let the root of the component tree have at least two sons. Each of the sons are roots of smaller component trees, so the theorem holds for them. These smaller component trees represent components of a formula and can be solved independently. The CDCL DPLL will indeed solve them separately. Thanks to lemma 4.1, we never return to a brotherly component, but we proceed to the next (if this component was satisfied) or go back to a predecessor (if this component is unsatisfiable). Thus we can sum the component values of the sons. We multiply it by two for the same reason as in the previous case.

We have proved, that the component value, as we have defined it, corresponds to the maximum number of decisions required to solve the formula. □

Thanks to this theorem we are able to estimate the time needed to solve a formula. A question is, how useful it is. Is it not a very rough estimation? Probably it is, since the component tree concept does not take account of the propagation. We will answer this question to some extent using experiments in the next chapter.

4.7 Combining CTH with Other Heuristics

The CTH instructs us a to select a free variable form the current compressed tree node. These nodes can contain numerous variables, which of them should we select first? We can select randomly, but this would produce a very poor heuristic. According to experiments, such a heuristic is similar to a full random decision heuristic (without a component tree). For this reason we will not consider this option. A better option is to combine CTH with some good heuristics. For example the heuristics we described in the first 5 sections of this chapter. We will now describe how exactly do we combine these heuristics with CTH.

The combination is simple. We use the original heuristic to select a free variable, but we restrict it to select from among the variables in the current node. When we combine JW with CTH, we select the variable from the current node with the highest JW score. Analogously for DLIS and VSIDS, but we use their score definition.

BerkMin and LEFV are combined a little bit differently. For LEFV we select such a variable from the last encountered clause, which is from the current node. If this can not be done, we select a random variable from the current node. For BerkMin we take the learned clauses in order from the most recent to the oldest, until we find a clause, which is not satisfied and also contains a variable from the current node. If there are more literals in that clause from the current node, we select one according to the BerkMin scoring system. Again, if we can not find such a clause, we select a random free variable from the current node.

Let us note, that all these combined heuristics are instances of the CTH. Thus lemma 4.1 and theorem 4.2 holds for them. The combined heuristics perform well. The better heuristic we use in the combination, the better the combined heuristic is. This property was observed from the experiments we made.

4.8 Phase Saving

There is a special kind of heuristics, called *phase heuristics*. These heuristics do not select a free variable, they only select the phase for a variable selected by someone else. In other words a phase heuristic decides, if a variable should be used as a positive or negative literal. It takes a variable or literal as input and returns a literal. A phase heuristic is called immediately after the decision heuristic in the solving algorithm.

Now we describe a concrete phase heuristic called *phase saving*[19]. For this heuristic we must keep record of all the assignments to variables, even those, which are now removed due to backtracking or restarts. From the input literal we extract the variable. If this variable already had a value assigned to it, then we assign the same value again. So if the last assigned value was false, we return a negative literal of the variable. If it was true, we return a positive literal. If the variable has never had any value assigned yet, we return the literal from the input. We do not change its phase.

This heuristic can be computed in constant time. We need some memory to store the assigned values of variables, but it is not much. The motivation of this heuristic is also component related. The idea is, that when we solve a component and then backtrack or restart, its variables are unassigned. Then we get to the component again. Now the phase log contains a solution for this component, so we just assign its variables the same way as they were before. This way we do not have to solve the same component again and again.

Chapter 5

Experiments

To measure the performance of the heuristics described in the previous chapter, we conducted experiments. We implemented two CDCL DPLL solvers using the usual state-of-the-art techniques. The first was implemented in Java and it could be better called a heuristic investigation tool. It allows testing of all the described heuristics and also can perform formula analysis and output the interaction graph or the component tree. The second solver is implemented in C++. Its aim is to implement the best combination of the parameters and properties we discovered using the Java solver. These are implemented in an efficient manner. The Java solver will be referred to as *SatCraftJava* and the C++ version as *SatCraft*. More information on the implementation of the solvers is presented in appendix A.

5.1 Benchmark Formulae

We used two sets of benchmark formulae. The first is a set of uniform random 3-SAT formulae from the phase transition area. These formulae are generated the following way. Let us assume, that we want a formula with n variables and k clauses (each contains 3 literals, hence the name 3-SAT). Each of the k clauses is generated the same way. We draw 3 literals from the $2n$ possible literals randomly, each literal has the same probability to be selected. Clauses which contain two copies of the same literal or are tautologous (contain a literal and its negation) are not accepted for the construction. We continue until we have k valid clauses.

The phase transition area[5] is a ratio of the number of variables and clauses, where a rapid change of solubility for random 3-SAT formulae can be observed. What we mean is, that when the number of variables is fixed and we increase the number of clauses systematically, then there is a number k that almost all formulae with less than k clauses are satisfiable and almost all formulae with more than k clauses are unsatisfiable. For random 3-SAT the phase transition occurs approximately at $k = 4.26 * n$. We can also say, that random 3-SAT formulae with this ratio of variables and clauses are satisfiable with the probability of 50%. Phase transition random formulae are considered to be the hardest. We selected 800 formulae of this kind. See table 5.1 for their description. The formulae were acquired from [13].

