Solving 3-SAT

Radboud University Nijmegen

Bachelor Thesis

Author:
Peter Maandag
s3047121

Supervisors:
Henk Barendregt
Alexandra Silva

July 2, 2012

Contents

1 Introduction 2
1.1 Problem context . . . 2
1.2 Research questions . . . 3

2 What is 3-SAT? 4
2.1 Definitions . . . 4
2.2 3-SAT solving paradigms . . . 5

3 The Davis, Logemann and Loveland algorithm 5
3.1 The Davis-Putnam algorithm . . . 6
3.1.1 Description . . . 6
3.1.2 Examples . . . 9
3.1.3 Complexity . . . 10
3.2 The Davis, Logemann and Loveland algorithm . . . 11
3.2.1 Description . . . 11
3.2.2 Clause Learning . . . 14
3.2.3 Heuristics . . . 15
3.2.4 Variable State Independent Decaying Sum (VSIDS) . . . 15
3.2.5 MOMs heuristics . . . 17
3.2.6 Complexity . . . 18

4 Binary Decision Diagrams 19
4.1 Description . . . 19
4.2 Solving 3-SAT with BDDs . . . 19
4.3 Optimization . . . 21
4.4 Complexity . . . 22

5 On solving 3-SAT using set theory 23
5.1 Definition . . . 23
5.1.1 Defining valuations . . . 23
5.1.2 Defining Clauses . . . 24
5.1.3 Simplifying sets . . . 25
5.1.4 Defining clauses in simpler set notation . . . 27
5.2 Deciding Satisfiability . . . 27
5.2.1 An algorithm to count shared elements of clauses . . . 28
5.2.2 Complexity . . . 29
5.2.3 Properties of 3-SAT formulas . . . 31
5.3 BDDs versus adapted set-theory . . . 31

6 Conclusion 32

Bibliography 34

1 Introduction

1.1 Problem context

Many problems in daily life, science, and technology can be translated into a mathematical form. In the process of solving these problems, computers can be of great help. But in order to make this possible one needs to find algorithms to solve these mathematical problems. In some cases, such as the halting problem, it is proven that there simply do not exist any algorithmic solutions. In other cases, such as the Towers of Hanoi, there are only inefficient algorithms, making the problem essentially exponential. In the most relevant cases there are efficient (polynomial) solutions, for example for the sorting of a list or searching in an ordered list.

However, there is a mathematical problem lying on the borderline that is the main subject of this thesis: Satisfiability-Solving (SAT-solving). It is not known whether there exists an efficient algorithm for this problem. Also there is no proof that the problem is essentially exponential.

The question whether there exists an efficient algorithm for SAT-solving is considered one of the most important problems in computer science and has been widely studied.[1, 6, 4] This is because SAT-solving algorithms are the only known algorithms that solve several applications in Artificial Intelligence, computational security, software verification and many more.[1] Thus with these algorithms an entire class of believed to be very hard problems (NP-Complete problems) can be solved. Hardware verification is maybe the prime example that uses satisfaction technologies at an industrial scale. A constraint satisfaction problem (CSP) that describes properties of the hardware can be formulated and written as a SAT instance to test for hardware validity. For this reason, many companies are performing intensive Research and Development on satisfiability algorithms.[19]

There are many different algorithms found in the literature, of which all are known to have a worst-case exponential runtime complexity. As of today it remains unknown whether there exist polynomial time algorithms for this problem. Because of its many possible applications the problem is considered very important and a one million dollar prize will be awarded to the person who can settle its algorithmic complexity.[5]

1.2 Research questions

After describing the 3-SAT problem, I will approach it from various perspectives and discuss several algorithms that solve it. Two of them are found in the literature and one is introduced by myself. The complexities of these algorithms will be compared. During the thesis the following research questions will be in mind.

1. What is 3-SAT?
2. How do the important state of the art algorithms to solve 3-SAT work? (Davis, Logemann and Loveland, BDDs)
   (a) What is their description?
   (b) What is their complexity?
3. How can 3-SAT be solved in a different way?
   (a) What is the description of a new algorithm?
   (b) What is the complexity of this algorithm?
   (c) How does it compare against other algorithms in terms of complexity?

2 What is 3-SAT?

In this section I will introduce the main concepts used in this thesis.
First, the definitions of SAT and k-CNF will be given, and after this 3-SAT is explained.

2.1 Definitions

Propositional satisfiability (SAT) is the problem of deciding whether it is possible for a given propositional Boolean formula ϕ to evaluate to *true*. A formal definition of the concepts involved is as follows.

$$\begin{array}{lcl} \langle \textit{Formula} \rangle & ::= & \langle \textit{Literal} \rangle \\ & | & \neg \langle \textit{Formula} \rangle \\ & | & \langle \textit{Formula} \rangle \vee \langle \textit{Formula} \rangle \\ & | & \langle \textit{Formula} \rangle \wedge \langle \textit{Formula} \rangle \\ & | & (\langle \textit{Formula} \rangle) \\ \langle \textit{Literal} \rangle & ::= & \langle \textit{Variable} \rangle \mid \neg \langle \textit{Variable} \rangle \\ \langle \textit{Variable} \rangle & ::= & x_{\langle \textit{Digit} \rangle} \\ \langle \textit{Digit} \rangle & ::= & [0-9]^+ \end{array}$$

A propositional formula ϕ is composed of variables, denoted by $x_0, \ldots, x_{n-1}$ for any $n \in \mathbb{N}$. These variables can be given any value from the set $\{0, 1\}$, representing false and true respectively. A variable is called a *free* variable if it has not yet been assigned a truth value. Furthermore a propositional Boolean formula can contain the Boolean connectives $\wedge$ (AND, conjunction), $\vee$ (OR, disjunction), $\neg$ (NOT, negation) and it can also contain parentheses to indicate priority. A function that maps each variable to a value $\in \{0, 1\}$ is called a *valuation*. The formula is said to be *satisfiable* if there exists a valuation v such that the entire formula evaluates to *true*. The formula is *unsatisfiable* if such a valuation does not exist.

Every propositional formula can be written in *conjunctive normal form* (CNF). The formal definition is as follows.

$$\begin{array}{lcl} \langle \textit{Clause} \rangle & ::= & \langle \textit{Literal} \rangle \mid \langle \textit{Literal} \rangle \vee \langle \textit{Clause} \rangle \\ \langle \textit{CNF} \rangle & ::= & \langle \textit{Clause} \rangle \mid (\langle \textit{Clause} \rangle) \wedge \langle \textit{CNF} \rangle \end{array}$$

A propositional formula is said to be in CNF if it is a conjuntion of one or more *clauses*. A *clause* is a disjuction of one or more *literals* $l \in \mathbb{L}$, the set of all literals. Each element $l \in \mathbb{L}$ is either a variable x or its negation $\neg x$. An example of a propositional formula ϕ in CNF form is the following.

$$\phi = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$$

A CNF-formula of which each clause contains at most k different literals is said to be a k-CNF formula. E.g. the above formula is a 2-CNF formula.

3-SAT is the short notation for 3-CNF satisfiability. The 3-SAT problem asks whether there is a valuation for a 3-CNF formula that evaluates the formula to *true*, i.e. it asks if a given 3-CNF formula is satisfiable.

2.2 3-SAT solving paradigms

The problem of SAT-solving can be approached in various ways. One can think about the problem from the perspective of the variables. This leads to search-algorithms that try to find correct variable instantiations, such as the Davis-Logemann-Loveland (DLL) algorithm[2, 3], which will be extensively described in this thesis. Another way is to think about the problem from the perspective of the constraints and derive a solution set from the clauses, which can be done by constructing Binary Decision Diagrams (BDDs).[23, 24] These are studied and explained in the second part of the thesis. Finally, one can look at the complete formula that describes the SAT instance and try to prove the property of satisfiability from it. In the last part of the thesis I will attempt to do this by translating the problem to set theory and prove that the resulting solution set for a given instance is either empty or must contain elements.

3 The Davis, Logemann and Loveland algorithm

The simplest but also the most time consuming idea to solve a 3-SAT problem is to try every possible valuation until a satisfying one is found or there are no more valuations. This approach is called brute-force solving and is guaranteed to find a solution. It is also guaranteed to have a very slow run-time performance, because in the worst case each valuation of the 2^n possibilities has to be tested. But what if one can search through this exponential space of valuations faster?

This is exactly what the conflict-driven, resolution based Davis-Logemann-Loveland (DLL) algorithm tries to do. It can be implemented and optimized in many different ways and forms the basic framework for many successful SAT-solving algorithms.[2, 3] Therefore, I will spend a great deal of my thesis on explaining this algorithm in detail. DLL however has been so extensively studied and extended that I cannot explain all of its aspects in a limited amount of time. I will try to explain the major aspects of the framework and give several examples to illustrate the idea behind them. While the main idea of DLL is search to find a solution, the roots of the DLL framework are found in the conflict-driven resolution based Davis-Putnam algorithm, which later evolved into the DLL framework that is used today. Explaining the Davis-Putnam algorithm will be the first step to explaining DLL.

