Boolean satisfiability problem

Tomasz Marzec

Jagiellonian University, Kraków, Poland
Email address: tomasz4.marzec@student.uj.edu.pl

Abstract. This paper aims to provide an overview of the Boolean satisfiability problem by analyzing various approaches to solving it. That analysis is supplemented by examples designed to help the reader understand the most important concepts in-depth. We will see the Davis–Putnam–Logemann–Loveland (DPLL) algorithm, on top of which we will explore two well-established classes of algorithms for solving the SAT problem — the conflict-driven clause learning and the look-ahead based solvers. We will also analyze the probabilistic algorithm for solving the 3-SAT problem. Then we will explore the MAX-SAT problem, which is an extension of the SAT problem. We will see some approaches to solving this problem: the branch and bound algorithm and two approximation algorithms.

Contents

CHAPTER 1

Introduction

The Boolean satisfiability (SAT) problem stands at the origins of Computer Science. In the study of computational complexity theory, it was the first problem proven to be NP-complete [**5**].

It appears in a broad range of applications, ranging from automatic theorem proving, to planning [**10**] in Artificial Intelligence, and job scheduling, and even has various uses in biology [**13, 4**]. Due to its importance, there is a lot of motivation to create better heuristics and new algorithms, that are vital for the progress of solving the SAT problem. Thanks to that, many problems involving the SAT problem, that seemed out of reach years ago, now are handled effortlessly. There are yearly competitions surrounding the progress of procedures for solving SAT instances, where researchers compete in solving SAT instances with state-of-the-art SAT solvers. This gives them a great opportunity to compare their results on SAT instances and present their research to a broader audience.

In this paper we will see the basic algorithm for solving the SAT problem, which is a version of the Davis–Putnam–Logemann–Loveland (DPLL) algorithm. On top of that, we will explore two various approaches to improve its solving capabilities. Firstly, we will see a conflict-driven learning algorithm, that uses encountered conflicts, that happen when exploring different assignments, to learn new clauses, which helps it learn more about the formula. By analysis of these conflicts, it is able to use non-chronological backtracking when recovering from a conflict. Then we will see another approach that focuses on doing extensive decision-making heuristics by exploring and ranking possible assignment choices at each step. We will then see a probabilistic approach to the 3-SAT problem, which achieves better asymptotic time complexity but comes with a price of correctness — there is a small probability that it might return an incorrect result.

Afterwards we will explore the MAX-SAT problem, which is an extension of the SAT problem. We will see an example implementation of a branch and bound (BnB) algorithm to this problem, which is a well-known approach to solving difficult optimization problems. With that, we will see a number of possible heuristics that improve its performance. Then we will explore two approximation algorithms for solving this problem, which provides an interesting perspective on the lower bound of the result of the MAX-SAT problem.

CHAPTER 2

SAT

2.1. Definition

The language of Boolean formulae consists of Boolean variables (usually denoted $x, y, \ldots$), which can be assigned **true** or **false** Boolean values, and Boolean operators - conjunction($\wedge$), disjunction ($\vee$), negation ($\neg$) and parentheses.
It is defined inductively as follows:

Definition 2.1. Variables are formulae. If A is a formula, then $\neg A$ is a formula. If A and B are formulae, then $(A \wedge B)$ and $(A \vee B)$ are formulae.

With grammar defined we go on to formally define the assignment of values to variables:

Definition 2.2. Truth assignment v for formula A is a function of type $Var(A) \rightarrow \{\mathbf{false}, \mathbf{true}\}$, where $Var(A)$ is a set of variables in A. It can be extended to assign **true** and **false** values to formulae in the following manner:

- $v^*(x) = v(x)$.
- $v^*(A \wedge B) = v^*(A) \wedge v^*(B)$.
- $v^*(A \vee B) = v^*(A) \vee v^*(B)$.
- $v^*(\neg A) = \neg v^*(A)$.

where v^* denotes the extended truth assignment.

Definition 2.3. A formula A is said to be *satisfiable* when there exists a truth assignment v such that $v^*(A) = \mathbf{true}$.

Definition 2.4. The Boolean satisfiability problem is the problem of deciding whether a given formula is *satisfiable*.

There is a number of different definitions of SAT, which differ on how restricted syntax the formula can have. One of the more common definitions requires the formula to be in Conjunctive normal form (CNF-SAT):

Definition 2.5. A literal is either a Boolean variable or its negation. A clause is a disjunction ($\vee$) of literals. A formula is said to be in CNF form when it is a clause or conjunction of clauses.

It is known that these two problems are equivalent. Using Tseitin encoding[**17**] any formula can be transformed into a CNF formula that is equisatisfiable, and its size is linear in terms of the size of the original formula.

From now on, formulae are assumed to be in CNF. Formula Φ in CNF can be treated as a set of clauses, where each clause is set of literals. For example $(x_1 \vee \neg x_2) \wedge (\neg x_3)$ corresponds to $\{\{x_1, \neg x_2\}, \{\neg x_3\}\}$.

2.2. DPLL algorithm

2.2.1. Introduction.

The Davis-Putnam-Logemann-Loveland (DPLL) algorithm corresponds to a backtracking-based search algorithm for solving the CNF-SAT problem. At each step of the algorithm, it chooses a literal and assigns a Boolean value to it. Then the formula is simplified using the logical consequences of the assignment. If at some point an unsatisfiable clause is identified, backtracking is executed. Assignment choices are undone until a branch with only one Boolean value checked is encountered. If the second choice is undone, then the formula is declared to be *unsatisfiable*.

2.2.2. The algorithm.

Algorithm 1 DPLL algorithm

```
procedure DPLL($\Phi$)
    if $\Phi = \emptyset$ then
        return SAT                                   ▷ Satisfied formula
    end if
    if $\emptyset \in \Phi$ then
        return UNSAT                                 ▷ Unsatisfiable clause
    end if
    $x \leftarrow$ choose variable in $\Phi$
    if DPLL($\Phi[x = \mathbf{true}]$) = SAT then
        return SAT
    end if
    return DPLL($\Phi[x = \mathbf{false}]$)
end procedure
```

Where $\Phi[l = \mathbf{B}]$ is Φ conditioned on literal l being assigned Boolean value $\mathbf{B}$. For example, for $\Phi = \{\{x, y\}, \{\neg x\}, \{\neg x, z\}\}$ it follows that $\Phi[x = \mathbf{true}] = \{\{\}, \{z\}\}$. $\Phi[l = \mathbf{true}]$ can be defined as a formula that results from the application of these two rules on Φ:

- From every clause: if it contains $\neg l$ as a literal, then this literal should be removed from the clause.
- Every clause containing literal l is deleted.

