Herwig Stütz, BSc

Algorithms for MAX-SAT

MASTERARBEIT

zur Erlangung des akademischen Grades
Diplom-Ingenieur

Masterstudium Mathematische Computerwissenschaften

Technische Universität Graz

Betreuerin:
Ao. Univ.-Prof. Dipl.-Ing. Dr.techn. Bettina Klinz

Institut für Diskrete Mathematik

Graz, März 2018

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am ....

(Unterschrift)

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

....

date (signature)

ABSTRACT

MAX-SAT is the optimization variant of the satisfiability (SAT) decision problem. The task for MAX-SAT is to find the maximum number of satisfiable clauses for a given boolean formula. Both problems are NP-hard.
The aim of this thesis is to present and implement a selection of algorithms for MAX-SAT. A computational study is conducted in which the implemented algorithms are compared with regard to their performances on test data. Johnson's greedy algorithm, a randomized variant from Poloczek and Schnitger, the metaheuristic simulated annealing, and the exact algorithm MSU3 are discussed.
Benchmark instances from two different MAX-SAT evaluation events are used to test the algorithms. From these experiments, the respective performance ratios and solver runtimes are collected. Furthermore, for the test set of partial instances, the approximation ratios are considered for those instances where reference values are available.
The results show good average performance ratios that are better than expected from the respective performance guarantees. Whenever simulated annealing was able to find a solution before the given timeout occurred, it was able to yield the best results but the greedy algorithms had a clear runtime advantage. The theoretically better approximation ratio of the randomized variant over the basic greedy algorithm could not be confirmed in practice. Finally, the results for partial instances suggest that the evaluated algorithms constitute suitable heuristics for certain instances.

ZUSAMMENFASSUNG

MAX-SAT ist die Optimierungsvariante des Entscheidungsproblems *Satisfiability* (SAT). Die Problemstellung bei MAX-SAT lautet die maximale Anzahl an erfüllbaren Klausen für die gegebene logische Formel zu finden. Beide Probleme sind NP-schwer.
Das Ziel dieser Masterarbeit ist es eine Auswahl von Algorithmen für MAX-SAT vorzustellen und zu implementieren. Eine experimentelle Untersuchung wird durchgeführt, in der die implementierten Algorithmen untereinander in Bezug auf deren Güte auf Testdaten verglichen werden. Johnsons Greedy-Algorithmus, eine randomisierte Variante von Poloczek und Schnitger, die Metaheuristik Simulated Annealing und der exakte Algorithmus MSU3 werden vorgestellt.
Benchmark Instanzen von zwei MAX-SAT Evaluation Bewerben werden verwendet um die Algorithmen zu testen. In diesen Experimenten werden die jeweiligen Güten und Laufzeiten der implementierten Lösungsroutinen erfasst. Weiters werden für Testdaten mit partiellen Instanzen die Approximationsgüten betrachtet, da bei diesen Testdaten für manche Referenzlösungen verfügbar sind.
Die Ergebnisse zeigen gute durchschnittliche Approximationsgüten, welche besser als die jeweiligen theoretischen Garantien sind. Wann immer Simulated Annealing ein Resultat vor dem Zeitlimit geliefert hat, war dieses das Beste unter den getesteten Algorithmen. Die Greedy-Algorithmen haben jedoch einen deutlichen Laufzeit-Vorteil. Die theoretisch besseren Approximationsgüten der randomisierten Variante gegenüber dem originalen Greedy-Algorithmus konnten in der Praxis nicht bestätigt werden. Schließlich zeigen die Ergebnisse für die partiellen Instanzen, dass die evaluierten Algorithmen sinnvolle Heuristiken für eine Klasse dieser Instanzen darstellen.

ACKNOWLEDGEMENT

First of all, I would like to thank my supervising professor, Dr. Bettina Klinz, for providing me with guidance in the choice of the topic and throughout the thesis.

A big thank you goes to my superiors at work, who supported me by granting my leave of absence to write this thesis, and my colleagues for taking over my tasks in the meantime.

I want to thank my wife, Gudrun, for her invaluable emotional support and linguistic advice.

Last, but not least, I would like to thank my mother, who always supported me and encouraged me to follow my interests.