Table 5.1: Phase transition random 3-SAT formulae

filename	#vars	#clauses	#instances	SAT
uf125*	125	538	100	yes
uf150*	150	645	100	yes
uf175*	175	753	100	yes
uf200*	200	860	100	yes
uuf125*	125	538	100	no
uuf150*	150	645	100	no
uuf175*	175	753	100	no
uuf200*	200	860	100	no

The second set is a compilation of structured formulae. These are various problems encoded to the language of Boolean satisfiability. Our goal was to select benchmark problems of as many kinds as possible. In fact there are not as many publicly available formulae as one would expect. The formulae we used are listed in table 5.2. With this kind of formulae, one can not estimate their difficulty by their size. For example the *bmc* formulae have tens of thousands of variables, but good solvers solve them in a matter of seconds. On the other hand problems like *hole* or *urq* are very difficult while having only a few hundred variables.

In the following sections we will present the results of our experiments with these formulae. We will present them visually using plots. If the reader is interested in more detailed results, they are to be found on the enclosed CD.

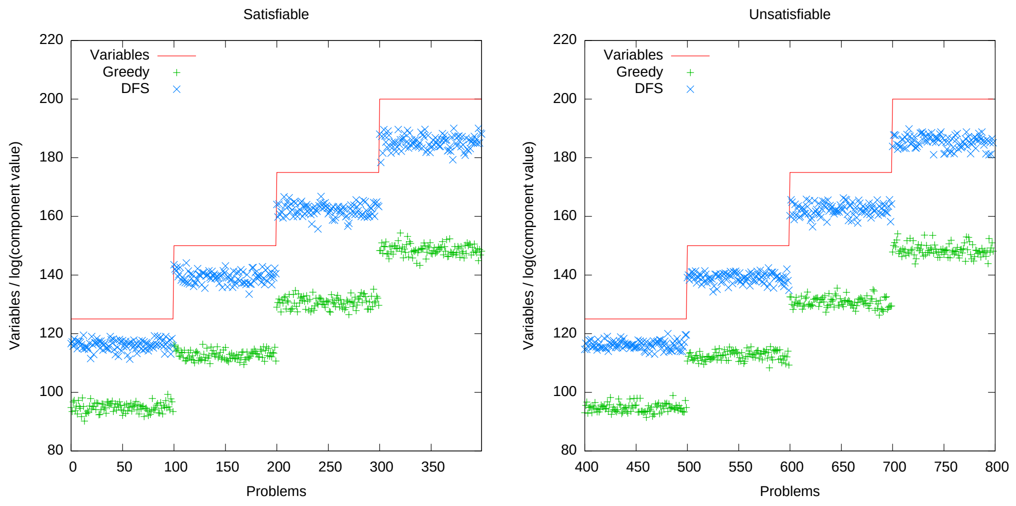
Table 5.2: Structured benchmark problems

name	description	vars	clauses	inst.	SAT	src
flat	flat graph coloring	600	2237	100	yes	[13]
parity	parity games	27 - 14896	53 - 153982	25	mixed	[10]
frb	forced satisfiable RM model	450 - 595	19084 - 29707	10	yes	[29]
qg	quasigroup (Latin square)	343 - 2197	9685 - 125464	22	mixed	[13]
jarvisalo	multiplicator equivalence	684 - 2010	2300 - 6802	6	no	[14]
hanoi	towers of Hanoi	718 - 1931	4934 - 14468	2	yes	[13]
bmc	bounded model checking	3628 - 63624	6572 - 368367	18	yes	[13]
logistics	logistics planning	828 - 4713	6718 - 21991	4	yes	[13]
bw	blocksworld planning	48 - 6325	261 - 131973	7	yes	[13]
difp_w	factorization (Wallace tree)	1755 - 2125	10446 - 12677	15	yes	[1]
difp_a	factorization (array multiplier)	1201 - 1453	6563 - 7967	14	yes	[1]
beijing	Beijing SAT competition benchmarks	125 - 8704	310 - 47820	10	mixed	[13]
urq	randomized Urquhart	46 - 327	470 - 3252	6	no	[1]
chnl fpga	FPGA switchbox	120 - 440	448 - 4220	15	mixed	[1]
hole	pigeon hole	56 - 156	204 - 949	6	no	[1]
s3	global routing	864 - 1056	7592 - 10862	5	yes	[1]

5.2 Component Values

In this section we compare the algorithms for obtaining component trees. We measured the component values of the trees produced by the DFS Find