Also $\Phi[l = \mathbf{false}]$ is equal to $\Phi[\neg l = \mathbf{true}]$.

2.3. CDCL Solvers

2.3.1. Introduction.

Conflict-driven clause learning SAT solvers are inspired by DPLL solvers. On top of the logic used by DPLL solvers, they involve some additional techniques, like learning from conflicts and exploiting the properties of their structure. Conflicts occur when some clause becomes unsatisfied. Using that additional knowledge allows the creation of very efficient algorithms, which has contributed to their widespread use in a number of practical applications. The main practical difference between DPLL and CDCL SAT solvers is that CDCL solvers use clause learning and non-chronological backtracking. Clause learning is a process of transforming a formula to an equivalent one in terms of the SAT problem using learned (inferred) clauses. Non-chronological backtracking allows backtracking to any decision level, rather than one immediately preceding the current one.

Another transformation of formulae used in CDCL is unit propagation. It is based on unit clauses (composed of a single literal). The existence of a unit clause in formulae implies that this literal has to be true. Result of a single unit propagation on clause (l) in formula Φ is equal to $\Phi[l = \textbf{true}]$.

2.3.2. The algorithm.

Algorithm 2 Typical CDCL algorithm [**16**]

```
procedure CDCL($\Phi, v$)
    if UnitPropagation($\Phi, v$) = CONFLICT then
        return UNSAT
    end if
    $dl \leftarrow 0$                                            ▷ Decision level
    while not AllVariablesAssigned($\Phi, v$) do
        $(x, y) \leftarrow$PickBranchingVariable($\Phi, v$)         ▷ Decide stage
        $dl \leftarrow dl + 1$            ▷ Increment decision level due to a new decision
        $v \leftarrow v \cup \{(x, y)\}$
        if UnitPropagation($\Phi, v$) = CONFLICT then          ▷ Deduce stage
            $\beta \leftarrow$ConflictAnalysis($\Phi, v$)               ▷ Diagnose stage
            if $\beta < 0$ then
                return UNSAT
            else
                Backtrack($\Phi, v, \beta$)
                $dl \leftarrow \beta$        ▷ Decrement decision level due to backtracking
            end if
        end if
    end while
    return SAT
end procedure
```

The procedures used within the algorithm work as follows:

- UnitPropagation - executes unit propagation as long as there is a unit clause. On top of that, it extends assignment v with the values implied from the unit clauses. It also detects conflicts (empty clauses).

- AllVariablesAssigned - returns true if and only if all variables from Φ have assigned **false** or **true** in the current assignment.
- PickBranchingVariable - chooses a variable and the logical value to be assigned.
- ConflictAnalysis - makes an analysis of the conflict and executes clause learning. As a result, at least one new clause is learned, and the resulting decision level is computed.
- Backtracking - backtracks to the decision level chosen by ConflictAnalysis.

2.3.3. An example.

One useful way to showcase the functionality of this algorithm is with *implication graphs* [**15**]. Its vertices are variables with assigned Boolean values and its directed edges represent direct dependencies — we say that literal l_1 *directly depends* on literal l_2 when l_1 was assigned with UnitPropagation executed on a clause which initially included l_2 or $\neg l_2$.

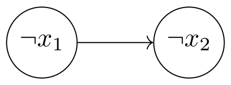
A good way to track the choices made by the algorithm is by keeping the trail of all the literals which were assigned true. The trail is a list of the assignments made by the algorithm. I will present it using a table (with design motivated by [**11**]) that keep track of decision levels, literals and their reasons. The reason column contains a list of literals, on which the literal *directly depends.* Each row in the table corresponds to either a choice made by the algorithm, or an assignment implied by unit propagation.

Note that edges of the *implication graph* can be derived from the table. There are edges going into each literal from all the literals (or their negations) in their reason. Loops are omitted for clarity.

Consider a formula:

$$(x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_5 \vee x_4) \wedge (\neg x_3 \vee \neg x_6 \vee \neg x_8) \wedge (\neg x_6 \vee x_8)$$
$$\wedge (x_1 \vee x_5 \vee x_7) \wedge (\neg x_9 \vee x_5 \vee x_{10}) \wedge (x_4 \vee \neg x_{10} \vee \neg x_9)$$

Because there are no unit clauses, assume that the algorithm sets x_1 to **false**. The clause $(x_1 \vee \neg x_2)$ implies that x_2 must be set to **false**.



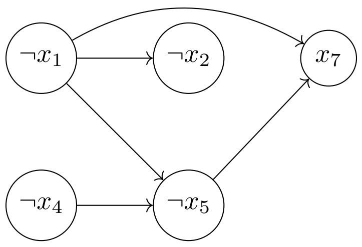
decision level	literal	reason
0	$\neg x_1$	-
0	$\neg x_2$	$\{x_1, \neg x_2\}$

Figure 1 & Table 1. The implication graph with the trail

After that the formula is as follows:

$$(x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_5 \vee x_4) \wedge (\neg x_3 \vee \neg x_6 \vee \neg x_8) \wedge (\neg x_6 \vee x_8)$$
$$\wedge (x_1 \vee x_5 \vee x_7) \wedge (\neg x_9 \vee x_5 \vee x_{10}) \wedge (x_4 \vee \neg x_{10} \vee \neg x_9)$$

The algorithm has to choose again. Assume it sets x_4 to **false**. That assignment has caused the second clause to be a unit clause, so it follows that $\neg x_5$ has to be assigned **true**. That in turn causes the fifth clause to become a unit clause, so it follows that x_7 is set to **true**.



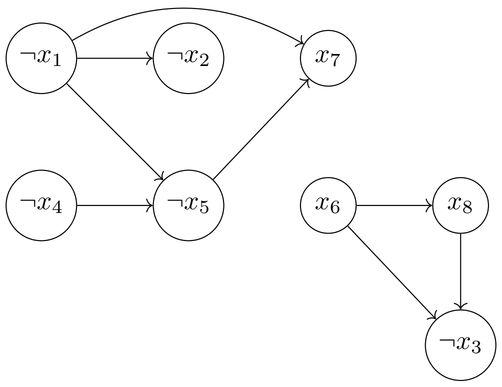
decision level	literal	reason
0	$\neg x_1$	-
0	$\neg x_2$	$\{x_1, \neg x_2\}$
1	$\neg x_4$	-
1	$\neg x_5$	$\{x_1, \neg x_5, x_4\}$
1	x_7	$\{x_1, x_5, x_7\}$

FIGURE 2 & TABLE 2. The implication graph with the trail

After that, the formula becomes:

$$(x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_5 \vee x_4) \wedge (\neg x_3 \vee \neg x_6 \vee \neg x_8) \wedge (\neg x_6 \vee x_8)$$
$$\wedge (x_1 \vee x_5 \vee x_7) \wedge (\neg x_9 \vee x_5 \vee x_{10}) \wedge (x_4 \vee \neg x_{10} \vee \neg x_9)$$

Assume that the algorithm sets x_6 to **true**. The unit propagation immediately implies that x_8 has to be set to **true**, which in turn forces x_3 to be **false**.