Contents

1 Introduction

This thesis deals with the optimization version of the well-known *Satisfiability problem (SAT)* which can be stated as follows: The input consists of a boolean formula φ with m clauses in n variables given in conjunctive normal form (CNF). Each clause contains one or more variables in positive or negated form, called literals. The task is to find a truth assignment of the variables which satisfies the formula φ. The well known *Cook-Levin theorem* [14] proves that this problem is NP-complete, making it infeasible to solve for large instances provided that P $\neq$ NP. Note that 2-SAT, the class of instances with at most 2 literals per clause, can be decided in polynomial time, whereas 3-SAT is already NP-complete.

A natural variant of SAT is its optimization version, where the question is not only to answer whether the formula is satisfiable but also to find the maximum number of clauses that can be satisfied simultaneously in case it is not. This problem is called MAX-SAT and it is the main subject of this thesis.

MAX-SAT has applications in various fields. A few examples are VLSI debugging [12], protein interaction interference [50], and cancer therapy design [34].

MAX-SAT turns out to be even harder than SAT. Garey, Johnson, and Stockmeyer [21] proved that MAX-2-SAT, the subclass of MAX-SAT with at most 2 literals per clause, is NP-hard. They did so by showing that 3-SAT can be reduced to MAX-2-SAT.

This motivates the use of approximation algorithms and heuristics for MAX-SAT in order to find a solution in acceptable time, even if it suboptimal.

Approximation algorithms typically are applied in situations where problems are too hard to be solved with exact optimization algorithms. Many different approaches for the design of approximation algorithms can be taken. The basic ideas are often quite simple but are gradually refined and tuned. It is also possible to attempt to apply methods for related problems and evaluate their suitability.

Theoretical performance guarantees often only consider the worst-case, which is not reflecting the performance for an average instance or an instance from practice. Thus, evaluating the theoretical methods with diverse benchmark instances, especially including such with real-world applications, can help to identify possible gaps between theory and practice.

This knowledge can then be used to identify approximation algorithms suitable for the respective practical task. As so often, this is usually a tradeoff between quality and time.

The goal of this thesis is to investigate the gaps between theoretical and

practical results for a selection of methods and test sets for MAX-SAT.

1.1 Thesis outline

In Section 2, we provide the notions and preliminaries needed in the rest of the thesis.

A short overview of previous work on approximation algorithms for MAX-SAT is given in Section 3.

Section 4 discusses a selection of algorithms for the MAX-SAT problem which we implemented in order to compare their performance on certain test sets. We first present the greedy approximation algorithms of Johnson, followed by a randomized variant from Poloczek and Schnitger. Then, the metaheuristic of simulated annealing in its general form, as well as applied to MAX-SAT, is introduced. Finally, to give a contrast to these approximation algorithms and heuristics, a third-party exact solver, Open-WBO, is presented along with the underlying MSU3 algorithm.

Section 5 reports on computational experiments with the implemented algorithms. We first comment on some practical considerations about simulated annealing and the greedy algorithms with respect to running the benchmarks. Then, the test environment in which the benchmarks are run is introduced. This includes the command line tool in which the algorithms are implemented, the format of the benchmarks as well as the machine specifications and the deployment of the solver on the machine. Finally, the benchmark instances are listed, followed by a discussion of the experimental results.

2 Preliminaries

We start by defining the basic notions involved in the problem description of MAX-SAT. Building on these, we can proceed to the problem statement of MAX-SAT and its variants. We then define the *approximation ratio* in order to talk about the theoretical quality of the presented algorithms.

Definition 2.1 (SAT formula)**.** Let V be a set of boolean variables taking values in $\{0, 1\}$, also written as **true** and **false**. A *literal* y can either be a variable x in V or its boolean negation $\overline{x}$.

Then, a *SAT formula* φ (in the n variables in V) consists of the conjunction of m *clauses*, i.e.

$$\varphi = \bigwedge_i c_i.$$

Therein, each clause c_i is composed of the disjunction of at least one literal:

$$c_i = \bigvee_j y_{ij}.$$

A clause with exactly one literal is called *unit clause*.

A clause c is satisfied by the assignment $\mathbf{x}$, if one of its literals y is satisfied by the assignment. If the literal y is the variable x, then y is satisfied if and only if x is assigned **true** in $\mathbf{x}$ and if y is the negation of a variable, $\overline{x}$, then y is satisfied if and only if x is assigned **false**.

