



Seminar Report

Propositional Logic - Algorithms, Data Structures and Theorems

Julian Parsert

`julian.parsert@student.uibk.ac.at`

27 February 2017

Supervisor: Sebastiaan Joosten

Abstract

The first part mainly revolves around satisfiability checking for propositional logic. At first we introduce the most widely spread algorithm DPLL and its preceding more naïve algorithm, the DP procedure. Subsequently, we describe and demonstrate a less popular algorithm called Stålmarck's method.

Due to the importance of SAT in complexity theory, we also briefly mention the complexity of SAT in general and especially *3-Satisfiability*. In addition to SAT solving algorithms we also introduce *Binary Decision Diagrams*. First, we explain the theory using an example. In the second part, we show parts of the implementation presented in John Harrison's book. The last section is based on the Compactness of *propositional logic* and its results.

Contents

1	Introduction	1
2	Satisfiability	1
2.1	Davis-Putnam(DP)	1
2.2	Davis-Putnam-Logemann-Loveland(DPLL)	2
2.3	k-CNF and Complexity	4
2.4	Stålmarck's method	4
3	Binary Decision Diagrams	7
3.1	Theory	7
3.2	Implementation	8
4	Compactness	9
5	Discussion	10
	Bibliography	11
	References	11

1 Introduction

This report gives an overview of Chapters 2.9 to 2.12 of John Harrison's "*Handbook of Practical Logic and Automated Reasoning*". These chapters give an understanding of algorithms, data structures and theorems based on or representing propositional logic. At first we explore methods for determining the satisfiability of formulas. Further, we have a look at *Binary Decision Diagrams*. And last we have a look at the *Compactness theorem* for propositional logic.

2 Satisfiability

Since many problems can be reduced to, or expressed in propositional logic, SAT is one of the most important problems in computer science. Also, SAT is not only at the core of the infamous $P \stackrel{?}{=} NP$ problem [6], but despite the fact that there is no known efficient algorithm¹, attempts are continuously being made at optimizing SAT solving algorithms. Hence, a lot of research is also put into determining complexities of said algorithms. Most algorithms assume the input formula to be in *conjunctive normal form* (CNF). A formula is in CNF if it is a conjunction of *clauses*, where each *clause* is a disjunction of (negated) atoms. The advantage of CNFs is that any formula can be transformed to an *equisatisfiable*² CNF, where the resulting CNF is only a few times larger than our original formula [2]. This is a desired property, especially since the run time increases as a function of input length.

John Harrison utilises the commonly used "sets of sets" representation of a CNF, where \top and \perp represent *True or satisfiable* and *False or not satisfiable* respectively, an empty set represents \top and a nonempty set containing an empty set means that our formula is *not satisfiable* [2]. An advantage of that representation is the implicit removal of duplicates.

2.1 Davis-Putnam(DP)

The Davis-Putnam (DP) algorithm was the first of its kind, developed by Martin Davis and Hilary Putnam. The basic idea of the DP procedure is transforming our input CNF to either an empty set (\top) or a non empty set containing an empty clause (\perp), using *satisfiability preserving transformations*, which are the following:

1-Literal Rule: This rule is applied when our CNF contains a *unit clause*. Meaning there is a set containing only one atom. Applying this rule entails eliminating all other clauses containing said atom, while eliminating all occurrences of the negated atom in further clauses. This rule, when applied, will always decrease the size of our formula, in case it contains a unit clause and otherwise will at not increase the size.

Affirmative-Negative rule: Sometimes also referred to as *pure literal rule*. This step describes the elimination of all atoms that exclusively appear either positive or

¹"Efficient algorithm" here means, algorithm with sub exponential run time complexity.

²Two formulas ϕ and ψ are equisatisfiable if ϕ is satisfiable if and only if ψ is.

2 Satisfiability

negative.

Resolution rule: In addition to the set of clauses, this rule also receives an atom as a parameter. Every atom in consideration for resolution must be contained positively and negatively at least once in the input. The theoretical basis of this rule lies within Theorem 1.

Theorem 1 (2.11). Given a literal p and a set S of clauses where

$$S = \{p \vee C_i \mid 1 \leq i \leq m\} \cup \{\neg p \vee D_j \mid 1 \leq j \leq n\} \cup S_0 \quad (1)$$

and

$$S' = \{C_i \vee D_j \mid 1 \leq i \leq m, 1 \leq j \leq n\} \cup S_0 \quad (2)$$

with

$$(p \in C_i \iff \neg p \in C_i) \wedge (p \in D_i \iff \neg p \in D_i).$$

we can say that S (our input) is satisfiable if and only if S' is [2].