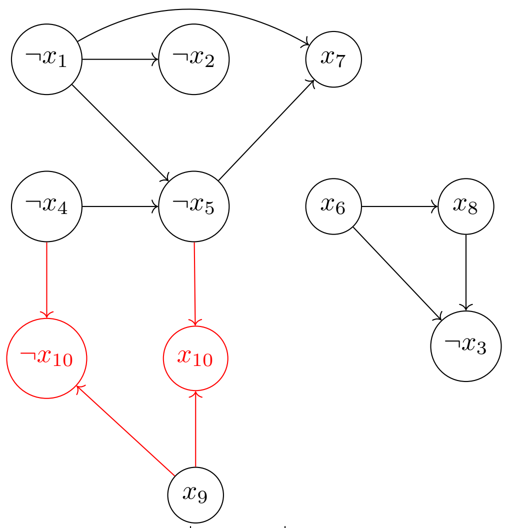
decision level	literal	reason
0	$\neg x_1$	-
0	$\neg x_2$	$\{x_1, \neg x_2\}$
1	$\neg x_4$	-
1	$\neg x_5$	$\{x_1, \neg x_5, x_4\}$
1	x_7	$\{x_1, x_5, x_7\}$
2	x_6	-
2	x_8	$\{\neg x_6, x_8\}$
2	$\neg x_3$	$\{\neg x_3, \neg x_6, \neg x_8\}$

FIGURE 3 & TABLE 3. The implication graph with the trail

Then the formula is:

$$(x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_5 \vee x_4) \wedge (\neg x_3 \vee \neg x_6 \vee \neg x_8) \wedge (\neg x_6 \vee x_8)$$
$$\wedge (x_1 \vee x_5 \vee x_7) \wedge (\neg x_9 \vee x_5 \vee x_{10}) \wedge (x_4 \vee \neg x_{10} \vee \neg x_9)$$

Now assume that the algorithm sets x_9 to **true**. This causes a conflict. In order for the remaining formula to be satisfied, x_{10} has to be both **true** and **false**.



decision level	literal	reason
0	$\neg x_1$	-
0	$\neg x_2$	$\{x_1, \neg x_2\}$
1	$\neg x_4$	-
1	$\neg x_5$	$\{x_1, \neg x_5, x_4\}$
1	x_7	$\{x_1, x_5, x_7\}$
2	x_6	-
2	x_8	$\{\neg x_6, x_8\}$
2	$\neg x_3$	$\{\neg x_3, \neg x_6, \neg x_8\}$
3	x_9	-
3	x_{10}	$\{\neg x_9, x_5, x_{10}\}$
3	$\neg x_{10}$	$\{x_4, \neg x_{10}, \neg x_9\}$

FIGURE 4 & TABLE 4. The implication graph with the trail

In the figure 4 red nodes (x_{10} and $\neg x_{10}$) correspond to conflicted literals. Now consider all literals with edges going into literals x_{10} and $\neg x_{10}$ (marked with red colour). Construction of the implication graph tells us that setting literals $x_9, \neg x_4, \neg x_5$ to **true** causes a conflict — x_{10} and $\neg x_{10}$ have to both be true. It is straightforward to verify that this in fact is the case:

$$\ldots \wedge (\neg x_9 \vee x_5 \vee x_{10}) \wedge (x_4 \vee \neg x_{10} \vee \neg x_9)$$

We then know that if $x_9, \neg x_4, \neg x_5$ are all assigned **true**, then the formula can not be satisfied. So, in order for the formula to be satisfied, the clause $(\neg x_9 \vee x_4 \vee x_5)$ has to be satisfied too. We call this clause a learned clause, and it can be appended to the initial formula.
Now non-chronological backtracking can be executed. We know that we have to change a value of one of the tree literals, so in our case backtracking chooses decision level 1 — the decision level of $\neg x_5$, which is the one with the second-highest decision

level of all literals directly affected by the new clause (x_9,$\neg x_4$ and $\neg x_5$).
After backtracking our formula is:

$$(x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_5 \vee x_4) \wedge (\neg x_3 \vee \neg x_6 \vee \neg x_8) \wedge (\neg x_6 \vee x_8)$$
$$\wedge (x_1 \vee x_5 \vee x_7) \wedge (\neg x_9 \vee x_5 \vee x_{10}) \wedge (x_4 \vee \neg x_{10} \vee \neg x_9) \wedge (\neg x_9 \vee x_4 \vee x_5)$$

Because we did not change values of literals $\neg x_4, \neg x_5$, the existence of a newly learned clause immediately implies that x_9 must be **false**.
The trail after backtracking becomes:

decision level	literal	reason
0	$\neg x_1$	-
0	$\neg x_2$	$\{x_1, \neg x_2\}$
1	$\neg x_4$	-
1	$\neg x_5$	$\{x_1, \neg x_5, x_4\}$
1	x_7	$\{x_1, x_5, x_7\}$
1	$\neg x_9$	$\{\neg x_9, x_4, x_5\}$

2.4. Look-Ahead Solvers

2.4.1. Introduction.

The Look-Ahead based SAT Solvers are also based on the DPLL algorithm. They extend the decision stage by exploring the immediate consequences of following various recursion branches. Different choices (of variables and Boolean values) are measured using various possible statistics, which try to predict which choice is the most optimal. It is a different strategy than the one used in CDCL Solvers, which focused on learning from detected conflicts and did not focus on exploring different choices before committing to one recursively. A good analogy that showcases that difference is the problem of finding a target in a maze. The Conflict-driven approach does not spend a lot of time considering different possible directions but keeps very detailed notes on which turns it followed. Then, based on complex analysis, it decides where to return to when hitting a dead end. On the other hand, the Look-Ahead strategy spends more time analyzing various directions it might take when standing at a crossroad.