Note that the empty formula with 0 clauses is trivially satisfied by convention.

In this thesis, the set of clauses of a formula φ is written as S and, for a given assignment of the variables, the subset of satisfied clauses is denoted by S_{SAT}.

Definition 2.2 (MAX-SAT)**.**

- Given: A SAT formula φ in the variables in V.
- Goal: Find a truth assignment $\mathbf{x}$ of the variables in V that maximizes the number of satisfied clauses $|S_{\text{SAT}}|$.

2.1 MAX-SAT variants

Derived from the original definition for MAX-SAT, several variations of the problem description are possible:

One variant is to equip the clauses with weights. To each clause c we assign a non-negative weight w_c. The objective changes to maximizing the weight of the satisfied clauses $\sum_{c \in S_{\text{SAT}}} w_c$, instead of their number. This is called *weighted* MAX-SAT. By setting $w_c = 1$ for all clauses, an unweighted instance can easily be transformed into an instances of the *weighted* variant.

Another variant is to restrict the number of literals per clause. This idea gives rise to two related problems: In MAX-k-SAT, each clause can have at most k literals, whereas in MAX-Ek-SAT each clause must contain exactly k literals.

Finally, we can consider instances where a subset of clauses has to be fully satisfied in order for a solution to be feasible. Clauses in this subset are called *hard* clauses, while the other clauses are called *soft*. Instances of this kind are also called *partial*.

In the following, we are mainly interested in the general case, but some of these variants also appear.

2.2 Approximation algorithms

Since we are particularly interested in MAX-SAT, the definitions given here only consider maximization problems. However, corresponding definitions can be made for minimization problems.

We start by giving the definition of an optimal solution for a given problem:

Definition 2.3 (Feasible and optimal solutions)**.** For a given input x, the set of *feasible solutions* is denoted by $S(x) \neq \emptyset$. Each solution $s \in S(x)$ is assigned an *objective value* $v(x, s) > 0$. For a maximization problem, an *optimal solution* for a given input x is a feasible solution s^* with

$$v(x, s^*) = \max_{s \in S(x)} v(x, s).$$

The objective value of an optimal solution s^* for input x is denoted by $v^*(x) = v(x, s^*)$.

In order to judge the solution quality of an algorithm that might produce suboptimal solutions, we need a measure of the produced solution quality. In the following we will provide such a measure. First, we consider algorithms in general.

Definition 2.4. An *algorithm* A for an optimization problem is an algorithm that, for each problem instance x, finds feasible solution $s_A(x) \in S(x)$. The *length* of a given input x, denoted by $|x|$, is defined as the length of the string that encodes the input to the algorithm A.

Note that this definition does not yet define *approximation algorithms*. In order to define approximation algorithms, we first need the notion of *approximation ratio*, a quality measure for optimization algorithms.

The definitions of the approximation ratios start by assessing the quality of a single solution and progress to make statements about the algorithm in general.

Definition 2.5 (Approximation ratio of a solution)**.** The *approximation ratio* of a feasible solution $s \in S(x)$ compares its objective value to that of the optimal solution by forming the ratio

$$r(x, s) = \frac{v(x, s)}{v^*(x)}.$$

The definition for the approximation ratio of a solution can be extended in a straight-forward way to a specific algorithm.

Definition 2.6 (Approximation ratio of an algorithm)**.** The *approximation ratio* of the algorithm A for the instance x is defined as

$$r_A(x) = r(x, s_A(x)).$$

Note that there are other possible ways to define these ratios. For example, one of the fundamental books on the topic, Garey and Johnson [20], defines the approximation ratio in a way that unifies maximization and minimization problems, which comes in handy when considering optimization problems in general.

In order to be able to use the approximation ratio without a specific instance in mind, we consider the worst-case possible. Let n be a positive integer, then

$$r_A(n) = \min\{r_A(x) \colon |x| \leq n\}$$

where $|x|$ is the size of the instance x in some encoding suitable for the algorithm.

Definition 2.7 (Absolute approximation ratio)**.** The *absolute approximation ratio* for an algorithm A is defined as

$$r_A = \sup\{r \leq 1 \colon r_A(x) \geq r \quad \text{for all instances } x\}.$$

However, depending on the actual problem, this ratio does not always tell the full truth, in particular for large instances since the ratio can include a term that is shrinking as the instance size is increasing. In such situations, it

can be more useful to consider an asymptotic version of the approximation ratio.