3.1 The Davis-Putnam algorithm

The original Davis and Putnam (DP) algorithm for deciding satisfiability of propositional formulas via a resolution proof dates back to the 1960's.[8] 2 years after this was published Davis, Logemann and Loveland presented a modified version of this algorithm,[7] which is now still a widely used algorithm to solve satisfiability. The original DP algorithm uses resolution to solve the problem, but suffers greatly from exponential memory use, because this algorithm builds up an exponential set of clauses to verify satisfiability.

3.1.1 Description

The algorithm known as the Davis-Putnam procedure[10, 9] is described in Algorithm 1. The algorithm works on a propositional formula ϕ which is in CNF form. The algorithm treats ϕ as a set of clauses that can be manipulated. There are some rules that define the algorithm, which I've denoted with I, II and III.

Rule I deals with simplifying the formula or determining satisfiability by analyzing one-literal clauses. There are three cases to be distinguished.
Case 1: an atomic formula p occurs as a one-literal clause in both positive and negated form. In this case the resulting formula ϕ is unsatisfiable, because the empty clause can be deduced by means of resolution.
Case 2: Case 1 does not apply and an atomic formula p occurs as a one-literal clause. Then all clauses that contain p can be deleted and $\neg p$ can be removed from the remaining clauses.
Case 3: Case 1 does not apply and an atomic formula $\neg p$ occurs as a one-literal clause. Then all clauses that contain $\neg p$ can be deleted and p can be removed from the remaining clauses. In modern literature this rule is now usually referred to as the Unit Propagation rule.

Rule II deals with redundant clauses. In modern literature this is usually referred to as the Pure Literal rule and nowadays done as a preprocessing step for efficiency reasons.[12] If *only* p or *only* $\neg p$ occurs in the formula ϕ then all clauses which contain p or $\neg p$ may be deleted.

Rule III uses the inference rule known as resolution in mathematical logic. Resolution can be applied to two clauses that have complementary literals L and L', such that $L \equiv \neg L'$. Via a resolution step the two clauses can be combined in a single clause, called the *resolvent* that contains all the literals without L and L'.

In general this means that $(C \vee L) \wedge (C' \vee L')$ is reduced to $(C \vee C')$, with C and C' the remainder of the clause that contains L and L' respectively. Resolution may be applied at only one complementary literal pair at once and repeating literals in the result may be removed.

Intuitively, the validity of this rule can be explained by the fact both clauses that contain C and C' have to be true in order to satisfy the instance. If the resolvent is true, then this means that at least C or C' has to be true, thus one of the two clauses is satisfied. In either case you can initialize the variable that belongs to L and L' in such a way that the other clause is satisfied as well. Therefore, L and L' can be eliminated in the resolvent.

Algorithm 1 The Davis-Putnam Procedure

```
procedure DP($\phi$)                                         ▷ Propositional formula $\phi$
    PreProcess($\phi$)                                      ▷ Translate $\phi$ to CNF form
    while true do
        while true do
            Remove one-literal clauses                                      ▷ Rule I
            if consistent($\phi$) $\neq$ UNDETERMINED then
                break
            end if
            Remove atomic formula's p which are only negative or only
            positive                                                        ▷ Rule II
            if consistent($\phi$) $\neq$ UNDETERMINED then
                break
            end if
        end while
        Pick a variable $x$
        Replace clauses which contain $\neg x$ and $x$ with the set of clauses
        obtained by resolution                                              ▷ Rule III
        if consistent($\phi$) $\neq$ UNDETERMINED then
            break
        end if
    end while
end procedure

function CONSISTENT($\phi$)                                             ▷ CNF formula $\phi$
    if empty clause is deduced then
        return UNSATISFIABLE
    end if
    if $\phi == \emptyset$ then
        return SATISFIABLE
    end if
    return UNDETERMINED                                ▷ Further deduction is needed
end function
```

3.1.2 Examples

Next, I will show two examples which illustrate how the algorithm works. The first example operates on an unsatisfiable formula ϕ.

Example 3.1.
Let the following CNF formula ϕ be given.

$$\begin{aligned}\phi = &(\neg a \vee b \vee d) \wedge (\neg b \vee c \vee d) \wedge (a \vee \neg c \vee d) \wedge (a \vee \neg b \vee \neg d)\\ &\wedge (b \vee \neg c \vee \neg d) \wedge (\neg a \vee c \vee \neg d) \wedge (a \vee b \vee c) \wedge (\neg a \vee \neg b \vee \neg c)\end{aligned}$$

This formula is unsatisfiable. The DP procedure will make the following deductions to show this.

$$\phi$$

$$\Big\Downarrow \text{ Use De Morgan's law to separate } a \text{ and } \neg a$$

$$\begin{gathered}[((b \vee d) \wedge (c \vee \neg d) \wedge (\neg b \vee \neg c)) \vee \neg a] \wedge\\ [((\neg c \vee d) \wedge (\neg b \vee \neg d) \wedge (b \vee c)) \vee a] \wedge\\ (\neg b \vee c \vee d) \wedge (b \vee \neg c \vee \neg d)\end{gathered}$$

$$\Big\Downarrow \text{ Rule III, elimination of } a$$

$$\begin{gathered}(b \vee \neg c \vee d) \wedge (b \vee c \vee d) \wedge (\neg b \vee c \vee \neg d) \wedge\\ (b \vee c \vee \neg d) \wedge (\neg b \vee \neg c \vee d) \wedge (\neg b \vee \neg c \vee \neg d)\\ (\neg b \vee c \vee d) \wedge (b \vee \neg c \vee \neg d)\end{gathered}$$

$$\Big\Downarrow \text{ Use De Morgan's law to separate } b \text{ and } \neg b$$

$$\begin{gathered}[((\neg c \vee d) \wedge (c \vee d) \wedge (c \vee \neg d) \wedge (\neg c \vee \neg d)) \vee b] \wedge\\ [((c \vee \neg d) \wedge (\neg c \vee d) \wedge (\neg c \vee \neg d) \wedge (c \vee d)) \vee \neg b]\end{gathered}$$

$$\Big\Downarrow \text{ Rule III, elimination of } b$$

$$(\neg c \vee d) \wedge (c \vee d) \wedge (c \vee \neg d) \wedge (\neg c \vee \neg d)$$

$$\Big\Downarrow \text{ Use De Morgan's law to separate } c \text{ and } \neg c$$

$$[(d \wedge \neg d) \vee \neg c] \wedge [(d \wedge \neg d) \vee c]$$

$$\Big\Downarrow \text{ Rule III, elimination of } c$$

$$d \wedge \neg d \Rightarrow \text{inconsistency}$$

Example 3.2.
Let the following CNF formula ϕ be given.

$$\begin{aligned}\phi = &(\neg a \vee b \vee d)\ \wedge (\neg b \vee c \vee d)\ \wedge (a \vee \neg c \vee d)\ \wedge (a \vee \neg b \vee \neg d)\\ &\wedge (b \vee \neg c \vee \neg d)\ \wedge (\neg a \vee c \vee \neg d)\ \wedge (a \vee b \vee c)\end{aligned}$$

This formula is satisfiable, because it has only seven clauses (later, this will be proven via Lemma 5.3). The DP procedure will make the following deductions to deduce unsatisfiability.

$$\phi$$

$$\Downarrow \text{Use De Morgan's law to separate } a \text{ and } \neg a$$

$$\begin{gathered}[((b \vee d)\ \wedge (c \vee \neg d)\ \vee \neg a] \wedge\\ [((\neg c \vee d)\ \wedge (\neg b \vee \neg d)\ \wedge (b \vee c))\ \vee a] \wedge\\ (\neg b \vee c \vee d)\ \wedge (b \vee \neg c \vee \neg d)\end{gathered}$$

$$\Downarrow \text{Rule III, elimination of } a$$

$$\begin{gathered}(b \vee \neg c \vee d)\ \wedge (b \vee c \vee d)\ \wedge (\neg b \vee c \vee \neg d) \wedge\\ (b \vee c \vee \neg d)\ \wedge (\neg b \vee c \vee d)\ \wedge (b \vee \neg c \vee \neg d)\end{gathered}$$

$$\Downarrow \text{Use De Morgan's law to separate } b \text{ and } \neg b$$

$$\begin{gathered}[((\neg c \vee d)\ \wedge (c \vee d)\ \wedge (c \vee \neg d)\ \wedge (\neg c \vee \neg d))\ \vee b] \wedge\\ [((c \vee \neg d)\ \wedge (c \vee d))\ \vee \neg b]\end{gathered}$$

$$\Downarrow \text{Rule III, elimination of } b$$

$$(c \vee d)\ \wedge (c \vee \neg d)$$

$$\Downarrow \text{Rule II}$$

$$\text{consistent}$$

3.1.3 Complexity

The fact that this is an exponential algorithm follows directly from the fact that regular resolution is exponential. This means that this algorithm will generate an exponential number of clauses and thus is very memory inefficient. The lower bound complexity of this algorithm has been proven to be 2^{cn} for some constant $c > 0$.[10]

3.2 The Davis, Logemann and Loveland algorithm

In 1962 M. Davis, G. Logemann and Donald Loveland introduced an improved version of the DP algorithm which nowadays serves as the most important framework for many SAT-Solving algorithms.