2.4.2. The algorithm.

Algorithm 3 Look-Ahead based DPLL algorithm [**7**]

```
procedure DPLL($\Phi$)
    if $\Phi = \emptyset$ then
        return SAT
    end if
    $< \Phi; x_{\text{decision}} > \leftarrow$ LookAhead($\Phi$)
    if $\emptyset \in \Phi$ then
        return UNSAT
    else if no $x_{\text{decision}}$ is selected then
        return DPLL($\Phi$)
    end if
    $\mathbf{B} \leftarrow$ DirectionHeuristic($x_{\text{decision}}, \Phi$)
    if DPLL($\Phi[x_{\text{decision}} = \mathbf{B}]$) = SAT then
        return SAT
    end if
    return DPLL($\Phi[x_{\text{decision}} = \neg\mathbf{B}]$)
end procedure
```

The LookAhead procedure is responsible for executing look-aheads on variables. A single look-ahead on variable x works by assigning to it truth values, which is followed by unit propagation. By doing that, the importance of x can be calculated. The LookAhead procedure consists of two heuristics:

- Firstly, it establishes the importance of a variable. It does that by measuring the reduction of the formula after the assignment of a Boolean value. The reduction of the assignment can be calculated using different statistics. The importance of a variable is then calculated as a combination of reductions for assignments of the Boolean values.

- Secondly, it detects conflicts while executing look-aheads. If the assignment of **true** to x has caused a conflict, then x must be assigned **false**. If both **true** and **false** assignments have caused a conflict, then the LookAhead procedure ends. These conflicts are used to simplify the formula or to conclude that the formula is *unsatisfiable*.

Algorithm 4 Look-Ahead algorithm[**7**]

```
procedure LookAhead($\Phi$)
    $P \leftarrow$ Preselect($\Phi$)
    repeat
        for all variables $x \in P$ do
            $\Phi \leftarrow$ LookAheadReasoning($\Phi, x$)
            if $\emptyset \in \Phi[x = \textbf{false}]$ and $\emptyset \in \Phi[x = \textbf{true}]$ then
                return $< \Phi[x = \textbf{false}]; * >$
            else if $\emptyset \in \Phi[x = \textbf{false}]$ then
                $\Phi \leftarrow \Phi[x = \textbf{true}]$
            else if $\emptyset \in \Phi[x = \textbf{true}]$ then
                $\Phi \leftarrow \Phi[x = \textbf{false}]$
            else
                $H(x) \leftarrow$ DecisionHeuristic($\Phi, \Phi[x = \textbf{false}], \Phi[x = \textbf{true}]$)
            end if
        end for
    until nothing (important) has been learned
    return $< \Phi; x$ with greatest $H(x) >$
end procedure
```

The procedures used within the algorithm work as follows:

- Preselect - returns a subset of variables from the formula. Instead of considering all variables as potential decision variables, only some of them can be considered. This presents an opportunity for optimization. For example, one can preselect the variables that are common in the formula, since common variables are most probable to reduce the formula significantly.
- LookAheadReasoning - performs additional decision heuristics. One example of such is local learning. Executing unit propagation can imply Boolean values for other literals. Local learning uses that fact and extends the formula by adding the learned clauses to the formula. There are other possible heuristics[**7**]: autarky detection and double look-ahead.
- DecisionHeuristic - a procedure that computes the importance of a variable. It takes into consideration the reduction of formula under Boolean value assignments.

The Algorithm 4 might return a simplified formula and a decision variable. The only two cases where it doesn't return a decision variable are when it discovers that the formula is *unsatisfiable* or all preselected variables have been assigned a value. That can mean that either the satisfying assignment has been found or the procedure has to be restarted with a different set of preselected variables.

2.4.3. An example of LookAheadReasoning.

LookAheadReasoning is a procedure executed on a variable and a formula, which it transforms using many possible heuristics. Here we will see an example of the execution of local learning. We saw an example of learning before, but it was global learning in CDCL Solvers. Based on conflict analysis, it extended the formula to an equivalent one by adding clauses, which were valid in the global context. Here the learned clauses will be only valid at a certain point of the search. As soon as the backtracking happens, they have to be removed.

Consider a formula:

$$(x_1 \lor \neg x_2 \lor x_3) \land (x_1 \lor x_2) \land (\neg x_1 \lor \neg x_2 \lor x_3) \land (\neg x_1 \lor \neg x_3 \lor x_4)$$

Assume that we are in a point of search where x_1 is assigned **true**. Assigning **true** to x_2 forces x_3 to be assigned **true**, which then forces **true** to be assigned to x_4. In other words, we know that $x_2 \implies x_4$, which can be described with clause $(\neg x_2 \lor x_4)$. It is important to note, that in general this clause possible depends on all the previous assignments that led to this point in the search - in this case on x_1 being assigned **false**. After backtracking from that choice this learned clause becomes invalid.

Another thing worth noting is that this example of local clause learning could be transformed into global learning by extending the learned implication with a prefix of all the choices that led to this learned clause. Following our example, learned global implication could be $x_1 \implies (x_2 \implies x_4)$, which is equivalent to a clause $(\neg x_1 \lor \neg x_2 \lor x_4)$. Practice shows that this approach is ineffective, and modern Look-Ahead Solvers use locally learned clauses.

A simple extension of that approach is what we call double look-ahead. Instead of considering only one variable, it learns from executing unit propagation after assignments on the two variables. Consider a formula:

$$\begin{gathered}(x_1 \lor \neg x_2) \land (x_1 \lor \neg x_5 \lor x_4) \land (\neg x_3 \lor \neg x_6 \lor \neg x_8) \land (\neg x_6 \lor x_8) \\ \land (x_1 \lor x_5 \lor x_7) \land (\neg x_9 \lor x_5 \lor x_{10}) \land (x_4 \lor \neg x_{10} \lor \neg x_9)\end{gathered}$$

Assume x_1 and x_4 are both assigned **false**.

$$\begin{gathered}(x_1 \lor \neg x_2) \land (x_1 \lor \neg x_5 \lor x_4) \land (\neg x_3 \lor \neg x_6 \lor \neg x_8) \land (\neg x_6 \lor x_8) \\ \land (x_1 \lor x_5 \lor x_7) \land (\neg x_9 \lor x_5 \lor x_{10}) \land (x_4 \lor \neg x_{10} \lor \neg x_9)\end{gathered}$$

Unit propagation forces x_5 and x_2 to be assigned **false**.

$$\begin{gathered}(x_1 \lor \neg x_2) \land (x_1 \lor \neg x_5 \lor x_4) \land (\neg x_3 \lor \neg x_6 \lor \neg x_8) \land (\neg x_6 \lor x_8) \\ \land (x_1 \lor x_5 \lor x_7) \land (\neg x_9 \lor x_5 \lor x_{10}) \land (x_4 \lor \neg x_{10} \lor \neg x_9)\end{gathered}$$

From that it must follow that x_7 is assigned **true**. With that we have learned the implication $(\neg x_1 \land \neg x_4) \implies (\neg x_5 \land \neg x_2 \land x_7)$, which is equivalent to the following clauses:

$$(x_1 \lor x_4 \lor \neg x_5) \land (x_1 \lor x_4 \lor \neg x_2) \land (x_1 \lor x_4 \lor x_7)$$

Every one of them except for the first one is a new one, so with that double look-ahead, two new local clauses can be learned.