Figure 5.1: Component values on random 3-SAT



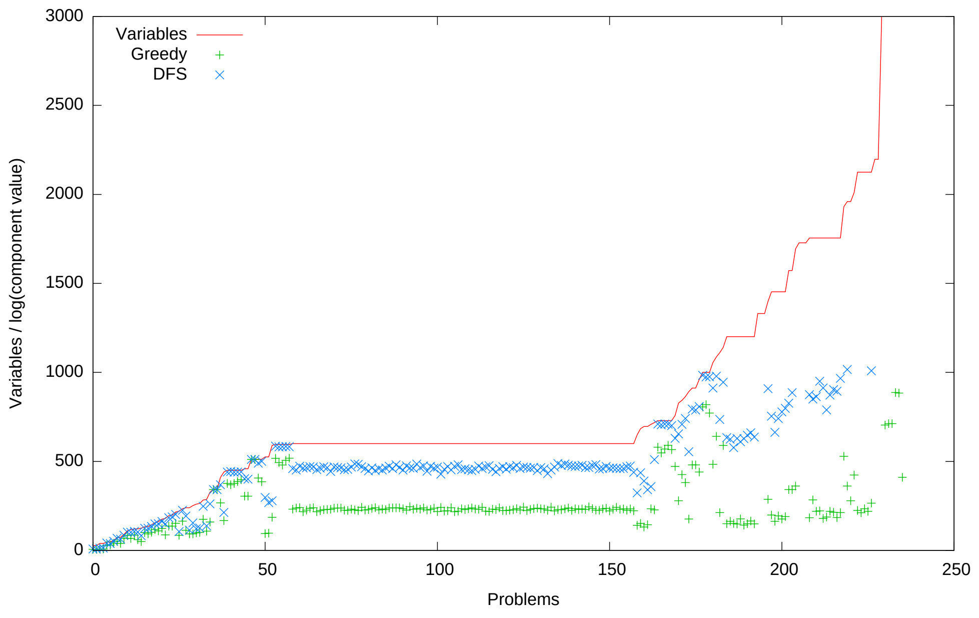
algorithm and the Greedy CTB algorithm.

On figure 5.1 we displayed the results on the random set of formulae. On the left side are the satisfiable instances, on the right the unsatisfiable. Instead of plotting the actual component values, we used their base 2 logarithms. We also plotted the number of variables for comparison. So we are actually comparing the trivial upper bound (2^{vars}) versus our upper bound ($2^{\log(v)}$, where v is the component value) in a logarithmic scale. As we can see from the plot, the Greedy CTB algorithm is consistently better than DFS Find, which was expected. The logarithm of Greedy CTB's value is about 75% of the number of variables. Thus our upper bound is a bit better than the trivial one, but not too much.

We performed the same experiment for our second set of problems. We omitted the *bmc* problems, because of their large size. The results are presented on figure 5.2. Again we used the logarithms of component values. The problems are sorted according to the number of variables. The y axis of the plot is cut at 3000 variables. Thus problems with more than 3000 variables are not displayed. The component value for them exceeded the range of the type double, so we would not see the results for those problems anyway. The mid section with 600 variables represents the flat graph coloring formulae.

The first thing to notice, is that the logarithms of component values are in many cases much lower than the number of variables. Especially the

Figure 5.2: Component values on structured problems



results of the Greedy CTB are very good. There are numerous examples, when the exponent of our bound is about 10 times smaller. This is a success compared to the results on random 3-SAT formulae. The classes of formulae, where our estimation is much smaller are *difp_w* (Factorization), *difp_a* (Factorization) and *parity* (Parity games). Formulae with small differences are *qg* (Quasigroup) and *hole* (Pidgeon hole).

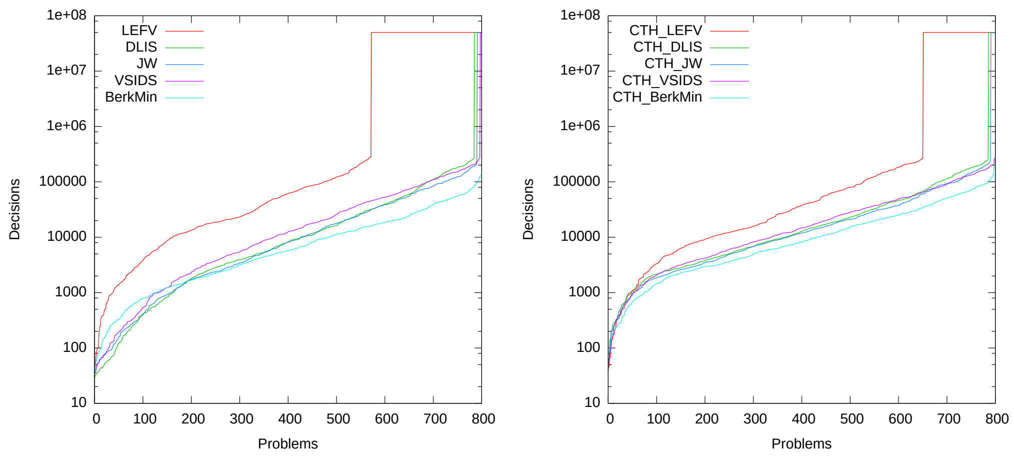
The DFS Find algorithm is again worse than the Greedy CTB.

5.3 Heuristics on Random Formulae

In the previous section we computed component values of formulae, now we are going to solve them. We will measure the number of decisions required to solve a formula. It is more convenient than measuring time, since it does not depend on the properties of the computer, the qualities of the compiler or the programming language. If we run the same solver on a formula two times, the number of decisions is equal, while the measured time is very often different. On the other hand, the number of decisions tells us nothing about the effectiveness of the implementation. For example, when comparing heuristics, the measurement of decisions does not reveal the slowness of the heuristic computation.

The described experiments were conducted on a computer with an Intel Core 2 Quad (Q9550 @ 2.83GHz, 6144KB cache) processor and 3GB main memory. We used the Java solver for these experiments. The limit was set to 5 minutes, if the solver with the specified heuristic did not manage to solve the formula in that time, its number of decision for that formula was set to 50 000 000.