In the improved version of the algorithm rule III was replaced by the Splitting Rule[7] in order to limit the amount of memory used during runtime. The Splitting Rule instantiates a variable and then continues to search for conflicting clauses without applying resolution. In this case there will no longer be an exponential amount of clauses generated; only clauses that have been satisfied will be deleted. Depending on the specific implementation of the DLL algorithm the memory requirements are usually predictable and it becomes only run-time limited.[1]

3.2.1 Description

The (basic) DLL algorithm is now used in the following way and can be nicely described as a recursive function described in Algorithm 2. Just as in the DP algorithm, ϕ is treated as a set of clauses that can be manipulated. Modern

Algorithm 2 The Davis Logemann Loveland Algorithm (Recursive)

```
function DLL($\phi$)                                  ▷ CNF formula $\phi$
    formulaDeduction()        ▷ Rule II (simplify formula / Pure Literals)
    UnitPropagate()                                               ▷ Rule I
    if $\phi == \emptyset$ then
        return SATISFIABLE
    else if empty clause deduced then
        return UNSATISFIABLE
    end if
    Pick an unassigned variable $x$ in $\phi$               ▷ Splitting Rule
    $x := 0$
    if DLL($\phi$) then                        ▷ DLL($\phi$) == SATISFIABLE
        return SATISFIABLE
    end if
    $x := 1$                                       ▷ Implicit backtracking
    if DLL($\phi$) then
        return SATISFIABLE
    end if
    return UNSATISFIABLE
end function
```

versions of the DLL algorithm include a lot of additions and modifications.

- The simplifying of the formula is now usually done as a preprocessing step for efficiency reasons.[12]
- Heuristics for the branching variable are used.[13]
- Backtracking has been replaced by backjumping or non-chronological backtracking in iterative versions of DLL, so that similar parts of the same tree are not searched over and over again.[14]
- New clauses can be learned during the search process to increase the amount of branches that can be cut away from the search tree later.[11]
- Improvements on Boolean Constraint Propagation algorithms are introduced.[11]
- Random restarts of the algorithm that make use of previously learned information increase the chances of coming to a solution faster.[15]

For these improvements to be efficient and also to allow non-chronological backtracking the DLL algorithm is usually implemented iteratively. An example of an algorithm based on the DLL framework that implements all these features is the popular zChaff solver algorithm[2], which is described in Algorithm 3.

DLL uses depth-first search as main tool to find a satisfying assignment to a problem. At each step, the algorithm picks a variable x and assigns a value to x. Each assigned variable x also has a decision level associated with it, which equals the level of the depth first search tree at which the decision was made. The decision level starts at 1 for the first decision (i.e. at the root node of the search tree) and is incremented each time a new variable is instantiated.

After this the formula is simplified in the same way as in the DP algorithm by using Boolean Constraint Propagation. This is done by the function deduce(). If deduce() detects a conflicting clause then the rest of the search tree does not have to be searched, because the current assignment cannot lead to a satisfaction of the formula. In this case deduce() will return CONFLICT. Then a reason for the conflict is found by the function analyzeConflict(). The reason for the conflict is obtained as a set of variable assignments from which a clause can be learned, because it implies the current conflict. The solver can add this clause to its database. This function then also determines the branching level to backtrack to. Finally the branching level is updated appropriately by the function backTrack(), which undoes variable assignments and makes sure that the next branch of the search tree is expanded. Backtracking to level 0 indicates that a variable is implied by the CNF formula to be both true and false. Therefore, the formula cannot be satisfiable.

Algorithm 3 The zChaff algorithm using the iterative DLL framework

```
 1: function ZCHAFF($\phi$)                              ▷ CNF formula $\phi$
 2:     Preprocess()           ▷ Rule II (simplify formula / Pure Literals)
 3:     while there exists a free variable in $\phi$ do
 4:         decideNextBranch() ▷ Use advanced heuristics to pick and assign
                                 free variables (Splitting Rule)
 5:         status = deduce()  ▷ Advanced Boolean Constraint Propagation
                                 (Rule I)
 6:         if status == CONFLICT then
 7:             blevel = analyzeConflict()    ▷ decide backtracking level and
                                                learn conflict clauses
 8:             if blevel > 0 then
 9:                 backTrack(blevel)        ▷ resolve conflict, backtrack non-
                                               chronologically
10:             else if blevel == 0 then
11:                 return UNSATISFIABLE            ▷ cannot resolve conflict
12:             end if
13:         end if
14:         runPeriodicFunctions()        ▷ periodic jobs such as restarts,
                                            clause deletion etc.
15:     end while
16:     return SATISFIABLE
17: end function
```

The function runPeriodicFunctions() makes sure that some learned clauses are deleted again to keep memory requirements in bounds and it can also decide to restart the searching process. If this happens, then all the initializations of variables are reset and the analysis is restarted, in which some of the information gained from the previous analysis is included in the new one. As a result, the solver will not repeat the previous analysis after the restart. In addition, randomness can be added to make sure that solver chooses a different path to analyze.

3.2.2 Clause Learning

Recall that using resolution in the DP algorithm was highly memory inefficient, because exponentially many clauses were generated from a 3-SAT formula ϕ. If we think of ϕ as a Boolean function f over variables $x_0, \ldots, x_{n-1}$, then we are only interested in clauses that do not change the corresponding function f. In the case of DP, all the clauses generated through resolution meet this requirement. During clause learning resolution is still used, but to keep memory requirements at a minimum, only clauses that help come to a conclusion about satisfiability faster are learned.[11] In other words: we are searching for new clauses that could have the potential to significantly reduce the search space or tend toward a negative conclusion. During this process, often binary and unary clauses are learned, i.e. clauses with at most two literals, since they put the most restrictions on the solution set. However, long clauses can also be considered important.

Various methods for clause learning have been studied. It can be done as an inefficient preprocessing step, or it can be applied during the solving process, adding clauses to the formula each time a dead end is found. The latter is known as *conflict-driven clause learning.* This technique tries to resolve clauses at each leaf in the search tree using resolution. At each conflict, it resolves only those clauses that were involved in BCP along the path to that leaf. In particular clauses that share literals are evaluated to minimize the length of the resolvent. Longer clauses can be be learned by analyzing each conflict assignment. Suppose the conflicting variable instantiation $(x_0 = 0, x_1 = 1, x_4 = 1, x_6 = 0)$ was encountered. Then a new clause $(x_0 \vee \neg x_1 \vee \neg x_4 \vee x_6)$ can be deduced from it.

Ultimately, the resulting learned clauses are often used as part of a heuristic in order to reduce the search space. This will be discussed in the next subsection.

3.2.3 Heuristics

It is common in algorithms that use search to try to speed up the process by predicting results during execution. In many cases these predictions cause significant performance increases and need for less resources. Such improvement techniques are called heuristics. To give an intuitive idea about what a heuristic can do to the search process, consider the idea of partial evaluation.

Suppose there is a random 3-SAT problem with n variables for which the constraint $c_0 = (x_0 \neq 0 \wedge x_1 \neq 0 \wedge x_2 \neq 0)$ holds. If the algorithm comes to the point where it initializes x_0, x_1 and x_2 to this forbidden combination, it should know that it doesn't have to search through all the 2^{n-3} combinations of the remaining $n-3$ free variables, because it can never result in a satisfying assignment. In this manner, a part of the search tree with 2^{n-3} nodes can be avoided.

Many different heuristics have been proposed in the quest for better DLL based algorithms. I will describe two successful heuristic ideas that are commonly used in the DLL algorithm.