This gives us the ability to eliminate atoms from the CNF, while preserving the satisfiability. Meaning that we can create a new *equisatisfiable* set of clauses S' (Eq. (2)), from our old set S (Eq. (1)) with the guarantee that the new CNF contains one atom less. It is important to note however, that although there are fewer atoms, we can potentially still end up with exponentially more clauses.

Listing 1: Resolution function

```
1 let resolve_on p clauses =
2   let p' = negate p and pos,notpos = partition (mem p) clauses in
3   let neg,other = partition (mem p') notpos in
4   let pos' = image (fun l -> l <> p) pos
5   and neg' = image (fun l -> l <> p') neg in
6   let res0 = allpairs union pos' neg' in
7   union other (filter (non trivial) res0);;
```

Listing 1 shows the implementation of a function that performs resolution on a set *clauses* and an atom p . It is clearly evident that, while the first two lines separate the input set to C_i and D_i as in Eqs. (1) and (2), the remaining part simply filters out the atoms and performs the union operation that results in S' as in Eq. (2).

2.2 Davis-Putnam-Logemann-Loveland(DPLL)

The first improvement made was to replace the resolution rule explained in Section 2.1 with a rule that performs a case split. This means that some literal p is chosen and the satisfiability of the clause set Δ is reduced to the satisfiability of either $\Delta_1 \cup \{\neg p\}$ or $\Delta_2 \cup \{p\}$ [2]. These can be checked separately, as can be seen in Listing 2.

As one can see in Listing 2, the algorithm still makes use of the *One Literal Rule* and the *Affirmative Negative Rule* (Section 2.1), helping to reduce the total number of clauses left.

Listing 2: Basic function performing DPLL.

```

1 let rec dpll clauses =
2   if clauses = [] then true else if mem [] clauses then false else
3   try dpll(one_literal_rule clauses) with Failure _ ->
4   try dpll(affirmative_negative_rule clauses) with Failure _ ->
5   let pvs = filter positive (unions clauses) in
6   let p = maximize (posneg_count clauses) pvs in
7   dpll (insert [p] clauses) or dpll (insert [negate p] clauses);;

```

Listing 3: Improved DPLL using Tail Recursion and Backtracking

```

1 let rec dpli cls trail =
2   let cls', trail' = unit_propagate (cls, trail) in
3   if mem [] cls' then
4     match backtrack trail with
5     (p, Guessed)::tt -> dpli cls ((negate p, Deduced)::tt)
6     | _ -> false
7   else
8     match unassigned cls trail' with
9     [] -> true
10    | ps -> let p = maximize (posneg_count cls') ps in
11            dpli cls ((p, Guessed)::trail');;

```

The code shown in Listing 2, is the most basic implementation of the DPLL procedure. However, are multiple ways this algorithm can be improved upon. Hence, some improvements will be shown.

Tail Recursion This can frequently be used in functional programming, in order to improve both *run time* and *memory allocations*. This is also the case with this algorithm. It is possible to continuously accumulate a list that contains information about the chosen literals. Once we derive \perp , we retreat to our last case split, and “choose” the other (negated) case. This is called *backtracking*. The use of this can be seen in Listing 3. The trail argument is the *accumulator*, and whenever an empty clause is derived we make use of *backtracking*.

Backjumping This can be seen as an extension of backtracking. Once a conflict is derived, we can utilize a *conflict graph* that can help us pin down the case split that lead to the current conflict. This means, that we are able to not only move back to the last case split as we did using *backtrack*, but we can go back multiple case splits at once. This helps us find the key case split that lead to our failure, ensuring that the same fail will not occur again.

Learning *Learning* is a procedure that adds a clause C , that contains only literals that also occur in the original formula, to said formula [4]. This can be done for multiple reasons, one being the occurrence of certain structures in a CNF, that will likely lead to a conflict at a later point. Adding a specially tailored clause can force an immediate conflict, instead of “wasting” time on a branch that will inevitably lead to a conflict. Corollary 2.1 shows how this can improve the algorithm dramatically.