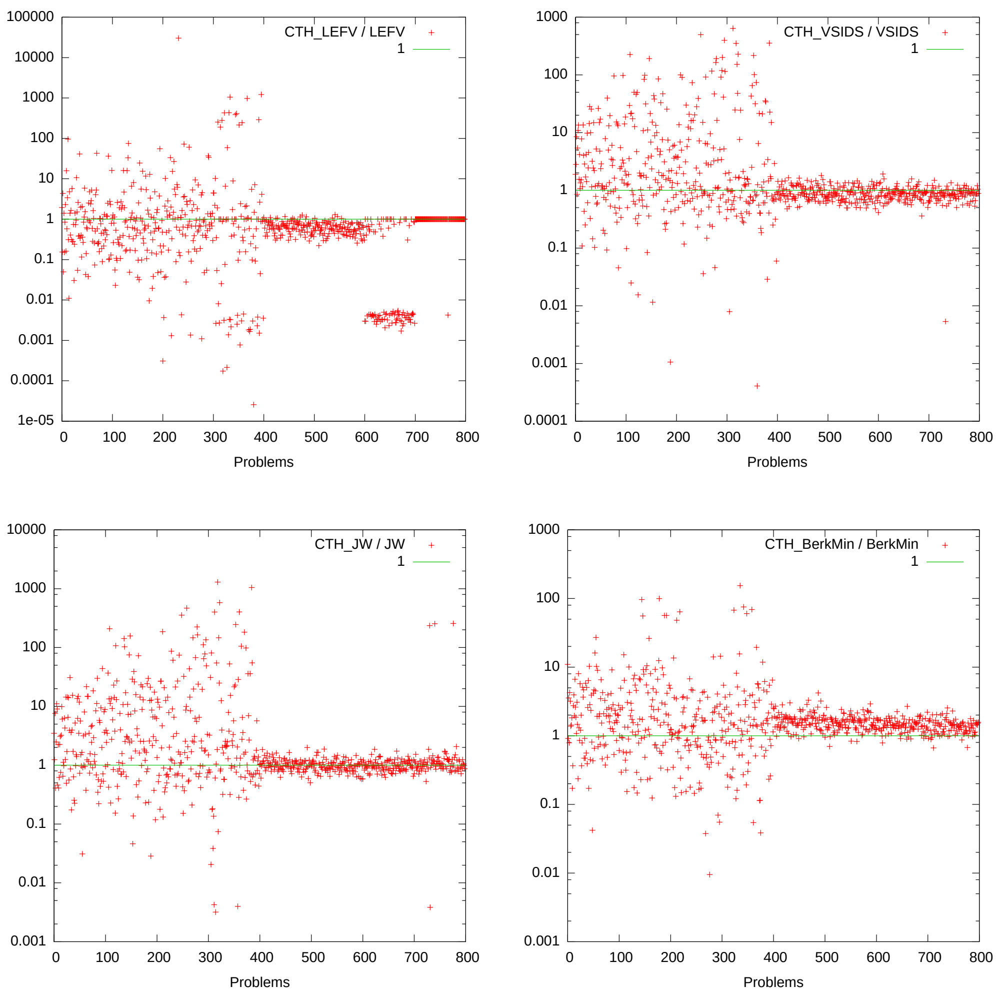
Figure 5.3: Heuristics on random formulae



We compared the 5 described heuristics and their combinations with the CTH - ten heuristics altogether. The phase saving heuristic for phase selection was used in each case. The results were sorted by the number of decisions for each heuristic and displayed on figure 5.3. We can see, that among the base heuristics LEFV is the weakest and BerkMin is the best. The other 3 heuristics perform equally well. The results for combined heuristics are analogous. The order of performance is the same. This shows, that the stronger base heuristic we use, the stronger is the combined heuristic.

To compare the performance of basic heuristics with their combined versions, we plotted the ratio of decisions made by the combined and the basic versions. The results are presented on figure 5.4. The problems are in their original order (see table 5.1). Problems 0-399 are satisfiable, problems 400-800 are unsatisfiable.

Figure 5.4: Basic vs combined heuristics on random formulae



From the plots we can see, that LEFV benefited from the combination the most. Especially on the unsatisfiable formulae with 175 variables (problems 600 - 700). Unsatisfiable problems with 200 variables were not solved by either LEFV or CTH_LEFV, that is why the ratio is 1 in that region. Also the performance of VSIDS was improved for the unsatisfiable formulae, but degraded for satisfiable. JW and DLIS had similar results, so we plotted only JW. The ratio is around 1 for the unsatisfiable formulae. For the satisfiable, the majority of the problems was solved faster by the basic version. Finally, BerkMin was clearly better on all formulae in its basic version. So the combination of BerkMin with CTH is apparently a weaker heuristic for these formulae.

Overall, the combined heuristics were often worse than the original ones. Only LEFV and VSIDS were improved on the unsatisfiable instances.

5.4 Heuristics on Structured Formulae

We performed a similar experiment as described in the previous section, but instead of the random formulae we used our structured set of problems. We compared the same heuristics on the same computers. We used phase saving and counted the number of decisions. The Java solver was used and the time limit was set to 5 minutes.