2.4.4. An example of DecisionHeuristic.

The DecisionHeuristic procedure is called when both x and $\neg x$ are not failed literals - assigning truth to them doesn't cause conflicts. It ranks the importance of possible decision variables using a number of possible statistics. To do that, it combines the reduction property of the formula after the assignment of each Boolean value. After that the LookAhead procedure decides to use the variable with the highest importance ($H(x)$). One notable decision heuristic is done by measuring the reduction of the formula with a given variable. There are also many ways to measure reduction. Some of them are:

- Number of assignments implied with UnitPropagation
- Number of newly satisfied clauses after assignment.

Consider a formula [**7**]:

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

Assigning values to the variables causes the following implications:

- $x_1 = \mathbf{false} \implies \{x_6 = \mathbf{false}, x_3 = \mathbf{true}\}$
- $x_1 = \mathbf{true} \implies \{x_2 = \mathbf{true}, x_3 = \mathbf{true}, x_4 = \mathbf{true}\}$
- $x_2 = \mathbf{false} \implies \{x_1 = \mathbf{false}, x_3 = \mathbf{true}, x_6 = \mathbf{false}\}$
- $x_2 = \mathbf{true} \implies \{\}$
- $x_3 = \mathbf{false} \implies \{\}$
- $x_3 = \mathbf{true} \implies \{\}$
- $x_4 = \mathbf{false} \implies \{\}$
- $x_4 = \mathbf{true} \implies \{\}$
- $x_5 = \mathbf{false} \implies \{x_4 = \mathbf{true}, x_6 = \mathbf{false}\}$
- $x_5 = \mathbf{true} \implies \{\}$
- $x_6 = \mathbf{false} \implies \{\}$
- $x_6 = \mathbf{true} \implies \{x_1 = \mathbf{true}, x_2 = \mathbf{true}, x_3 = \mathbf{true}, x_4 = \mathbf{true}, x_5 = \mathbf{true}\}$

The calculated importance of the variable has to take into account metrics for both assignments. Here we will consider very simple combining method. When reduction after assignment of **false** has a value a and reduction after assignment of **true** has a value b, then importance of that variable is calculated as $a \cdot b$. Ranking of the variables using the first method goes as follows:

- $H(x_1) = 3 \cdot 4 = 12$
- $H(x_2) = 4 \cdot 1 = 4$
- $H(x_3) = 0$
- $H(x_4) = 0$
- $H(x_5) = 3 \cdot 1 = 3$
- $H(x_6) = 1 \cdot 6 = 6$

Based on that method the variable x_1 has the greatest importance.

We will now consider the second metric. Assume the clauses are labeled from left to right as $c_i, i = 1 \dots 8$. The set on the right side of the implication will contain newly satisfied clauses.

- $x_1 = \mathbf{false} \implies \{c_1, c_2, c_3, c_4, c_5, c_6, c_8\}$
- $x_1 = \mathbf{true} \implies \{c_1, c_2, c_3, c_4, c_5, c_6, c_7\}$

- $x_2 =$ **false** $\implies \{c_1, c_2, c_3, c_4, c_5, c_6, c_8\}$
- $x_2 =$ **true** $\implies \{c_1\}$
- $x_3 =$ **false** $\implies \{c_3\}$
- $x_3 =$ **true** $\implies \{c_2, c_4\}$
- $x_4 =$ **false** $\implies \{\}$
- $x_4 =$ **true** $\implies \{c_3, c_5, c_7\}$
- $x_5 =$ **false** $\implies \{c_3, c_5, c_6, c_7, c_8\}$
- $x_5 =$ **true** $\implies \{c_7, c_8\}$
- $x_6 =$ **false** $\implies \{c_6, c_8\}$
- $x_6 =$ **true** $\implies \{c_1, c_2, c_3, c_4, c_5, c_6, c_7\}$

Assume the same importance combining method as before.

- $H(x_1) = 7 \cdot 7 = 49$
- $H(x_2) = 7 \cdot 1 = 7$
- $H(x_3) = 1 \cdot 2 = 2$
- $H(x_4) = 0 \cdot 3 = 0$
- $H(x_5) = 5 \cdot 2 = 10$
- $H(x_6) = 2 \cdot 7 = 14$

This heuristic results in the same choice for this formula - variable x_1 has the highest importance again. Of course, that is not always the case. There are also other more advanced heuristics.

2.5. Random Satisfiability

In this section, the formulae are restricted to be in the 3-CNF (all clauses have three literals).

In the previous sections, we explored two examples of complete algorithms for solving CNF-SAT instances. Now we will delve into a Monte Carlo algorithm for solving 3-CNF SAT instances. A Monte Carlo algorithm is an algorithm whose output might be incorrect with some probability. Here we will allow false negatives - an algorithm may return **UNSAT**, even though the formula is *satisfiable*.

Algorithm 5 Random 3-SAT algorithm[**14**] // n - length of formula, m - arbitrary constant

```
procedure 3-SAT($\Phi$)
    $i \leftarrow 0$
    repeat
        $v \leftarrow$ RandomAssignment($\Phi$)
        $j \leftarrow 0$
        repeat
            $\{l_1, l_2, l_3\} \leftarrow$ UnsatisfiedClause($\Phi, v$)
            $x \leftarrow$ RandomVariable($l_1, l_2, l_3$)
            $v \leftarrow v[x = \neg v(x)]$
            $j \leftarrow j + 1$
        until a satisfying assignment is found or $j \geq 3n$
        $i \leftarrow i + 1$
    until all clauses are satisfied or $i \geq m$
    if a satisfying truth assignment has been found then
        return SAT
    else
        return UNSAT
    end if
end procedure
```

Where $v[x = \mathbf{B}]$ is an assignment equal to v, except for a value of x, which is assigned **B**. The procedures used within the algorithm work as follows:

- RandomAssignment - returns a truth assignment chosen uniformly at random.
- UnsatisfiedClause - returns any clause from Φ unsatisfied under an assignment v.
- RandomVariable - chooses a variable at random from the specified literals.