Example 2.1. Given CNF (ψ) in clausal form:

$$\{\{\neg p_1, \neg p_2, \neg p_{100}\}, \{\neg p_1, \neg p_2, p_{100}\} \cup D\}$$

Let the indexing indicate that atom p_i is eliminated at the iteration i of our algorithm. Further assume that D does not contain any of $\{p_1, p_2, p_{100}\}$. As we can see in our constructed example, the algorithm will set p_1 and p_2 false, and eventually 98 steps later will reach a conflict. By adding $\{\neg p_1, \neg p_2\}$ to our CNF(ψ) we can save 98 steps by deriving a conflict immediately.

2.3 k-CNF and Complexity

k -CNF are specific form of CNF where each clause contains at most k atoms. As it turns out there exist algorithms that can solve 2-SAT in polynomial time [2]. However, 2-SAT is incomplete, which means that only a proper subset of propositional formulas can be transformed to an equisatisfiable 2-CNF. It can be shown, that an arbitrary propositional logic formula can be transformed to a k -CNF where $k > 2$, making 3-SAT the smallest k -SAT while still being complete. Therefore most complexity analysis is done on 3-SAT.

Due to the extensive use and applications of SAT across many fields, there are continued attempts, not only at finding a polynomial time algorithm, but also at improving upon the existing algorithms. The most naïve implementation of a SAT solving algorithm³, has a complexity of $O(2^n)$ for a formula with n atoms. Many successful attempts were made at improving the algorithms to reach complexities $O(2^{kn})$ where $k < 1$. Ramamohan Paturi et al. proposed an algorithm which is based on a randomised search, which has a time complexity of $O(2^{0.378...n})$ [5].

2.4 Stålmarck's method

Currently, most SAT-solvers are based on a *DPLL*. However, as mentioned in Section 2.3, there are also other approaches to this problem. One approach is Stålmarck's method, which is rarely used as it is being protected by patent law.

Since Stålmarck's method at its core tries to prove validity of a formula and a formula is satisfiable if and only if its negation is not valid, we can negate our input formula and use Stålmarck's method to show that it is not valid. Therefore proving our original formula to be at least satisfiable [7].

The graph displayed in Fig. 1 shows a graphical illustration of the procedure. At first Stålmarck's method repeatedly tries to apply the *simple rules*, which we will introduce further down, to the set Γ of triples (first line in Fig. 1). In case we managed to derive a contradiction we are done. Otherwise, once no further *simple rule* can be applied, hence no logical consequences can be reached anymore, the *dilemma rule* is applied (second line). This rule performs a branching on an atom p . For each of the branches we again, apply the *simple rules* as often as possible (transition from line 2 to 3 resulting in substitutions

³For example, a search through a truth table.

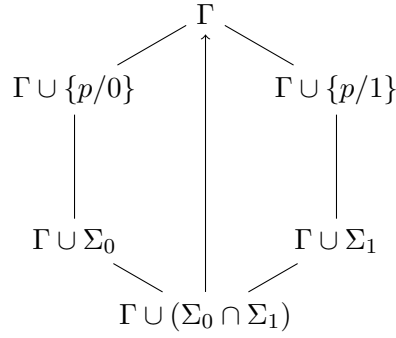


Figure 1: Illustration of algorithm.

Σ_0 and Σ_1). If we derive a contradiction in one of the branches, the substitutions of the other branch are applied to our set of triples $(\Gamma \cup \Sigma_i)$. In case both branches reach a contradiction, we are done. Finally, if none of the branches derive contradictions this means that any substitution gathered in both branches (i.e. the intersection of the substitutions) must hold independently of p or $\neg p$ and can therefore be applied to our set (line 4). Note that is comparable to the resolution in Section 2.1 [7]. From here on, with our updated set of triples, we start from the beginning again, by applying our *simple rules* [2].

Now that we have laid out the theoretical basis for the method, we can introduce the previously explained rules. As mentioned, the procedure is based on having the input formula in a special format which consists of triples. These triples express structures in propositional logic. We define our interpretation such that (x, y, z) denotes $x \iff (y \wedge z)$. It is worth noting, that when implementing this algorithm, any interpretation can be chosen as long as it is adequate⁴. Having fixed our interpretation we can now derive this set of *simple rules*,

$$\frac{(x, y, 1)}{x/y} \quad (3)$$

$$\frac{(x, 0, z)}{x/0} \quad (4)$$

$$\frac{(x, y, 0)}{x/0} \quad (5)$$

$$\frac{(1, y, z)}{y/1, z/1} \quad (6)$$

$$\frac{(x, y, z) \text{ where } x \equiv \neg y}{x/0, y/1, z/0} \quad (7)$$

⁴A set of functions is adequate if for every formula there is an equivalent formula with only connectives from that set [3].