Figure 5.5: Heuristics on structured formulae

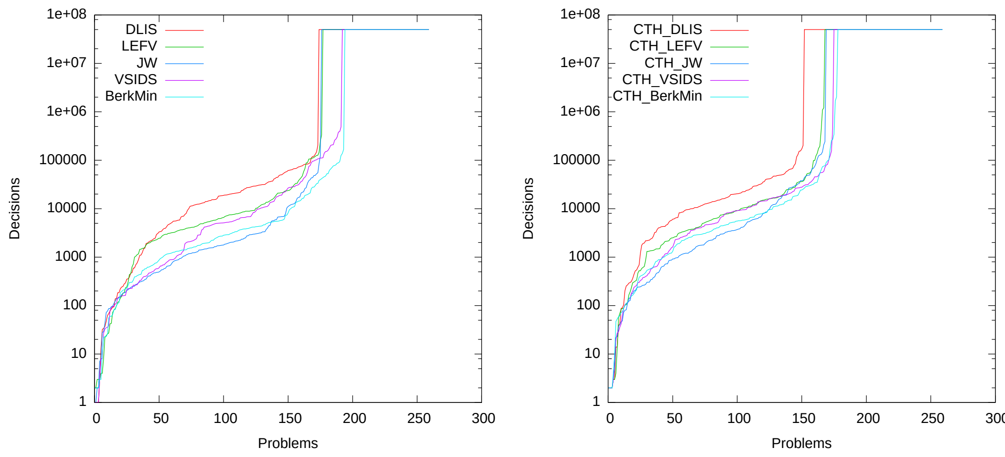
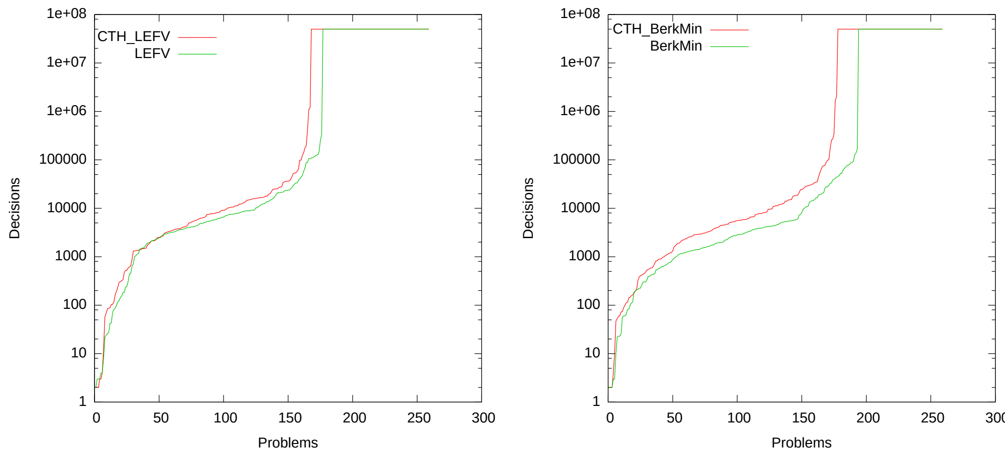


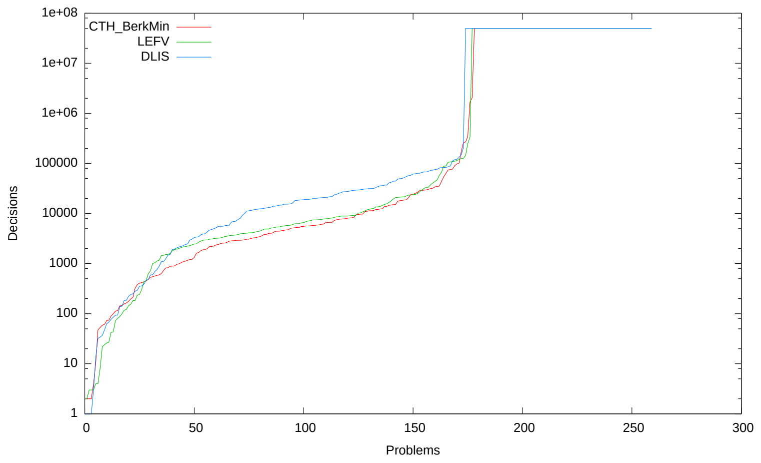
Figure 5.5 shows, that the order of the performance of heuristics is again preserved after the combination with CTH. The DLIS heuristic is the weakest on this set of problems. JW and LEFV are the second weakest. An improvement in the number of solved formulae is produced by VSIDS. BerkMin is again the best. DLIS is left behind especially after the combination with CTH. Considering its high computation cost and poor performance, DLIS is clearly the worst heuristic among the presented ones. The best heuristic is BerkMin. It was the best on both our benchmark sets.