Definition 2.8 (Asymptotic approximation ratio)**.** The *asymptotic approximation ratio* for the algorithm A is defined as

$$r_A^\infty = \sup\{r \leq 1 \colon \forall \varepsilon > 0 \colon \exists r(\varepsilon) > 0 \colon \forall x \text{ with } v^*(x) \geq r(\varepsilon) \colon r_A(x) \geq r - \varepsilon\}.$$

One example where the asymptotic ratio is more useful than the absolute one is the *first fit decreasing* (FFD) algorithm for BIN-PACKING. Here, the absolute approximation ratio is $r_{\text{FFD}} \leq \frac{2}{3}$, whereas the asymptotic ratio is $r_{\text{FFD}}^\infty = \frac{9}{11}$. See for example [20] for the details.

With the definitions of approximation ratios established, we can formulate approximation algorithms.

Definition 2.9 (Approximation algorithm)**.** An *approximation algorithm* A for an optimization problem is an algorithm that

- for each input x yields a feasible solution $s_A(x) \in S(x)$,
- runs in polynomial time in $|x|$, and
- for which there exists a constant $r \leq 1$ that bounds the approximation ratio of the algorithm, i.e. $r_A \geq r$.

Definition 2.10 (APX)**.** The complexity class APX consists of all problems for which there exists a polynomial-time approximation algorithm with an absolute approximation ratio of r for some constant $r \leq 1$.

APX is the class of all problems that are approximable up to a constant ratio.

3 Known approximation results for MAX-SAT

This section gives a quick summary of the evolution of approximation algorithms for MAX-SAT problems.

The first approximation algorithm for MAX-SAT is due to Johnson [30], who in 1974 provided a $\frac{1}{2}$-approximation algorithm. More specifically, he proved that the greedy algorithm guarantees a $(1 - \frac{1}{2^k})$-approximation if each clause contains at least k literals. The $\frac{1}{2}$ ratio was later improved by Chen, Friesen, and Zheng [11], who showed that Johnson's algorithm is actually a $\frac{2}{3}$-approximation algorithm and that this bound is tight.

In [49], Yannakakis presented an algorithm which first transforms a MAX-SAT instance to an instance that is equivalent in terms of approximability but which contains no unit clauses, that is, clauses with only one literal. Yannakakis' algorithm is based on non-trivial network flow techniques and leads to an approximation ratio of $\frac{3}{4}$ for MAX-SAT.

By using the probabilistic method on a solution to a linear programming relaxation in combination with Johnson's algorithm, Goemans and Williamson [25] found a simpler $\frac{3}{4}$-approximation solution to MAX-SAT.

With the use of semidefinite programming techniques (SDP), Goemans and Williamson were able to achieve a 0.87856-approximation ratio for MAX-2-SAT and a 0.7584-approximation for MAX-SAT [24]. The algorithms are extensions of SDP algorithms for MAX-CUT, which are also presented in the same paper.

Based on [18] and [24], Feige and Goemans [17] improved the algorithm for MAX-2-SAT to a 0.931-approximation via an extension of the formulation of MAX-SAT, that is making use of an extended SDP formulation, which includes additional constraints (triangle inequalities).

Building on the work of Goemans and Williamson and combining it with [17] and [31], Asano and Williamson [5] obtained a 0.7846 performance guarantee for MAX-SAT. Asano [4] later simplified and improved the analysis from [5] and obtained a performance guarantee of 0.7877.

To date, the best result for MAX-SAT was achieved by Avidor, Berkovitch, and Zwick [6]. It improves upon [4] by using a hybrid algorithm, and has an approximation ratio of 0.7968.

Regarding hardness, Håstad [27] showed that approximating instances containing exactly 3 literals (i.e. MAX-E3-SAT instances) with an approximation ratio of $\frac{7}{8} + \varepsilon$ is NP-hard for $\varepsilon > 0$.

Recently, work has been done on a special type of $\frac{3}{4}$-approximation algorithm: Given an initial ordering of the variables, the variables are successively

set to true or false with certain probabilities that depend on the built-up set of satisfied and unsatisfied clauses. Poloczek, Williamson, and Zuylen [43] give an overview of such algorithms and present a data structure for implementing them in linear time and space. One such example of a $\frac{3}{4}$-approximation algorithm is given in Buchbinder et al. [9] as a special case of their work on submodular maximization.