2 Satisfiability

and the generic *dilemma rule*,

$$\frac{\Gamma}{\Gamma[x/1] \quad \Gamma[x/0]} \quad (8)$$

where Γ is a set of triples and x/y denotes a substitution of x by y

Recalling the definition of our triples, it is quite logical how these rules can be derived. Take Eq. (3) for instance, which is simply utilizing the *conjunctive identity*. As one might be able to see, there exist further possible inferences, however, by enforcing conventions (e.g. $(x, 1, z)$ is changed to $(x, z, 1)$ by commutativity) the rules Eqs. (3) to (7) will suffice. This can be demonstrated by listing all the possible states a triple can be in and ruling out the ones, where additional information cannot be gathered with certainty, and the ones which we avoid using aforementioned conventions. Take $(0, x, y)$ for example, where it is impossible determine which of x , y , or both are false. Similar problems occur for (x, y, z) , where even less information can be gathered. In the end, the rules remaining will be the ones used in Eqs. (3) to (7).

These rules form the basis for this procedure. The *simple rules* are applied during the simplification phases, where we search for triples with the same layout as the antecedents in Eqs. (3) to (7). If we find a fitting triple, we apply the substitution shown in the consequent of the rule, to all triples in the current set. For example, assuming we have a triple $(1, b, a)$ in our set of triples, we apply Eq. (6) and replace all occurrences of a and b with 1. Remember that this follows, since the first element can only be 1 if and only if $a \equiv b \equiv 1$.

Example 2.2 (We will use Stålmarck's method to show that $\phi = a \wedge \neg c \wedge (d \wedge \neg d)$ is not satisfiable). Since the algorithm requires the input to have this special format of triples, we need to transform our formula first. This requires us to introduce new variables x_0 to x_2 and assign them to the sub trees of the parse tree as shown in Fig. 2. This results in the following set of triples $\{(x_0, d, \neg d), (x_1, x_0, \neg c), (x_2, x_1, a)\}$.

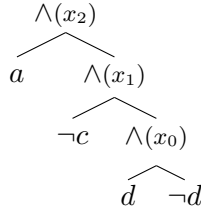


Figure 2: Parse tree of formula with assigned variables x_0 to x_2 .

Now that our input has the right format, we can commence with the procedure. As one can see, x_2 represents the entire formula. So setting x_2 to \top or 1 we should be able to derive a contradiction.

As we can see in Table 1, we derived a contradiction in *both* branches as a result of applying the *dilemma rule*. Therefore our original assumption, that the formula ϕ is valid has proven to be false.

	1.		2.	
$(x_0, d, \neg d)$	Eq. (6) hence substitutions $x_1/1, a/1$	$(x_0, d, \neg d)$	Eq. (6) hence substitutions $x_0/1, c/0$	$(1, d, \neg d)$
$(x_1, x_0, \neg c)$		$(1, x_0, \neg c)$		$(1, 1, 1)$
$(1, x_1, a)$		$(1, 1, 1)$		$(1, 1, 1)$
	3.	d/1	d/0	
	split on d : $d/1, d/0$	$(1, 1, 0),$	$(1, 0, 1)$	
		ζ	ζ	

Table 1: Applying Stålmarck’s method to our 3 triples using Eqs. (3) to (7).

3 Binary Decision Diagrams

3.1 Theory

Binary Decision Diagrams, also BDDs, are binary tree representations of propositional logic formulas. Each node represents an atom. A path from the root to a leaf denotes a valuation of each of the atoms visited in the path. If we have reached the \top or \perp leaf it means that our current valuation evaluates to true or false respectively. Given a formula we can build a BDD from it. In addition we can also optimize or reduce and order the tree, resulting in a *reduced ordered binary decision diagram* [1]. Reducing a *BDD* entails the removal of duplicate subtrees. In addition we can also replace nodes that have no impact on the final value of the formula, with their subtree. In other words, we can replace all nodes where the outgoing edges point to the same sub tree by that sub tree, since setting that node (variable) to *true* or *false* is irrelevant.

Example 3.1. Fig. 3 shows the BDD for the formula $\phi \equiv p \implies s \wedge \neg r$ with alphabetic ordering, whereas Fig. 4 shows a BDD expressing the same formula, but in a reduced fashion.