3.2.4 Variable State Independent Decaying Sum (VSIDS)

The Variable State Independent Decaying Sum (VSIDS) is a branching heuristic that is used in zChaff to predict which free variable the DLL algorithm must choose at each step in order to keep the size of the search space mininimal. Below is explained how it works.[2, 15, 1]

A counter is assigned to each literal, which records a score. The counters are initialized to the number of occurrences of a literal in the initial formula. During the search process, the zChaff SAT-Solver learns new clauses, constructed by the function analyzeConflict(), which are added to the clause database during the search process. When these clauses are added to the database, the score associated with each literal in the clause is incremented. When a branching decision is made, the literal with the highest score is chosen and the variable which belongs to this literal is initialized in a way such that the literal is satisfied. If there are multiple literals with the same score the literal is chosen randomly among them. Finally, all the counters are periodically divided by a constant. This has the effect that VSIDS puts more weight on the most recently added clauses. In other words, VSIDS is an attempt to satisfy conflict clauses, but in particular an attempt to satisfy the most recent clauses. It is considered a quasi-static heuristic, because it doesn't depend on the variable state such as MOMs heuristics (discussed in next subsection), but it still gradually changes as new clauses are added; it takes search history into account. The following example shows this property.

Given the following formula clause database ϕ.

$$\phi = (x_1 \vee x_4), (x_1 \vee \neg x_3 \vee \neg x_8), (x_1 \vee x_8 \vee x_{12}), (x_1 \vee x_8 \vee \neg x_{10}), (x_2 \vee x_{11})$$
$$(\neg x_7 \vee \neg x_3 \vee x_9), (\neg x_7 \vee x_8 \vee \neg x_9)$$

Then the initial VSIDS scores are the following:

$4 : x_1$

$3 : x_8$

$2 : \neg x_3, \neg x_7$

$1 : x_2, x_4, \neg x_8, \neg x_9, x_9, \neg x_{10}, x_{11}, x_{12}$

Suppose that a new clause $c_1 = (x_8 \vee x_{10} \vee \neg x_{12})$ is added to ϕ. The VSIDS scores of $\phi \wedge c_1$ then change to the following. You can see that the new clause is not necessarily the first one to be satisfied since it does not contain x_1.

$4 : x_1, x_8$

$2 : \neg x_3, \neg x_7$

$1 : x_2, x_4, \neg x_8, \neg x_9, x_9, \neg x_{10}, x_{10}, x_{11}, \neg x_{12}, x_{12}$

Below are the results of dividing the counters by two after adding c_1. You can see that not much has changed. There are still multiple variables with score two to choose from.

$2 : x_1, x_8$

$1 : \neg x_7, \neg x_3$

$0 : x_2, x_4, \neg x_8, \neg x_9, x_9, \neg x_{10}, x_{10}, x_{11}, \neg x_{12}, x_{12}$

However if we first divide the counters by two (left) and then add the new clause (right), we see that the new clause is the one to be initialized first. The scores are floored if the score was one and rounded up otherwise.

$3 :$	$3 : x_8$
$2 : x_1, x_8$	$2 : x_1$
$1 : \neg x_3, \neg x_7$	$1 : \neg x_3, \neg x_7, x_{10}, \neg x_{12}$
$0 : x_2, x_4, \neg x_8, \neg x_9, x_9, \neg x_{10}, x_{11}, x_{12}$	$0 : x_2, x_4, \neg x_8, \neg x_9, x_9, \neg x_{10}, x_{11}, x_{12}$

3.2.5 MOMs heuristics

Predating VSIDS, another branch of popular heuristics, which are easy and efficient to implement was MOMs.[16, 17] It is called MOMs, because these heuristics prefer the literal having <u>M</u>aximum number of <u>O</u>ccurrences in the <u>M</u>inimum length clauses. This means that only the clauses having minimal length are considered. The idea behind this approach is that these clauses can be considered as more crucial, because there are less possibilities to satisfy them. This means that branching on them will maximize the effect of BCP and increase the possibility of cutting away parts of the search tree or finding a satisfiable assignment.

An example of a MOMs heuristic function was taken from[18] where $f^*(l)$ is the number of occurrences of an open literal l in the smallest non-satisfied clauses of a 3-SAT formula ϕ.

$$[f^*(x) + f^*(\neg x)] * 2^k + f^*(x) * f^*(\neg x) \text{ for some } k \in \mathbb{N}$$

Depending on the value that is chosen for k, a higher preference is given to variables x that occur in a large number of minimum clauses. Also variables that appear more often in both positive and negative literals are ranked higher. When the highest ranked variable is found, it is instantiated to true if the variable appears in more smallest clauses as a positive literal and to false otherwise.

In the following example there are only few satisfying assignments. In fact, ϕ can only be satisfied if x_1, x_2 and x_3 are set to 1. However, if a MOMs heuristic is used in combination with BCP the solution is quickly found.

Given the following clause database.

$$\phi = (x_1 \vee x_2 \vee x_3), (\neg x_1, \neg x_2, x_3), (x_1 \vee x_4), (x_1 \vee \neg x_4), (\neg x_1 \vee x_2), (x_2, x_3)$$

Then the MOMs score for each variable can be calculated, using $k = 2$.

$$x_1 : (2 + 1) * 4 + 2 * 1 = 14$$
$$x_2 : (2 + 0) * 4 + 2 * 0 = 8$$
$$x_3 : (1 + 0) * 4 + 1 * 0 = 4$$
$$x_4 : (1 + 1) * 4 + 1 * 1 = 9$$

This means that x_1 is instantiated to 1 since it occurs twice as a positive literal and once as a negative literal in the smallest clauses. Using BCP, ϕ is then reduced to $(\neg x_2 \vee x_3), (x_2), (x_2 \vee x_3)$. Since x_2 is now the only variable occurring in a unary clause it is chosen to be instantiated to 1 in the next step. Finally, ϕ is reduced to (x_3) after applying BCP and the solution is found in a total of just three steps.

The main disadvantage of this heuristic however is that its effectiveness highly depends on the shape of the CNF formula: if there are little binary or unary clauses, such as in the beginning of the search process, the heuristic provides little guidance for the right choice. And especially in the beginning, the right variable instantiation can dramatically effect the size of the resulting search tree. Therefore, most SAT-solving algorithms have turned to other more successful heuristic ideas.

3.2.6 Complexity

While analyzing the complexity of SAT-solving algorithms, the space of satisfiability problems can be divided into three regions:[20] the so called *under-constrained* problems, the *critically-constrained* problems and the *over-constrained* problems. The *under-constrained* problems are problems with a small number of constraints, which appear to be easy, because they generally have many solutions. Search algorithms such as DLL will have a higher probability of finding a solution. The *over-constrained* problems are problems with a very large number of constraints that also appear to be easy, because intelligent algorithms, such as DLL will generally be able to cut off big parts of the search tree to come to a conclusion.

DLL tends to have fast (some times polynomial) performance on SAT instances of these regions. However, there are also hard problems in between; problems that have few solutions but lots of partial solutions, on which DLL has exponential runtime performance. These problems are called the *critically-constrained* problems and are instances with a clause to variable ratio roughly around 4.26.[21] This point is also referred to as the *crossover point*. This is where DLL performance is the worst, because it takes longer to decide whether a part of the search-tree can be cut off and/or only smaller parts of the search tree can be cut off at the time, yielding only a minor increase in performance.

4 Binary Decision Diagrams

The DLL framework is the most successful and widely used framework to solve satisfiability, but using DLL is not the only way to solve satisfiability. Instead of using search to find a satisfying variable assignment, one can also try to create the set of satisfying assignments by looking at the constraints. This can be accomplished via the use of Reduced Ordered Binary Decision Diagrams (ROBDDs), which allows for an efficient representation of Boolean functions. An algorithm that uses these ROBDDs does not search for a single satisfying truth assignment, but rather a compact symbolic representation of the complete set of satisfying truth assignments. Satisfiability is then determined by checking whether this set is non-empty.[21]

4.1 Description

A BDD represents a formula ϕ (seen as a Boolean function) as a rooted directed acyclic graph $G = (V, T, E)$. Each non-terminal node or vertex $v_i \in V$ is labeled by a variable x and has edges directed towards two successor nodes, called the then-child and the else-child. In a graphical representation, the then-child represents $x = 1$ via an uninterrupted line and the else-child represents $x = 0$ via a dotted line. The lines are depicted as undirected for simplicity, but must always be considered as directed to the next node. Each terminal node $t_i \in T$ is labeled with 0 or 1 and does not have any children. For a given assignment of the variables, the value of the function is found by tracing a path from the root to a terminal vertex following the branches indicated by the values assigned to the variables.[23]

This means that a BDD can also be seen as a decision tree and therefore each variable can only occur once in each path. In the next subsection, examples of BDDs and 3-SAT solving with BDDs will be shown.

4.2 Solving 3-SAT with BDDs

Let us think of a 3-SAT problem in terms of constraints. Each constraint tells us which variable assignments are forbidden and implicitly, shows us a path of variable instantiations that must be followed to make the formula unsatisfiable. These paths can be written down quite intuitively in a BDD. By adding all these constraint paths to the BDD you will end up with the representation of the complete function of which you are checking satisfiability. The process starts with the empty BDD and during construction it keeps adding clauses until the complete formula is captured. The following example shows how it's done.