Figure 5.6: Basic vs combined heuristics on structured formulae



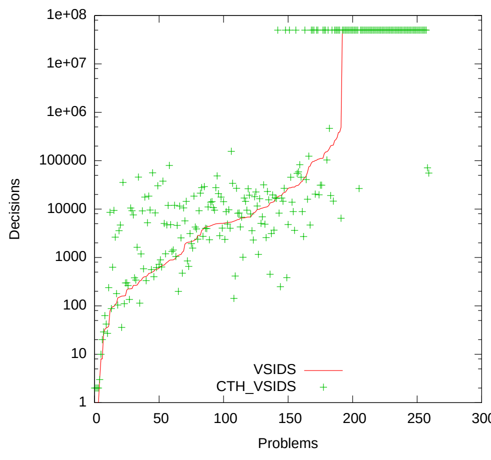
Now, let us compare the basic heuristics with their combined versions. On figure 5.6 we plotted a comparison for LEFV and BerkMin. In both cases the combined version is worse. The difference is greater for BerkMin. Also for the other 3 heuristics, the combined versions are worse with various differences. Although the combination weakens the heuristic, it does not make it that much worse. For example, as figure 5.7 shows, CTH BerkMin is still better than basic DLIS or LEFV.

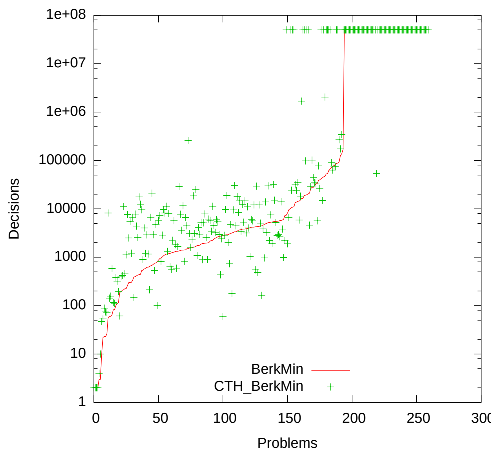
Figure 5.7: CTH BerkMin vs some basic heuristics



Up to this point, we compared the performance of the heuristics in a global sense. We sorted the results by the number of decisions on the entire set of problems and plotted them. If we compare the performance on the individual formulae, the results show, that there is a large number of formulae, where the combined heuristic is better. To visualize these comparisons, we sort the results of one heuristic and plot it as a line. The other heuristic's results are plotted as points in a way, that the number of decisions for the same formulae have the same y coordinates. We compared our best heuristics, VSIDS and BerkMin, with their combined versions. See figure 5.8 for the results. When a point is below the line, then the combined heuristic was better on that formula. Unfortunately, there are no concrete classes of formulae in our benchmark set, where the combined heuristic is always better. It seems to be a random sample of problems where CTH wins.

Figure 5.8: VSIDS and BerkMin basic vs combined





5.5 CTH Strategies Comparison

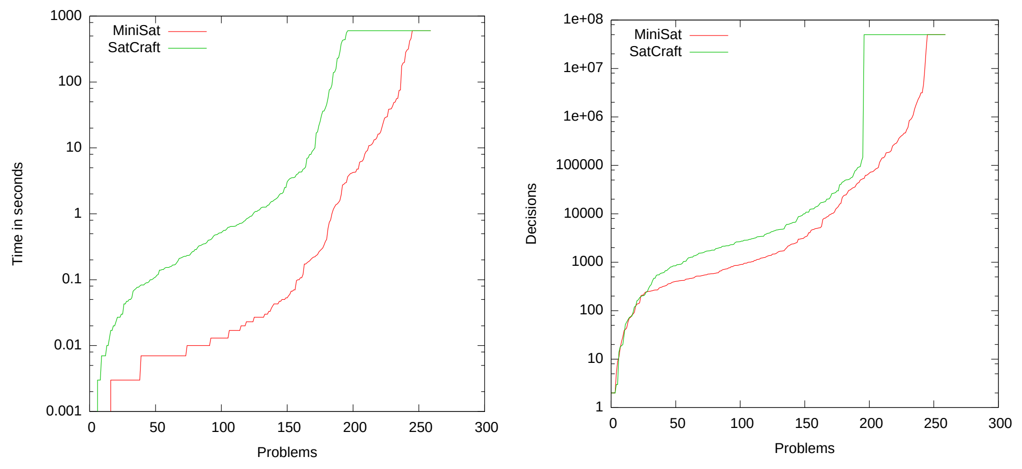
In the previous chapter, when describing the CTH, we mentioned 3 strategies for the ordering of sons in the DFS of the component tree. The strategies were BigFirst, SmallFirst and Random. We theorized, that SmallFirst or Random should be the best.

We conducted experiments to compare the 3 strategies, but the results were very ambiguous. None of the strategies was better than the other. We measured the total number of decisions for the formulae and also counted the number of wins and loses for each possible pair of strategies. The sums were almost equal as well as the number of wins and loses for each strategy. The strategy we selected for the final solver is the random son selection. We did so for the sake of diversification. All the experiments described in the previous and following sections were done with the random son selection strategy.

5.6 The C++ Solver Evaluation

As mentioned before, we also created a C++ implementation of the solver. This solver uses the BerkMin heuristic and phase saving. When compared with the Java implementation, the difference in running speed was not that significant. On small or easy problems, the C++ solver is several times faster than the Java implementation. This can be explained by the startup overhead, which is required for the Java virtual machine initialization. For difficult formulae, which are being solved longer than 1 minute, the difference is minimal.