It is clear why this algorithm is Monte Carlo - the only possible mistake it can make is when there is a satisfying assignment, but it is unable to find it in the specified number of steps. The inner loop modifies the current assignment by finding any unsatisfied clause and changes the truth value in v of the variable in any literal. That makes that clause, previously unsatisfied, satisfied. Of course, it might influence the satisfiability of other clauses. The intuition behind the outer loop is that in the case of random assignments 3-SAT formulae, the strategy from the inner loop is more probable to diverge from the satisfying assignment. For that reason, the assignment is reset, and that process is executed m times.

Of course, with 3-SAT being an NP-complete problem, it is unexpected that there exists a randomized algorithm working in polynomial time (with expected polynomial running time) [**14**]. Algorithm 5 optimizes the exponent base of the expected time complexity. The complexity of the algorithm, that checks all possible assignments, has a complexity of $O(2^n \cdot n)$, where n is the length of the formula. Probabilistic analysis [**14**] shows that the expected complexity of the algorithm 5 is $O((\frac{4}{3})^n \cdot n^{\frac{3}{2}})$. One thing that should also be considered, is the probability of the algorithm making a mistake. It is dependent on the m parameter used in the algorithm. It follows that if a is an upper bound of the expected number of steps until a satisfying assignment is found, and m is set to $2ab$, then the probability of the algorithm making a mistake (returning **UNSAT** when the formula is *satisfiable*) is bounded from above by 2^{-b} [**14**].

CHAPTER 3

MAX-SAT

3.1. Definition

Definition 3.1. Let $s_A(v)$ be the number of satisfied clauses in a formula A in CNF under a truth assignment v.

Definition 3.2. The MAX-SAT problem is the problem of finding the value of $\max\{s_A(v)\}$ over all possible truth assignments v for a given formula A.

The MAX-SAT problem is an optimization version of the SAT problem. Instead of determining only if the formula is *satisfiable*, it solves the problem of maximizing the number of clauses satisfied. Of course, the formula is *satisfiable* if and only if there is an assignment that satisfies all the clauses. That fact shows that the MAX-SAT problem is a generalization of the CNF-SAT problem. In order to solve the SAT problem for a given formula, we can simply solve the MAX-SAT problem for the same formula and check whether its result matches the length of that formula.

There are also other interesting variations of the MAX-SAT problem, including Weighted MAX-SAT and Partial MAX-SAT:

The weighted MAX-SAT is an extension of the MAX-SAT problem that also assigns weights to all the clauses. It then searches to maximize the sum of weights of the satisfied clauses. It is worth noting that the initial MAX-SAT problem can be defined as a weighted MAX-SAT with each clause having weight 1.

The partial MAX-SAT distinguishes two kinds of clauses: *soft* and *hard*. It is a problem of finding an assignment that satisfies all *hard* clauses and maximizes the number of satisfied *soft* clauses. In other words, it is a mix of the SAT problem and the weighted MAX-SAT.

3.1.1. Example of MAX-SAT.

Consider a formula:

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

With the corresponding truth table:

x_1	x_2	x_3	$(x_1 \vee x_2) \wedge (x_3 \vee \neg x_1)$	number of satisfied clauses
false	**false**	**false**	**false**	1
false	**false**	**true**	**false**	1
false	**true**	**false**	**true**	2
false	**true**	**true**	**true**	2
true	**false**	**false**	**false**	1
true	**false**	**true**	**true**	2
true	**true**	**false**	**false**	1
true	**true**	**true**	**true**	2

The resulting maximum number of satisfied clauses is two.

3.2. The naive algorithm

Algorithm 6 MAX-SAT algorithm

```
procedure MAX-SAT($\Phi$)
    if $\Phi = \emptyset$ or $\Phi$ only contains empty clauses then
        return 0
    end if
    $x \leftarrow$ choose variable in $\Phi$
    $f \leftarrow$ NEWLYSATISFIED($\Phi, \Phi[x = \mathbf{false}]$) + MAX-SAT($\Phi[x = \mathbf{false}]$)
    $t \leftarrow$ NEWLYSATISFIED($\Phi, \Phi[x = \mathbf{true}]$) + MAX-SAT($\Phi[x = \mathbf{true}]$)
    return max($f, t$)
end procedure
```

The NEWLYSATISFIED procedure returns a number of clauses that become satisfied after assignment. We are assuming set representation of formulas, so it can be computed as $|\Phi| - |\Phi[x = \mathbf{B}]|$. As a remainder, the assignment $\Phi[x = \mathbf{B}]$ removes satisfied clauses and all falsified literals (x or $\neg x$ depending on the value of $\mathbf{B}$).

The complexity of this approach to solving the MAX-SAT problem is $O(2^n \cdot n)$. This approach is similar to the DPLL algorithm but extended by counting satisfied clauses instead of returning **false** or **truth** values.

3.3. Branch and bound approach

In this section, we will explore a Branch and bound algorithm for solving the MAX-SAT problem. Branch and bound can be defined as an algorithm design paradigm for solving optimization problems. It describes the solution space as a tree with edges representing different choices and nodes representing choice points (in our case variables). When considering a sub-tree rooted in a node, it calculates an upper bound for all the possible scores of this sub-tree. Using that estimation it can then discard branches that yield only worse solutions than the ones already found. In our problem, the sub-tree rooted in a node contains all possible extensions of the current assignment. It discards the sub-tree if it knows that no possible extension of the current partial assignment can be better than one previously found.

From now on, we will consider an equivalent approach to MAX-SAT. Instead of finding a maximum number of satisfied clauses, we will find a minimum number of unsatisfied ones.