Suppose we have the following formula $\phi = c_1 \wedge c_2$, with clause $c_1 = (x_0 \vee \neg x_1 \vee x_2)$ and $c_2 = (\neg x_3 \vee \neg x_4 \vee \neg x_5)$.
(In this example I have chosen different variables for $c1$ and $c2$ so that I show an optimization of the resulting BDD in the next section) Then, the construction of a BDD can begin by adding the constraints one by one. The construction is started with the empty BDD, which is always true, because a formula without constraints is always satisfiable.

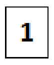

Figure 1: BDD of the empty formula

Then the clause c_1 is added according to the constraints: the clause would evaluate to 0 if $x_0 = 0 \wedge x_1 = 1 \wedge x_2 = 0$. As soon as one of these conditions doesn't hold it satisfies the formula. The resulting BDD is shown in Figure 2.

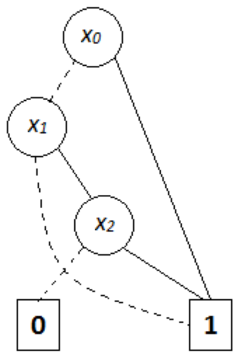

Figure 2: BDD after adding c_1

After this, the rest of the formula is added to the BDD. This means that at each node that points to 1, the additional constraints of the next clause must be added in a similar way as done with c_1 if they haven't been added already. It makes no sense to add the additional constraints to nodes that point to 0, because in that path the formula can only evaluate to 0, because of the previous constraints. The resulting BDD after adding clause c_2 is shown in Figure 3.

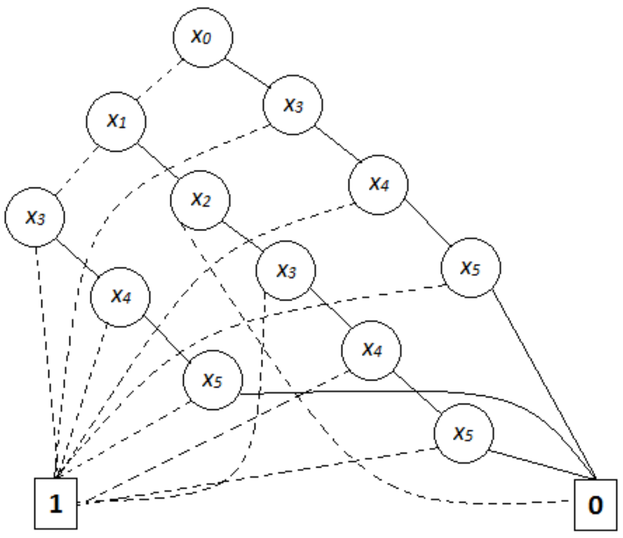

Figure 3: BDD of $\phi = c_1 \wedge c_2$

We can now determine that ϕ is satisfiable because there is a path in the tree that leads to 1. This BDD however, is not optimal because the same subgraph with nodes x_3, x_4, x_5 appears three times. In the next section, I will show how the size of this BDD can be reduced.

4.3 Optimization

When one speaks of BDDs in real world SAT-solving applications, what one usually means are Reduced Ordered Binary Decision Diagrams (ROBDDS). A BDD is called reduced if the following two reduction rules have been applied. The first rule states that any isomorphic subgraphs should be merged. The second rule states that nodes whose children point to an isomorphic subgraph should be eliminated. The BDD is ordered if the nodes are ordered according to some strict ordering $x_0 < \ldots < x_{n-1}$ between each variable.

The reason that one uses ROBDDs is that they have some interesting properties. ROBDDs do not only provide compact representations of Boolean expressions, but there are also efficient algorithms for performing logical operations on them. They are all based on the crucial fact that for any function $f : \mathbb{B}^n \to \mathbb{B}$ there is exactly one ROBDD representing it. This property is called *Canonicity.* Canonicity can be guaranteed by applying the two reduction rules mentioned above, but proving this fact falls out of the scope of this thesis.

The following example shows how the size of a BDD can be reduced by merging isomorphic subgraphs. In the example shown in Figure 3, adding the second clause resulted into adding the same subgraph three times. By adding this graph only once and redirecting the previous nodes to this graph the size of the BDD can be reduced. The result is shown in Figure 4.

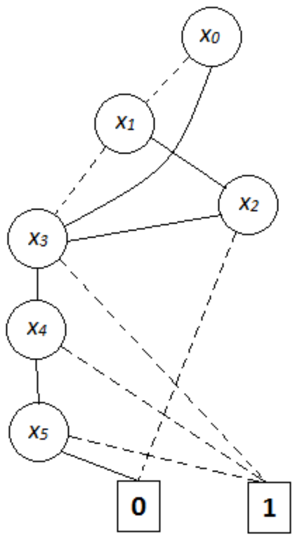

Figure 4: Merged isomorphic subgraphs of Figure 3

This pruning of the decision tree also comes in handy when all branches of a subgraph point to 0 or 1. Then the entire subgraph can be replaced with a terminal node 0 or 1 respectively.

4.4 Complexity

In the earlier discussed problem regions of 3-SAT instances, the crossover point means nothing special for ROBBDs. In fact, in earlier research[21] it is shown that for clause to variable ratios between 0.5 and 15 the running time is exponential in both time and space, but this depends on the algorithm implementation.[25] This is because the running time of ROBDD-based algorithms is determined mostly by the size of the ROBDDs that are being constructed. This explains the fact that ROBDDs perform worse in the *under-constrained* set of problems, because the solution set can be irregular and there are exponentially many solutions to capture in an ROBDD. Also in the *critically-constrained* problems, ROBDDs could perform exponentially, because there can be many partial solutions that blow up the size of the manipulated ROBDD. This leads to the conclusion that ROBDDs perform the best on *over-constrained* problems, where there are not many solutions to capture.

It has also been shown that BDDs perform polynomially on certain problem instances where the performance of DLL is still exponential[24], but for most instances the opposite is true.

5 On solving 3-SAT using set theory

The DLL algorithm described in section 2 is heavily based on searching for the right answer. The use of deduction by resolution (standard proofs) in DLL-based algorithms is limited in order to keep the memory requirements low. BDDs are tackling the problem from a different view point, building up the solution set in order to verify satisfiability. An interesting research question to pose would be: What is needed in existing theories to decide the unsatisfiability of a given formula by means other than standard resolution? While most solvers are resolution based to solve this problem in propositional logic or use exhaustive search to find a solution, I'm going to try to describe the problem in set theory and will investigate what is needed to decide the (un)satisfiability of a given SAT formula.

5.1 Definition

To decide satisfiability using set theory we first need to define how clauses and satisfying variable assignments of a 3-SAT formula are represented as sets.

5.1.1 Defining valuations

For a 3-SAT instance ϕ with n variables the set $\mathcal{V}_\phi$ of possible *valuations* (variable assignments) v is determined for which $v(\phi)$ evaluates to 1.

$$\mathcal{V}_\phi = \{v \in 2^n \mid v(\phi) = 1\}$$

Here v is a mapping $\{x_0 \ldots x_{n-1}\} \to \{0,1\}$ that instantiates every variable to 0 or 1. This means that there are a maximum of $2_1 * 2_2 * \ldots * 2_{n-1} * 2_n$ different valuations that could satisfy ϕ. So each function v_i can be captured as a unique number in binary representation, which is captured in the set 2^n that denotes n-tuples over a binary alphabet $\{0,1\}^n$. For simplicity, each valuation v_i is then written in binary notation with n bits. Furthermore, the mapping is defined such that every bit b_i in the binary representation of v_i

assigns variable x_i to the value of b_i. In other words, each valuation v_i is denoted by an indexed n-tuple.

Similarly, the complement of $\mathcal{V}_\phi$ with possible valuations v for which $v(\phi)$ evaluates to 0 can be determined. This set will be called $\mathcal{V}_{\neg\phi}$.

$$\mathcal{V}_{\neg\phi} = \{v \in 2^n \mid v(\phi) = 0\}$$

Finally, the set of all possible valuations is called $\mathcal{V}$.

$$\mathcal{V} = 2^n$$

Example 5.1. *Given the CNF-formula $\phi = (x_0 \vee x_1 \vee x_2) \wedge (\neg x_0 \vee x_1 \vee \neg x_2)$ with $n = 3$, the sets $\mathcal{V}_\phi, \mathcal{V}_{\neg\phi}$ and $\mathcal{V}$ would be the following.*

$$\begin{aligned} \mathcal{V}_\phi &= \{001, 010, 011, 100, 110, 111\} \\ \mathcal{V}_{\neg\phi} &= \{000, 101\} \\ \mathcal{V} &= \{000, 001, 010, 011, 100, 101, 110, 111\} \end{aligned}$$

Each element $v_i \in \mathcal{V}_\phi$ is written in binary notation. The valuation $v_2 = 011$ instantiates x_0 to 0, x_1 to 1 and x_2 to 1 such that $v(\phi) = 1$. Similarly, all the other elements in $\mathcal{V}_\phi$ also cause $v(\phi) = 1$.
000 is part of $\mathcal{V}_{\neg\phi}$ because the first clause evaluates to 0 and thus $v(\phi) = 0$. This holds similarly for 101 with respect to the second clause.