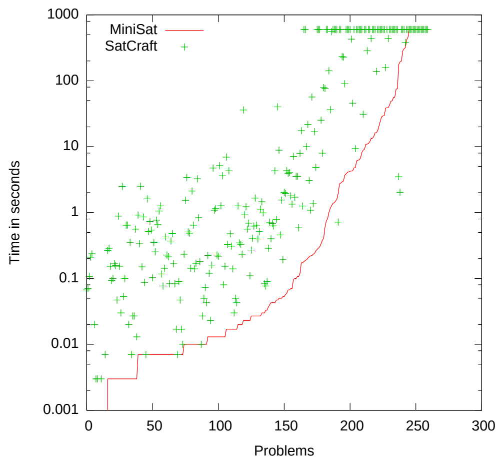
Figure 5.9: MiniSat vs SatCraft global comparison

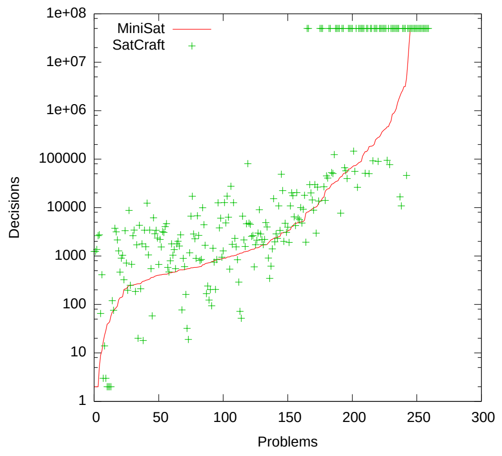


We compared our solver with one of the most famous state-of-the-art SAT solvers - MiniSat[9]. We measured their time and number of decisions on our structured benchmark set. The time limit was set to 10 minutes. If a solver did not manage to solve a formula in that time, then the number of decisions was set to 50 000 000.

On figure 5.9 we did a global comparison for time and number of decisions. MiniSat is apparently better in both categories. The difference is more significant for time. This shows, that MiniSat is implemented much more efficiently. Indeed, MiniSat uses a lot of low level so-called speed hacks. All data structures and procedures are highly optimized. But MiniSat is also better in the number of decisions. This is due to additional techniques employed by MiniSat, which our solver does not implement. These are for example conflict clause minimization[24] and effective preprocessing through variable and clause elimination[8].

Figure 5.10: MiniSat vs SatCraft individual comparison





We also compared the solvers on the individual formulae. Figure 5.10 shows, that there are 3 formulae, where SatCraft outperformed MiniSat in speed. This is very weak, but if we compare the number of decisions, the results are much better.

The reason, why we included this comparison with MiniSat is the following. Although our solver is not a competition for the top solvers, the difference is not unconquerable. With a more efficient implementation and a better setting of the solver's constants, we could probably reach the level of the best current SAT solvers. To set the solver's constants properly, extensive experiments are required. These constants are for example the restart interval, learned clauses limit and the growth rate of these values. More about the constants and solver implementation is written in appendix A.

Chapter 6

Conclusion

Solving hard problems by decomposing them into smaller ones and dealing with them separately is not a new idea. It is called the divide and conquer strategy. For some problems, like sorting, the decomposition is trivial. For others, like SAT, it is not. It this thesis, an attempt was made to formalize, what does it mean to decompose SAT in a good way. We defined a new term - component trees. We described some of its properties and proposed algorithms for its construction.

We showed, how the quality of the component tree can give us an upper bound for the number of decisions required to solve a formula. Also some other possible applications of component trees were suggested.

We implemented a SAT solver using the state-of-the-art algorithms and did extensive experiments to compare various decision heuristics. Some of these were well known existing heuristics, but we also introduced and tested new ones. The new heuristics were based on component trees. They did not manage to outperform the best known heuristics in a global sense, but there were several examples of formulae, where they succeeded.

6.1 Future Work

There are still many concepts, that would probably improve our solver, which we did not implement. Also the component tree idea could be furthered in several ways. One is to find algorithms, which can construct better component trees faster. Another is to somehow redefine the component tree, so it would consider unit propagation.

A very actual issue is parallelization. It is nowadays common to have multicore processors, but current SAT solvers still does not take advantage of them properly. “Multithreading is everywhere except in our solvers” is readable on the website of 2009 SAT Competition[6]. Component trees could be useful in this area of research.

Bibliography

[1] Fadi Aloul. Sat benchmarks. `http://www.aloul.net/benchmarks.html`, 2010.

[2] Tomas Balyo and Pavel Surynek. Efektivni heuristika pro sat zalozena na znalosti komponent souvislosti grafu problemu. *Proceedings of the Conference Znalosti 2009*, pages 35–46, 2009.

[3] Paul Beame, Henry A. Kautz, and Ashish Sabharwal. Understanding the power of clause learning. In Georg Gottlob and Toby Walsh, editors, *IJCAI*, pages 1194–1201. Morgan Kaufmann, 2003.

[4] Armin Biere and Carsten Sinz. Decomposing sat problems into connected components. *JSAT*, 2(1-4):201–208, 2006.