3.3.1. The algorithm.

Algorithm 7 A basic branch and bound algorithm for MAX-SAT[**12**] // *UB* - upper bound

```
procedure MAX-SAT($\Phi$, $UB$)
    $\Phi \leftarrow$ simplifyFormula($\Phi$)
    if $\Phi = \emptyset$ or $\Phi$ only contains empty clauses then
        return #emptyClauses($\Phi$)
    end if
    $LB \leftarrow$ #emptyClauses($\Phi$) + underestimation($\Phi$)
    if $LB \geq UB$ then
        return $UB$
    end if
    $x \leftarrow$ selectVariable($\Phi$)
    $UB \leftarrow \min(UB, \text{MAX-SAT}(\Phi[x = \textbf{false}], UB))$
    return $\min(UB, \text{MAX-SAT}(\Phi[x = \textbf{true}], UB))$
end procedure
```

- *UB* - an upper bound for the number of unsatisfied clauses in an optimal solution. During the running of the algorithm, it will become the greatest number of unsatisfied clauses for complete assignments known. It can be first initialized with the length of the initial formula.
- #emptyClauses - returns a number of empty (unsatisfied) clauses of a given formula.
- underestimation - calculates an underestimation (a lower bound) for a number of currently non-empty clauses, that could become unsatisfied after extending the current partial assignment.
- *LB* - a lower bound for a number of unsatisfied clauses for any possible extensions of current partial assignment to a complete one. It is calculated for the current branch by combining a number of unsatisfied clauses in the current formula with a result of the underestimation procedure.
- selectVariable - returns a variable from Φ. Like with previous SAT Solvers, it can be implemented with many possible heuristics.
- simplifyFormula - transforms the formula into an equivalent one. It can use a number of inference rules to help us learn more about it. An example of this approach was seen before with global and local learning approaches.

First **if** statement handles the base case - current assignment is complete. If that is not the case, the algorithm calculates a lower bound for the score in the current branch. If that bound happens to be greater or equal to the upper bound, then we know that it is not better than the solution already found - remember that we are trying to minimize the score. Otherwise, the optimal solution might be in the current sub-tree, so the algorithm will recursively explore two possible assignments, and return the minimum of upper bounds for both of them.

3.3.2. The lower bound.

The performance of this algorithm heavily depends on the heuristics used for choosing the variable and the formula for calculating a lower bound. For example, if the lower bound would always end up being equal to 0, then the algorithm would have to explore the whole solution tree anyway. Here we will see possible examples of the underestimation procedure.

$$LB \leftarrow \#\text{EMPTYCLAUSES}(\Phi) + \text{UNDERESTIMATION}(\Phi)$$

One possible implementation of UNDERESTIMATION is to always return 0. That approach is used in [**3**]. The lower bound would be then calculated as a number of unsatisfied clauses with current partial assignment:

$$LB \leftarrow \ \#\text{EMPTYCLAUSES}(\Phi)$$

More elaborate method calculates UNDERESTIMATION by analysing unit clauses in the formula [**18**]:

$$\text{UNDERESTIMATION}(\Phi) = \sum_{x \in Var(\Phi)} \min\left(\text{ic}(x), \text{ic}(\neg x)\right)$$

Where $\text{ic}(l)$ is an inconsistency count of literal l and can be defined as a number of unit clauses $\{\neg l\}$ in Φ. Notice that the expression $\min\left(\text{ic}(x), \text{ic}(\neg x)\right)$ calculates a lower bound for a number of newly unsatisfied clauses after extending current assignment with a value for x.

Consider a formula:

$$\Phi := (y \vee x) \wedge (x) \wedge (\neg x) \wedge (x) \wedge (z \vee \neg x \vee \neg z) \wedge ()$$

A lower bound would then be calculated as:

$$LB = \#\text{EMPTYCLAUSES}(\Phi) + \sum_{v \in Var(\Phi)} \min\left(\text{ic}(v), (\text{ic}(\neg v)\right) = 1 + \min\left(\text{ic}(x), (\text{ic}(\neg x)\right) = 2$$

3.3.3. Resolution.

Here we will consider an example of what the SIMPLIFYFORMULA might do. Consider very straightforward resolution on literals:

$$\frac{x \vee l_1 \quad \neg x \vee l_2}{l_1 \vee l_2}$$

In a case of normal SAT problem we could use this rule to learn (conclude) a new clause $l_1 \vee l_2$. On the other hand, this approach cannot be straightforwardly used in MAX-SAT solving. To see that, consider two formulas:

$$(x \vee x_1) \wedge (\neg x \vee x_2) \text{ and } (x \vee x_1) \wedge (\neg x \vee x_2) \wedge (x_1 \vee x_2)$$

Consider an assignment $x = x_1 = x_2 = \textbf{false}$. The former clause has one unsatisfied clause, while the latter has two. That observation shows that these two formulas differ in terms of a number of unsatisfied clauses under the same assignment. It means that they are not equivalent with respect to the MAX-SAT problem.
Because of that, we need to extend that resolution approach by adding additional clauses, that compensate for logically implied clauses and preserve equivalence in terms of the MAX-SAT problem:

$$\begin{array}{c} x \vee l_1 \\ \neg x \vee l_2 \end{array} \underset{\text{MAX-SAT}}{\Longleftrightarrow} \begin{array}{r} l_1 \vee l_2 \quad \} \text{ resolvent} \\ \left.\begin{array}{r} x \vee l_1 \vee \neg l_2 \\ \neg x \vee \neg l_1 \vee l_2 \end{array}\right\} \text{compensation clauses} \end{array}$$

Let $\Phi_1 = (x \vee l_1) \wedge (\neg x \vee l_2)$ and $\Phi_2 = (l_1 \vee l_2) \wedge (x \vee l_1 \vee \neg l_2) \wedge (\neg x \vee \neg l_1 \vee l_2)$.

x	l_1	l_2	# of unsatisfied clauses in Φ_1	# of unsatisfied clauses in Φ_2
false	**false**	**false**	1	1
false	**false**	**true**	1	1
false	**true**	**false**	0	0
false	**true**	**true**	0	0
true	**false**	**false**	1	1
true	**false**	**true**	0	0
true	**true**	**false**	1	1
true	**true**	**true**	0	0

That analysis shows that this resolution rule preserves the equivalence. Notice that it changes the maximum number of satisfied clauses, but that does not matter since transforming the formula using this changes the size of the formula, which is taken into consideration when converting between the two approaches.

This approach can be generalized to clauses of arbitrary length [**1, 2, 6**]:

$$\begin{array}{c} x \vee l_1 \vee \ldots \vee l_m \\ \neg x \vee o_1 \vee \ldots \vee o_n \end{array} \underset{\text{MAX-SAT}}{\Longleftrightarrow} \begin{array}{l} l_1 \vee \ldots \vee l_m \vee o_1 \vee \ldots \vee o_n \\ x \vee l_1 \vee \ldots \vee l_m \vee \neg o_1 \\ x \vee l_1 \vee \ldots \vee l_m \vee o_1 \vee \neg o_2 \\ \vdots \\ x \vee l_1 \vee \ldots \vee l_m \vee o_1 \vee \ldots \vee o_{n-1} \vee \neg o_n \\ \\ \neg x \vee o_1 \vee \ldots \vee o_n \vee \neg l_1 \\ \neg x \vee o_1 \vee \ldots \vee o_n \vee l_1 \vee \neg l_2 \\ \vdots \\ \neg x \vee o_1 \vee \ldots \vee o_n \vee l_1 \vee \ldots \vee l_{m-1} \vee \neg l_m \end{array}$$

3.4. Approximation algorithms

When tackling hard algorithmic challenges, we can sometimes give up the accuracy of the result in favor of better complexity. In this paper, we saw an example of that when considering a randomized algorithm for the 3-CNF SAT problem.