5.1.2 Defining Clauses

A clause c is a disjunction of three variables such that $c = (x_h \vee x_j \vee x_k)$ with $h, j, k \in \{0, \dots, n-1\}$ and $h \neq j$, $h \neq k$, $j \neq k$.
This c is then determined by the set $\mathcal{C}_c$ of valuations v such that $v(c) = 0$.

$$\mathcal{C}_c = \{v \in 2^n \mid v(c) = 0\}$$

During the thesis I will use the notation $\mathcal{C}_i = \mathcal{C}_{c_i}$ for readability.

Example 5.2. *Given a clause $c_i = (\neg x_0 \vee x_1 \vee \neg x_2)$ and $n = 4$, then c_i is represented as a set in the following way.*

$$\mathcal{C}_i = \{1010, 1011\}$$

because $\forall v_j \in \mathcal{C}_i : v_j(c_i) = 0$.

A set that represents a clause cannot efficiently be stored by separately storing all its elements, because the set becomes exponentially large. Representing the set as an abstract formula that defines a clause is also not practical, because if the set is manipulated then the structure of the formula may not hold anymore. On top of that there are also many cases to be distinguished to represent each possible clause. I'll give one example of how a clause may be represented as an abstract formula. In this case I do not use binary notation.

Given the following clause $c_i = (l_{n-2} \vee l_{n-1} \vee l_n)$, then c_i can be represented as an abstract set as follows.

$$\mathcal{C}_i = \{base10(val(l_{n-2})\, val(l_{n-1})\, val(l_n)) + 2^{n-3}k \mid k \in \mathbb{N}, 0 \leq k \leq 2^{n-3}\}$$

The function $base10$ is a mapping $\{0,1\}^* \to \mathbb{N}$, that converts a binary number to its decimal counterpart. The function $val : \mathbb{L} \to \{0,1\}$ maps positive literals to 0 and negative literals to 1.

Example 5.3. *Given an arbitrary CNF-formula with $n = 6$ and the following clause $c_i = (x_3 \vee \neg x_4 \vee x_5)$, then the set of valuations for c_i is:*

$$\begin{aligned} \mathcal{C}_i &= \{base10(010) + 2^{6-3}k \mid 0 \leq k \leq 2^{6-3}\} = \{2 + 2^3k \mid 0 \leq k \leq 2^3\} \\ &= \{2, 10, 18, 26, 34, 42, 50, 58\} \\ &= \{000010, 001010, 010010, 011010, 100010, 101010, 110010, 111010\} \end{aligned}$$

In order to represent clauses in a simpler way the definition of sets is adjusted in the next subsection.

5.1.3 Simplifying sets

In regular sets the elements are unique natural numbers or sequences of bits $b \in \{0,1\}$. This poses several problems:

- If there are a lot of elements in a set which cannot be easily enumerated by some formula, the set becomes exponentially large to store.
- Intersections and unions of several sets can take exponential time and/or space to compute.
- Complex sets are difficult to read when denoted by several formulas or consisting of many elements.

To tackle some of these problems a bit b can now take one out of three values $\in \{0, 1, X\}$. Intuitively, the value X is to be treated as: it can be either 0 or 1. Consider for instance the following equivalent sets.

$$\begin{aligned}\{X\} &\equiv \{0, 1\} \\ \{1X\} &\equiv \{10, 11\} \\ \{XX\} &\equiv \{00, 01, 10, 11\} \\ \{1XX, X1X, XX1\} &\equiv \{001, 010, 011, 100, 101, 110, 111\}\end{aligned}$$

As a consequence, one element of a set in this X-notation can now represent an exponential number of elements in normal bit notation. In other words, some collection of binary numbers can now be stored as sequences $t \in \{0, 1, X\}^*$.
However, using this notation does not reduce the size of all possible sets. For example, $\{000, 111\}$ cannot be simplified using the X-notation.

Printing all the binary numbers contained in one element can be achieved with Algorithm 4.

Algorithm 4 Printing a regular set from an element in X notation

```
procedure PRINT ELEMENTS($S$)                ▷ A binary number in X-notation
    PrintRecursive($S$, $|S|$)                ▷ Enumerate all possible numbers
end procedure

procedure PRINTRECURSIVE($S$, $pos$)
    if $pos == 0$ then                        ▷ Stop condition
        print($S$)
    else          ▷ Recursively fill in the X's, starting at the last one
        $pos \leftarrow pos - 1$
        if getVar($S$, pos) == X then
            setVar($S$, $pos$, 0)              ▷ set $S_{pos}$ to 0
            printRecursive($S$, $pos$)
            setVar($S$, $pos$, 1)              ▷ set $S_{pos}$ to 1
            printRecursive($S$, $pos$)        ▷ Total of $2^{|S|}$ recursive calls
        else                                  ▷ Skip the already assigned bits
            printRecursive($S$, $pos$)        ▷ total of $|S|$ recursive calls
        end if
    end if
end procedure
```

Algorithm 4 also proves the $O(2^{|S|})$ worst case complexity of one such element S, because if S only contains X's, the second else statement is never reached and a total of $2^{|S|}$ recursive calls is executed.

5.1.4 Defining clauses in simpler set notation

Using the X-notation, a clause can now easily be transformed into a set.

Given a 3-SAT formula with n variables and a clause c_i that consists of literals l_h, l_j and l_k with $h, j, k \in \{0, \dots, n-1\}$ and $h \neq j$, $h \neq k$, $j \neq k$ and $h < j < k$ the resulting set is:

$$\mathcal{C}_i = \{\underbrace{X, \dots, X,}_{h\ X\text{'s}} val(l_h),\ \underbrace{X, \dots X,}_{j-h-1\ X\text{'s}}\ val(l_j),\ \underbrace{X, \dots X,}_{k-j-1\ X\text{'s}}\ val(l_k),\ \underbrace{X, \dots X,}_{n-k-1\ X\text{'s}}\ \}$$

Recall that the function $val : \mathbb{L} \to \{0, 1\}$ maps positive literals to 0 and negative literals to 1. Note that the resulting set $\mathcal{C}_i$ now always contains just one element. A concrete example is shown below.

Example 5.4. *A 3-SAT formula with $n = 6$ and the following clause $c_i = (x_1 \vee \neg x_2 \vee x_5)$ is given. If the theory above is applied then $h = 1, j = 2$ and $k = 5$. This means there is $h = 1$ X in front, $2 - 1 - 1 = 0$ X's after x_h, and so on. The set that represents this clause is then defined as follows.*

$$\mathcal{C}_i = \{X01XX0\}$$

5.2 Deciding Satisfiability

An arbitrary set $\mathcal{C}_i$ that belongs to a clause c_i that is part of a 3-SAT formula ϕ represents some of the forbidden valuations of ϕ. The reason for this is that if any clause that is part of ϕ is unsatisfiable, then ϕ is also unsatisfiable. I.e. $v(c_i) = 0 \implies v(\phi) = 0$. Using this information, any 3-SAT formula can be proven (un)satisfiable using sets, as will shown in the next Lemma's.

Lemma 5.1. *A 3-SAT formula ϕ with n variables and m sets of clauses $\mathcal{C}_i$ is satisfiable if and only if $\mathcal{V} \setminus \mathcal{C}_1 \cup ... \cup \mathcal{C}_m \neq \emptyset$*

Proof 5.1.
($\Rightarrow$) We know that $\mathcal{C}_1 \cup ... \cup \mathcal{C}_m$ represents all valuations v such that $v(\phi) = 0$, because all such valuations are captured in at least one of the clauses. Therefore this set must be equal to $\mathcal{V}_{\neg\phi}$. So any set $\mathcal{C}_i$ is a subset of $\mathcal{V}_{\neg\phi}$.
$\mathcal{V} = \mathcal{V}_\phi \cup \mathcal{V}_{\neg\phi}$ contains all possible valuations, because $\mathcal{V}_{\neg\phi}$ is defined as the complement of $\mathcal{V}_\phi$.