[5] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the really hard problems are. In *IJCAI*, pages 331–340, 1991.

[6] SAT 2009 Conference. Sat competition 2009 website. `http://www.satcompetition.org/2009/`, 2010.

[7] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC*, pages 151–158, 1971.

[8] Niklas Eén and Armin Biere. Effective preprocessing in sat through variable and clause elimination. In *In proc. SAT'05, volume 3569 of LNCS*, pages 61–75. Springer, 2005.

[9] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[10] Olivier Friedman. Sat benchmarks stemming from an analysis of parity games. `http://www2.tcs.ifi.lmu.de/~friedman/`, 2010.

[11] W.I. Gasarch. The p=?np poll. *SIGACT NEWS*, 2002.

[12] Eugene Goldberg and Yakov Novikov. Berkmin: A fast and robust sat-solver. *Discrete Applied Mathematics*, 155(12):1549–1561, 2007.

[13] Holger H. Hoos and Thomas Stutzle. Satlib: An online resource for research on sat. pages 283–292. IOS Press, 2000.

[14] Matti Järvisalo. Unsatisfiable cnfs encoding the problem of equivalence checking two hardware designs for integer multiplication. `http://www.tcs.hut.fi/~mjj/benchmarks/`, 2010.

[15] Robert G. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Ann. Math. Artif. Intell.*, 1:167–187, 1990.

[16] Roberto J. Bayardo Jr. and Robert Schrag. Using csp look-back techniques to solve real-world sat instances. In *AAAI/IAAI*, pages 203–208, 1997.

[17] Henry A. Kautz and Bart Selman. Planning as satisfiability. In *ECAI*, pages 359–363, 1992.

[18] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *DAC*, pages 530–535. ACM, 2001.

[19] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In João Marques-Silva and Karem A. Sakallah, editors, *SAT*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299. Springer, 2007.

[20] John Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes).* Elsevier and MIT Press, 2001.

[21] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, second edition, 2003.

[22] Lawrence Ryan. Efficient algorithms for clause-learning sat solvers, 2004.

[23] João P. Marques Silva and Karem A. Sakallah. Grasp - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996.

[24] Niklas Sörensson and Armin Biere. Minimizing learned clauses. In *SAT '09: Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, pages 237–243, Berlin, Heidelberg, 2009. Springer-Verlag.

[25] G. Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic*, pages 115–125, 1968.

[26] Alasdair Urquhart. Hard examples for resolution. *J. ACM*, 34(1):209–219, 1987.

[27] Rutgers University USA. Satisfiability suggested format. `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/`, 2010.

[28] Miroslav N. Velev and Randal E. Bryant. Effective use of boolean satisfiability procedures in the formal verification of superscalar and vliw microprocessors. *J. Symb. Comput.*, 35(2):73–106, 2003.

[29] Ke XU. Forced satisfiable csp and sat benchmarks of model rb. `http://www.nlsde.buaa.edu.cn/~kexu/benchmarks/benchmarks.htm`, 2010.

[30] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.

Appendix A

Solver Implementation Details

Now we briefly describe some implementation details of our solvers. The full source code is available on the enclosed CD. Basically, we implemented the CDCL DPLL algorithm as it was described in section 2.4. How some of the key procedures are implemented is described below.

Unit propagation was implemented using the 2 watched literals scheme[18], which works the following way. In each clause we watch 2 unassigned literals. This is possible until a clause is neither unit nor satisfied. When we perform unit propagation for a new assignment, normally we visit each clause, where the negation of the assignment literal occurs to test, if it has become unit. Thanks to 2 watched literals, we only need to visit clauses, in which our literal is watched. If our literal is in a clause but is not watched, then that clause contains at least 2 other unassigned literals (watched literals), so it is surely not unit. If the assigned literal is watched in a clause, and that clause is still not unit, we select an other literal to be watched. The 2 watched literals scheme brings a great improvement of propagation speed. We use it for clauses longer than 2. Binary clauses are treated specially, which brings further speedup and some memory conservation.

We used restarting in our solvers. The restart interval is initially set to 1000. This means, that after the first 1000 decisions, the solver is restarted. The learned clauses are preserved, but all assignments with decision levels higher than 0 are removed. After each restart, the restart interval is increased by 20%. So the second restart happens 1200 decisions later, the third 1440 decision after the second restart and so on.

For clause learning we used the first UIP scheme[30]. Now we describe our clause deletion strategy. The initial limit for the number of learned clauses is set to twice the number of original clauses. When the limit is reached, some of the learned clauses are removed and also the limit is increased by 50%. Now we describe which learned clauses are removed when the limit is reached. For each learned clause we count how many times they were used to deduce an assignment. This values are called hits. When clause deletion is required, we take the learned clauses in the order of their age. We start with the oldest. If a clause is longer than 3 literals and has less than 3 hits, it is removed. Otherwise its number of hits is halved. We keep deleting clauses until one half of the learned clauses is deleted. This strategy prefers young short clauses with many recent hits.

The reader surely noticed, how many constants are involved in a SAT solving algorithm. Their values are very significant for the performance of the solver and their proper setting is a hard task. Our constants were set more or less randomly. Some short experiments were done on randomly generated problems to test different values, but by far not enough to set the constants properly.