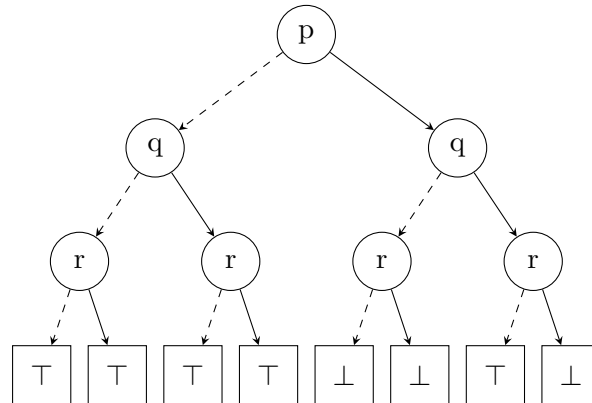


Figure 3: Ordered, but not reduced, BDD of $p \implies s \wedge \neg r$.

3 Binary Decision Diagrams

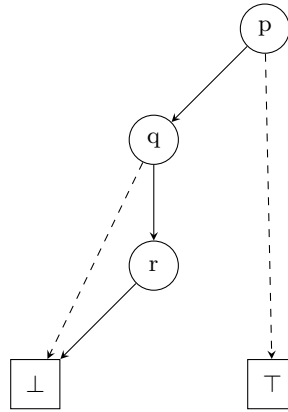


Figure 4: Ordered and reduced BDD of $p \implies s \wedge \neg r$.

3.2 Implementation

John Harrison’s implementation of Binary decision diagrams is quite pragmatic. Each node consists of a propositional variable and two integers, that determine the next node, where a negative number indicates complementation. The Tree itself is represented using 2 partial functions, one mapping an integer to a node, and one in the other direction. For convenience we also store the smallest unused integer as well as a function that defines an order over the propositional variables [2]. This method gives rise to the two definitions in Listing 4. In addition we have the two constant nodes with indices 1 and -1 representing \top and \perp respectively.

Listing 4: Representation of a BDD in OCaml.

```

1 type bddnode = prop * int * int;;
2
3 type bdd = Bdd of ((bddnode, int) func * (int, bddnode) func * int) *
4                 (prop->prop->bool);;
```

Using the adequacy property of binary functions, incorporating all binary connectives, requires us to only implement the *and* connective, since negation, \top , and \perp are already implicit. Using that we can construct all other connectives. Listing 5 shows the construction of a BDD expressing a conjunction.

Listing 5: Function constructing a conjunction between nodes m1 and m2.

```

1 let rec bdd_and (bdd, comp as bddcomp) (m1, m2) =
2   if m1 = -1 or m2 = -1 then bddcomp, -1
3   else if m1 = 1 then bddcomp, m2 else if m2 = 1 then bddcomp, m1 else
4   try bddcomp, apply comp (m1, m2) with Failure _ ->
5   try bddcomp, apply comp (m2, m1) with Failure _ ->
6   let (p1, l1, r1) = expand_node bdd m1
7   and (p2, l2, r2) = expand_node bdd m2 in
8   let (p, lpair, rpair) =
9     if p1 = p2 then p1, (l1, l2), (r1, r2)
10    else if order bdd p1 p2 then p1, (l1, m2), (r1, m2)
11    else p2, (m1, l2), (m1, r2) in
12   let (bdd', comp') = (lnew, rnew) =
```

```

13   thread bddcomp (fun s z -> s, z) (bdd_and, lpair) (bdd_and, rpair) in
14   let bdd', n = mk_node bdd' (p, lnew, rnew) in
15   (bdd'', ((m1, m2) |-> n) comp'), n;;

```

The function displayed in Listing 5 first checks for the trivial cases 1 and -1 , if the check fails, it continues by looking, if an equivalent node is already present, using the `expand_node` function. Should that not be the case the function proceeds to create and add the new node to the existing BDD. As previously mentioned, we utilize the fact, that constants and negation are implicit, and in addition to the representation of the *and* connective shown in Listing 5, we have formed a complete set of functions. Now we are able to build a BDD representing full fledged *propositional logic*, with all its connectives as shown in Listing 6.