So if ϕ is satisfiable then $\mathcal{V}_\phi \neq \emptyset$. So there must be at least one valuation $v \in \mathcal{V}$ that is not in $\mathcal{V}_{\neg\phi}$. Therefore $\mathcal{V} \setminus \mathcal{C}_1 \cup ... \cup \mathcal{C}_m \neq \emptyset$

($\Leftarrow$) If $\mathcal{V} \setminus \mathcal{C}_1 \cup ... \cup \mathcal{C}_m \neq \emptyset$ then there is a valuation $v \notin \mathcal{V}_{\neg\phi}$ that therefore can only be in $\mathcal{V}_\phi$ and thus satisfies ϕ. $\square$

Lemma 5.2. *A 3-SAT formula ϕ with n variables and m clauses is satisfiable if and only if $|\mathcal{V}| \neq |\{\mathcal{C}_1 \cup ... \cup \mathcal{C}_m\}|$*

Proof 5.2.
($\Rightarrow$) $\mathcal{C}_1 \cup ... \cup \mathcal{C}_m$ contains at most 2^n elements. If ϕ is satisfiable then there must be an assignment that is not in $\mathcal{C}_1 \cup ... \cup \mathcal{C}_m$. Therefore $|\mathcal{C}_1 \cup ... \cup \mathcal{C}_m| < |\mathcal{V}|$, thus $|\mathcal{V}| \neq |\mathcal{C}_1 \cup ... \cup \mathcal{C}_m|$.

($\Leftarrow$) If $|\mathcal{V}| \neq |\mathcal{C}_1 \cup ... \cup \mathcal{C}_m|$ then $\mathcal{V}_\phi \neq \emptyset$, thus $|\mathcal{V}_\phi| > 0$. That means there is always an element in $\mathcal{V}_\phi$ that is not in $\mathcal{C}_1 \cup ... \cup \mathcal{C}_m$ and satisfies ϕ. $\square$

While the 3-SAT problem is now translated in set theory, it still remains exponentially hard to calculate the union of several clauses as will be shown in the next sections.

5.2.1 An algorithm to count shared elements of clauses

To answer the question whether a given 3-SAT formula ϕ is satisfiable according to Lemma 5.1 or Lemma 5.2, a procedure must be defined to either calculate the union of several sets, or to calculate just the number of valuations that are contained in that set. Both methods are non trivial and take an exponential amount of time or space to compute. In this thesis I will solely present an algorithm to count valuations in a set of clauses. I've chosen not to generate big solution sets, since that might be too similar to BDDs.

Recall that a clause $\mathcal{C}_i$ is a set that consists of only one element, but, because we are using the X-notation contains several valuations. So in a sense, one element in X-notation is a set in itself on which we can also apply the Union and Intersection operations with respect to another element. For example, if we want to determine the shared valuations of two elements, we can calculate the intersection. This is done by bitwise comparison of each bit b_i in both elements. If one bit equals 0 and the other equals 1, then the intersection of the two elements is empty (1). If both bits are equal then the resulting bit in the intersection has the same value too (2,3). If one bit equals X then the resulting bit in the intersection equals the value of the other bit (2,3).

For example:

$$X00 \cap XX1 = \emptyset \tag{1}$$
$$X00 \cap 10X = 100 \tag{2}$$
$$XX0 \cap X0X = X00 \tag{3}$$

Determining the amount of valuations in a single element is done as follows. Let k be #X in an element x in X-notation. Then, the total number of valuations that x can contain equals 2^k. For example:

$$|X00| = 2^1 = 2$$
$$|000| = 2^0 = 1$$
$$|XXX| = 2^3 = 8$$

In Algorithm 5 this is what the function numberOfValuations(x) calculates, where x is an element in X-notation. Algorithm 5 in its entirety calculates the number of valuations in a set $\mathcal{S}$. It is invoked with the set of clauses $\mathcal{V}_{\neg\phi} = \mathcal{C}_1 \cup ... \cup \mathcal{C}_m$ belonging to a CNF-formula ϕ with n variables and m clauses. Satisfiability of ϕ can then be easily determined by checking if getNumUnion($\mathcal{V}_{\neg\phi}$) does not equal 2^n. If this is the case then ϕ is satisfiable, because this means that $|\mathcal{V}_{\neg\phi}| < 2^n$, which means that not all valuations evaluate ϕ to 0. So if all valuations evaluate ϕ to 0, i.e. $|\mathcal{V}_{\neg\phi}| = 2^n$, then ϕ is unsatisfiable.

5.2.2 Complexity

So Algorithm 5 does not solve satisfiability on itself, it is however the answer to the #3-SAT problem, which is also NP-complete and asks the question how many satisfying assignments there are for a given 3-SAT instance. This problem is actually harder than the 3-SAT problem and the best known algorithm has an $O(1.6423^n)$ worst case complexity expressed in the number of variables or $O(1.4142^m)$ expressed in the number of clauses.[22]

However, calling getNumUnion() takes an even higher exponential amount of time. The reason for this is that during every iteration of the main for-loop, which runs $|\mathcal{S}|$ times, the same function is called again recursively with another set $\mathcal{I}$, which in its turn also has to run $|\mathcal{I}|$ times. In the worst case, rules 6-8 could add i elements to $\mathcal{I}$, where i is the number of the current iteration (starting at 0). Each of these sets $\mathcal{I}_i$ that are generated then has a maximum size of $0, \ldots, |\mathcal{S} - 1|$ respectively. Recursively in their turn, they can generate even more sets $\mathcal{I}'_i$ and this can happen several times depending on the size of the original set $\mathcal{S}$. To discover the worst case complexity of the function, consider the following example.

Algorithm 5 Calculating the number of valuations in a set

```
1: function GETNUMUNION($S$)                          ▷ A set of elements in X-notation
2:     $count \leftarrow 0$
3:     for all $e_i \in \mathcal{S}$ do   ▷ Add the number of valuations in each element
4:         $count \leftarrow count +$ numberOfValuations($e_i$)
5:         $\mathcal{I} \leftarrow \emptyset$
6:         for all $e_j \mid j < i$ do               ▷ Calculate the shared valuations
7:             $\mathcal{I} \leftarrow \mathcal{I} \cup (e_i \cap e_j)$
8:         end for
9:         $count \leftarrow count -$ getNumUnion($\mathcal{I}$)   ▷ Substract the number of shared valuations (very inefficient)
10:    end for
11:    return $count$
12: end function
```

Suppose that getNumUnion() is called with 4 elements in $\mathcal{S}$. The following table then illustrates some worst case statistics.

i	$\lvert\mathcal{I}_i\rvert$	Extra iterations	Total iterations performed
0	0	0	$1 = 1 + 0$
1	1	1	$2 = 1 + 1$
2	2	$3 = 1 + 2$	$4 = 1 + 3$
3	3	$7 = 1 + 2 + 4$	$8 = 1 + 7$

A value in the Extra iterations column is equal to the sum of all previous Total iterations performed values. Let's determine what happens for each iteration in the for-loop. For simplicity it is assumed that a call to getNumUnion() with an empty set takes 0 iterations.
For iteration 0 e_i has 0 preceding elements, so rules 6-8 can generate no new set, so there is only 1 iteration performed.
For iteration 1 e_i has 1 preceding elements, so rules 6-8 can generate a new set with 1 element. This newly generated set undergoes the same procedure up to iteration 0. So a total of $1 + 1 = 2$ iterations is performed .
For iteration 2 e_i has 2 preceding elements, so rules 6-8 can generate a set with 2 elements. The first element undergoes the same procedure up to iteration 0 and the second element up to iteration 1. So a total of $1 + (1 + 2) = 4$ iterations are performed.
And finally iteration 3 is performed analogue to the previous ones.

This pattern suggests that the added complexity of this function is $O(2^{|\mathcal{S}|})$, which is equal to 2^m if m is the number of clauses. So this solution is not computationally feasible with a high amount of clauses and in fact dramatically more time consuming than brute-forcing all 2^n possible valuations.

5.2.3 Properties of 3-SAT formulas

There are some general properties that can be deduced from the structure of any 3-SAT formula. These properties will be described and proven in this section.

Lemma 5.3. *Any arbitrary set $\mathcal{C}_i$ that represents a clause always contains exactly 2^{n-3} valuations if $n \geq 3$.*

Proof 5.3. $\mathcal{C}_i$ is always based on three different literals by definition. The resulting bits that are mapped to these literals in the only element in $\mathcal{C}_i$ are always instantiated to either 0 or 1 by definition. The other $n-3$ bits can take any value and thus can be instantiated in 2^{n-3} possible ways. Therefore, the resulting set contains one element that represents exactly 2^{n-3} valuations. □

A direct consequence of Lemma 5.3 is Lemma 5.4.