For approximation algorithms solving optimization problems, we call an algorithm f an $\alpha-$approximation of a problem X, when for any instance of the problem X a solution returned by f is within a factor α of the optimal solution. When thinking about randomized algorithms, we can alter that definition so that it requires an expected value of the returned solution to be within a factor α of the optimal solution.

Algorithm 8 A non-randomized $\frac{1}{2}-$approximation algorithm for MAX-SAT

```
procedure MAX-SAT($\Phi$)
    $v_F, v_T \leftarrow \emptyset$                          ▷ Initialize assignments
    for all $x \in Var(\Phi)$ do
        $v_F \leftarrow v_F[x = \mathbf{false}]$
        $v_T \leftarrow v_T[x = \mathbf{true}]$
    end for
    $F \leftarrow$ number of satisfied clauses under $v_F$
    $T \leftarrow$ number of satisfied clauses under $v_T$
    return max($F, T$)
end procedure
```

Lemma 3.3. *The algorithm* 8 *always returns a solution equal to at least* $\frac{1}{2}$ *of the optimal solution.*

Proof. Let m be the value of the optimal solution (a maximum number of satisfied clauses), and let n be the length of the formula Φ.

Every clause in Φ must be satisfied under v_F or v_T, so it follows that $F + T \geq n \geq m$. From that we can conclude that max$(F, T) \geq \frac{m}{2}$. □

We can also consider an equally interesting $\frac{1}{2}-$approximation randomized algorithm:

Algorithm 9 A randomized $\frac{1}{2}-$approximation algorithm for MAX-SAT

```
procedure MAX-SAT($\Phi$)
    $v \leftarrow \emptyset$                          ▷ Initialize an assignment
    for all $x \in Var(\Phi)$ do
        if RANDOMBIT() = 0 then
            $v \leftarrow v[x = \mathbf{false}]$
        else
            $v \leftarrow v[x = \mathbf{true}]$
        end if
    end for
    return a number of satisfied clauses in $\Phi$ under $v$
end procedure
```

Lemma 3.4. *The returned value of an algorithm* 9 *has an expected value equal to at least* $\frac{1}{2}$ *of the optimal solution.*

PROOF. Let C_i be equal to 1 when clause i is satisfied and 0 otherwise. Let n be equal to the length of the formula Φ and let l_i be the size of clause i.

$$\begin{aligned}
&E[\text{number of satisfied clauses in } \Phi] \\
&= \sum_{i=1}^{n} E[C_i] \\
&= \sum_{i=1}^{n} P(\text{clause } i \text{ is satisfied}) \\
&= \sum_{i=1}^{n} \left(1 - \left(\frac{1}{2}\right)^{l_i}\right) \\
&\geq \sum_{i=1}^{n} \left(1 - \frac{1}{2}\right) \\
&= \frac{n}{2}
\end{aligned}$$

The value of the optimal solution is at most equal to the length of the formula, so it follows that the expected value is greater or equal to $\frac{1}{2}$ of the optimal solution. □

It has been proven that unless $P = NP$, no $\alpha-$approximation algorithms solving MAX-SAT problem with $\alpha > \frac{7}{8}$ exists [**8**]. We also know that this bound is tight - there exists an $\frac{7}{8}-$approximation algorithm [**9**].

Bibliography

[1] María Luisa Bonet, Jordi Levy, and Felip Manyà. A complete calculus for max-sat. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006*, pages 240–251, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[2] María Luisa Bonet, Jordi Levy, and Felip Manyà. Resolution for max-sat. *Artificial Intelligence*, 171(8):606–618, 2007.

[3] Brian Borchers and Judith Furman. A Two-Phase Exact Algorithm for MAX-SAT and Weighted MAX-SAT Problems. *Journal of Combinatorial Optimization*, 2(4):299–306, December 1998.

[4] Daniel Brooks, Esra Erdem, Selim Erdoğan, James Minett, and Don Ringe. Inferring phylogenetic trees using answer set programming. *Journal of Automated Reasoning*, 39:471–511, 11 2007.

[5] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing - STOC '71*. ACM Press, 1971.

[6] Federico Heras and Javier Larrosa. New inference rules for efficient max-sat solving. In *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1*, AAAI'06, page 68–73. AAAI Press, 2006.

[7] Marijn Heule and Hans Maaren. Look-ahead based sat solvers. *Frontiers in Artificial Intelligence and Applications*, 185, 01 2009.

[8] Johan Håstad. Some optimal inapproximability results. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '97, page 1–10, New York, NY, USA, 1997. Association for Computing Machinery.

[9] H. Karloff and U. Zwick. A 7/8-approximation algorithm for max 3sat? In *Proceedings 38th Annual Symposium on Foundations of Computer Science*, pages 406–415, 1997.

[10] Henry Kautz and Bart Selman. Planning as satisfiability. pages 359–363, 01 1992.

[11] Donald E Knuth. *The art of computer programming, volume 4, fascicle 6*. Addison-Wesley Educational, Boston, MA, December 2015.

[12] Chu-Min Li and Felip Manyà. Maxsat, hard and soft constraints. *Frontiers in Artificial Intelligence and Applications*, 185, 01 2009.

[13] Inês Lynce and João Marques-silva. Efficient haplotype inference with boolean satisfiability. *Proceedings of the National Conference on Artificial Intelligence*, 1, 01 2006.

[14] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis*. Cambridge University Press, USA, 2nd edition, 2017.

[15] João P. Marques Silva and Karem A. Sakallah. Grasp—a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '96, page 220–227, USA, 1997. IEEE Computer Society.

[16] João Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning sat solvers. *Frontiers in Artificial Intelligence and Applications*, 185, 01 2009.

[17] G. S. Tseitin. On the complexity of derivation in propositional calculus. In Jörg H. Siekmann and Graham Wrightson, editors, *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*, pages 466–483. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983.

[18] R. J. Wallace and E. Freuder. Comparative studies of constraint satisfaction and davis-putnam algorithms for maximum satisfiability problems. *Cliques, Coloring and Satisfiability*, 26:587–615, 1996.