Listing 6: Make BDD

```

1 let rec mkbdd (bdd, comp as bddcomp) fm =
2   match fm with
3   | False -> bddcomp, -1
4   | True -> bddcomp, 1
5   | Atom(s) -> let bdd', n = mk_node bdd (s, 1, -1) in (bdd', comp), n
6   | Not(p) -> let bddcomp', n = mkbdd bddcomp p in bddcomp', -n
7   | And(p, q) -> thread bddcomp bdd_and (mkbdd, p) (mkbdd, q)
8   | Or(p, q) -> thread bddcomp bdd_or (mkbdd, p) (mkbdd, q)
9   | Imp(p, q) -> thread bddcomp bdd_imp (mkbdd, p) (mkbdd, q)
10  | Iff(p, q) -> thread bddcomp bdd_iff (mkbdd, p) (mkbdd, q);;

```

4 Compactness

Compactness defines an important property about propositional logic and first order logic. The Theorem concerning propositional logic is stated in Corollary 4.1.

Theorem 4.1 (Compactness [2]). *For any set Γ of propositional formulas⁵, if each finite subset $\Delta \subseteq \Gamma$ is satisfiable, then Γ itself is satisfiable.*

An application for the compactness theorem is in the vicinity of the four colour theorem. It is possible to encode 4-colourability using the sets $C = \{1, 2, 3, 4\}$ and $\{p_v^i \mid v \in V \wedge i \in C\}$ [2]. Meaning that if vertex v was assigned colour i p_v^i evaluate to true and C simply being the available colours. Having defined the atoms, we can now apply three constraints on them:

- $\forall v \in V \mid \bigwedge_{i \in C} p_v^i$: This simply expresses the existence of a colour.
- $\forall v \in V \mid \{\neg(p_v^1 \wedge p_v^2) \wedge \neg(p_v^1 \wedge p_v^3) \wedge \neg(p_v^1 \wedge p_v^4) \wedge \neg(p_v^2 \wedge p_v^3) \wedge \neg(p_v^2 \wedge p_v^4) \vee \neg(p_v^3 \wedge p_v^4)\}$: Every vertex has exactly one colour.
- $\forall E(a, b) \mid \{\neg(p_a^1 \wedge p_b^1) \wedge \neg(p_a^2 \wedge p_b^2) \wedge \neg(p_a^3 \wedge p_b^3) \wedge \neg(p_a^4 \wedge p_b^4)\}$: No adjacent vertices can have the same colour.

⁵A set Γ of formulas is said to be satisfiable, if $\forall i \in \Gamma$, i is satisfiable.

5 Discussion

A planar graph, (V, E) , is 4-colourable if and only if the corresponding formula ϕ , incorporating the constraints from Section 4 holds. Considering an infinite graph G and its infinite 4-colour formula Γ , we can now consider every finite subset Δ of it. The finiteness of every Δ implies, that the variables $p_v^i \in \Delta$ and therefore all v considered in Δ are finite and planar. Using the 4-colour theorem and applying the *compactness theorem* we have proven this infinite graph G to be 4-colourable [2].

5 Discussion

We discussed various techniques of reasoning about propositional logic formulas. We first showed multiple traditional SAT solving algorithms, chronologically starting with the *DP procedure*. After having a look at *DPLL* and its improvements, we took a more refined look at the lesser known algorithm that is *Stålmarck's method*, first by explaining the theory, followed by a detailed example.

After that we briefly introduced *Binary Decision Diagrams*, Looking at both theory and the implementation that John Harrison presents in his book.

Finally, we addressed the *compactness theorem* for propositional logic. We then showed the applicability of the *compactness theorem* by applying it to the *4-colour theorem*. From the original *4-colour theorem* together with the *compactness theorem*, we concluded that the 4-colourability also holds for infinite planar graphs.

References

- [1] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug 1986.
- [2] J. Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, New York, NY, USA, 1st edition, 2009.
- [3] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge University Press, New York, NY, USA, 2004.
- [4] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, 53(6):937–977, Nov. 2006.
- [5] R. Paturi, P. Pudlák, M. E. Saks, and F. Zane. An improved exponential-time algorithm for k-SAT. *J. ACM*, 52(3):337–364, May 2005.
- [6] Scott Aaronson. $P=?NP$, chapter 1, pages 1–122. Springer, 2016. Content of Book "Open Problems in Mathematics".
- [7] M. Sheeran and G. Stålmarck. A tutorial on Stålmarck’s proof procedure for propositional logic. *Formal Methods in System Design*, 16(1):23–58, 2000.