Lemma 5.4. *There are at least 8 clauses needed to make any 3-SAT formula with at least 3 variables unsatisfiable.*

Proof 5.4. Suppose 8 non-overlapping sets are given of which their intersection is empty. Then the number of valuations in their union is $8 * 2^{n-3}$, because we know according to Lemma 5.3 that each set contains 2^{n-3} valuations. Therefore the total amount of valuations is $2^3 * 2^{n-3} = 2^n$, which is equal to the set $\mathcal{V}$, which contains all possible valuations. Therefore all valuations are forbidden and the formula is unsatisfiable. This cannot be done with fewer than 8 sets, because a set always contains exactly 2^{n-3} valuations, so we would need at least 8. □

5.3 BDDs versus adapted set-theory

BDDs are an efficient way to represent the entire set of valuations, yet the memory requirements are steep, because some formula's may still generate exponential sized diagrams. However, the memory requirements for the adapted set theory solver are minimal, because it only calculates the number of elements that are *supposed* to be in the conflicting set; it is not actually generated. But this strategy does not come without a price, as the running time is highly exponential.

Closer analysis of these observations brings some further questions to mind: What *if* these sets were actually generated? Will the size of these sets be similar to the size of a BDD? A BDD captures both the satisfying as the non-satisfying sets in a single diagram, but for the above solver to work, only one of the two sets has to be constructed. Does this mean that generating a solution set in X-notation is more efficient? Or is the adapted set theory just another representation of a BDD in the end?

These are questions that this thesis does not yet answer, but might form the basis for future research and novel insights in SAT-solving algorithms to come.

6 Conclusion

There is no doubt that many different ways exist to solve 3-SAT. From searching the space of valuations, to constructing the entire solution set to proving (un)satisfiability by manipulating the formula; each different type of solver has its own strengths and weaknesses.

The DP algorithm and BDDs for example have a worst case exponential memory requirement. In case where the solution set is very small or very big, BDDs are good to use as the diagrams are structured in such a way that these sets can be represented efficiently. The DP algorithm is usually more inefficient, because every resolution step that it takes creates a new lengthy clause and in many cases, there are exponentially many resolution actions to take. This is also the reason that this algorithm is not used much in practice anymore.

Recall that to improve memory requirements DLL was invented. The memory usage of this algorithm is highly controllable and thus better than that of BDDs and especially DP. But in terms of run-time performance, BDDs can be very efficient where DLL is not. For example in those cases of critically constrained problems - where DLL struggles to find a solution because there are many partial solutions - the size of a BDD remains small while it is constructed, resulting in a solution that is quickly found. But also the opposite scenario can occur. If the solution sets are more equal in size, or if the solution set of the formula is structured oddly, running DLL might have a better chance at deciding the instance fast than constructing a BDD.

Last but not least, it would not be wise to use the theory discussed in the last chapter in practice, since it is exponentially dependent on the number of clauses. It might have better memory requirements than most algorithms since the solution sets are not explicitly generated, but the algorithm run-time becomes intractable very quickly, since almost all formulas have more clauses than variables. As a result, brute-forcing all possible variable combinations would be even faster in many cases.

In the end many real world problem instances can be solved efficiently enough by at least one of these algorithms, but there always remain some hard random 3-SAT instances for which every algorithm struggles. Especially in region around the class of critically constrained problems do solvers and provers have trouble to stay out of the exponential zone. However the class of exponential problems continues to become smaller as the algorithms are optimized and new ideas are invented. Is this really an essential worst case exponential problem? Only time will tell.

References

[1] Lintao Zhang, *Searching For Truth: Techniques For Satisfiability of Boolean Formulas*, PhD Thesis, Princeton University, 2003.

[2] Yogesh S. Mahajan, Zhaochui Fu and Sharad Malik, *Zchaff2004: An Efficient SAT Solver*, Lecture Notes in Computer Science, 2005, Volume 3542/2005, 898, DOI: 10.1007/11527695_27.

[3] J. M. Howe and A. King, *A Pearl on SAT Solving in Prolog*, In Functional and Logic Programming, volume 6009 of Lecture Notes in Computer Science, pages 165-174, Springer, 2010

[4] Quirin Meyer, Fabian Schnfeld, Marc Stamminger, Rolf Wanka, *3-SAT on CUDA: Towards a Massively Parallel SAT Solver*, Proc. High Performance Computing and Simulation Conference (HPSC) ; 2010. pp. 306-313. [doi:10.1109/HPCS.2010.5547116]

[5] `http://www.claymath.org/millennium/` (2012 Clay Mathematics Institute)

[6] Norbert Manthey, *Improving SAT Solvers Using State-of-the-Art Techniques*, Thesis, Technische Universitt Dresden (2010)

[7] G. Logemann, M. Davis and D. Loveland. *A machine program for theorem proving* Communications of the ACM, pages 394-397, 1962

[8] Philippe Chatalic and Laurent Simon, *Davis and Putnam 40 years later: a first experitmentation*, 2000

[9] M. Davis and H. Putnam, *A computing procedure for quantification theory*, J. ACM 1, pages 201-215, 1960

[10] Zvi Galil, *On the complexity of regular resolution and the Davis-Putnam procedure*, Theoretical Computer Science, Volume 4, Issue 1, Pages 23-46, February 1977. [doi: 10.1016/0304-3975(77)90054-8]

[11] Lawrence Ryan, *Efficient algorithms for clause-learning SAT solvers*, M. Sc. Thesis, Simon Fraser University, 2004

[12] Robert NieuwenHuis, Albert Oliveras, Technical University of Catalonia, Barcelona, Spain, Cesare Tinelli, The University of Iowa, Iowa City, Iowa, *Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T)*, Journal of the ACM, Volume 53 Issue 6, November 2006, [doi:10.1145/1217856.1217859]

[13] Paolo Liberatore, *On the complexity of choosing the branching literal in DPLL*, Dipartimento di Informatica e Sistemistica, Università di Roma “La Sapienza”, Via Salaria 113, I-00198, Roma, Italy, `http://dx.doi.org/10.1016/S0004-3702(99)00097-1`

[14] Carsten Sinz, *Visualizing SAT Instances and Runs of the DPLL Algorithm*, Journal of Automated Reasoning, Volume 39, Number 2, 219-243, DOI: 10.1007/s10817-007-9074-1

[15] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik, *Chaff: engineering an efficient SAT solver.*, Proceedings of the 38th annual Design Automation Conference (DAC ’01). ACM, New York, NY, USA, 530-535. DOI: 10.1145/378239.379017

[16] Freeman, J.W. *Improvements To Propositional Satisfiability Search Algorithms.* Ph.D. Dissertation, Department of Computer and Information Science, University of Pennsylvania, May 1995.

[17] D. Pretolani. *Efficiency and stability of hypergraph SAT algorithms.* In D. S. Johnson and M. Trick, editors, Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge., DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1995.

[18] João Marques-Silva, *The Impact of Branching Heuristics in Propositional Satisfiability Algorithms*, Lecture Notes in Computer Science, Volume 1695/1999, 850, 1999. DOI: 10.1007/3-540-48159-1_5

[19] Yehuda Naveh, *The Big Deal: Applying Constraint Satisfaction Technologies Where It Make the Difference* In Theory and Applications of Satisfiability Testing - SAT 2010, Ofer Strichman

[20] James M. Crawford, Larry D. Auton, *Experimental results on the crossover point in random 3-SAT*, Artificial Intelligence, Volume 81, Issues 12, March 1996, Pages 31-57, ISSN 0004-3702, 10.1016/0004-3702(95)00046-1. (`http://www.sciencedirect.com/science/article/pii/0004370295000461`)

[21] Cristian Coarfa, Demetrios D. Demopoulos, Alfonso San Miguel Aguirre, Devika Subramanian and Moshe Y. Vardi. *Random 3-SAT: The Plot Thickens* Principles and Practice of Constraint Programming CP 2000, Pages 143-159, 2000. Doi: 10.1007/3-540-45349-0_12

[22] Junping Zhou, Minghao Yin, Chunguang Zhou, *New Worst-Case Upper Bound for #2-SAT and #3-SAT with the Number of Clauses as the Parameter*, Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI-10), 2010

[23] A. Mishchenko, *An Introduction to Zero-Suppressed Binary Decision Diagrams.* `http://www.ee.pdx.edu/~alanmi/research/`, 2001.

[24] DoRon B. Motter and Igor L. Markov, *A Compressed Breadth-First Search for Satisfiability*, Lecture Notes in Computer Science, 2002, Volume 2409/2002, 55, DOI: 10.1007/3-540-45643-0_3

[25] Alfonso San Miguel Aguirre and MosheY. Vardi, *Random 3-SAT and BDDs: The Plot Thickens Further*, Lecture Notes in Computer Science, 2001, Volume 2239/2001, 121-136, DOI: 10.1007/3-540-45578-7_9