An overview of approximation algorithms with respect to performance on real world MAX-SAT instances is given by Poloczek and Williamson [42].

4 A selection of algorithms for MAX-SAT

This section provides a detailed description of those MAX-SAT algorithms that are part of the computational study which is reported on in Section 5.

First, Johnson's greedy approximation algorithm and the Slack algorithm, a variant of Johnson's algorithm, are described. For both of them, first an overview is given, followed by the detailed pseudocode and an analysis of their respective approximation guarantees. We then describe the data structures that are used for the implementation.

Then, the metaheuristic simulated annealing is presented. Starting with the conceptual origins, the method is motivated, finally leading to the general form of the algorithm. Furthermore, we state some results on the theoretical convergence of the algorithm with references to more details. The section is closed by applying simulated annealing to MAX-SAT, giving specific pseudocode for the algorithm together with notes for its implementation.

Finally, the third-party exact solver Open-WBO is described. This contrasts the previously discussed approximation algorithms and heuristic. The solver is described from a practical side first before giving a brief overview of the underlying algorithm.

4.1 Johnson's Greedy Algorithm

Greedy algorithms are widely used in the field of optimization, due to their inherent simplicity. They work by incrementally fixing parts of the solution, only considering local information for the respective decision. For MAX-SAT, the parts that are fixed are the truth assignments of the variables $x \in V$. In each step, a variable is fixed to either 0 or 1 based on a metric for either the number of newly satisfied clauses or the weight of the clauses containing the variable.

We start with the greedy algorithm that is referred to as *B2* in [30]. Johnson uses a set-representation for the instances. With this set-representation, the greedy algorithm can be described as follows: The SAT formula φ is represented as the set of its clauses, denoted by S. Each clause c is itself represented as the set of its literals y and is assigned a clause weight w_c.

During the computation, the algorithm keeps track of the set of satisfied clauses S_{SAT} and the set of clauses S_{LEFT} which are left to be considered. Additionally, the set of unfixed literals is denoted by L_{LEFT} and the set of literals fixed to **true** is L_{SAT}.

All clauses that are not yet satisfied but still contain at least one unfixed literal are called *alive*. Clauses that are alive but contain literals that were set to **false** are called *wounded*.

Initially, all clauses are assigned some weight depending on the number of literals in the respective clause and the original clause weight in the formula. Then, iterating over the variables in an arbitrary but fixed order, the algorithm chooses a truth value for each variable such that it maximizes the (modified) weight of the newly satisfied clauses. Clauses that contain the negated literal, and thus were wounded in this step, are modified by doubling their weights. This is supposed to mitigate the reduced probability of such a clause being satisfied.

The complete pseudocode can be found in the listing for Algorithm 1. It closely resembles the original structure of the algorithm in [30], with only minor changes to the names of the involved sets and the straight-forward extension to weighted clauses.

Algorithm 1 Greedy algorithm

procedure GREEDY(S) ▷ Maximizing the weight of sat. clauses in S
  for all $c \in S$ **do** ▷ Set initial weights
    $w(c) \leftarrow w_c 2^{-|c|}$
  end for
  $S_{\text{SAT}} \leftarrow \emptyset, L_{\text{SAT}} \leftarrow \emptyset, S_{\text{LEFT}} \leftarrow S, L_{\text{LEFT}} \leftarrow$ literals in S
  while $L_{\text{LEFT}} \cap S_{\text{LEFT}} \neq \emptyset$ **do** ▷ Loop over literals
    Let $y \in L_{\text{LEFT}}$ such that y occurs in S_{LEFT}
    $S_y \leftarrow$ clauses in S_{LEFT} containing y
    $S_{\overline{y}} \leftarrow$ clauses in S_{LEFT} containing $\overline{y}$
    if $\sum_{c \in S_y} w(c) \geq \sum_{c \in S_{\overline{y}}} w(c)$ **then** ▷ Choose "heavier" literal
      $L_{\text{SAT}} \leftarrow L_{\text{SAT}} \cup \{y\}, S_{\text{SAT}} \leftarrow S_{\text{SAT}} \cup S_y, S_{\text{LEFT}} \leftarrow S_{\text{LEFT}} \setminus S_y$
      for all $c \in S_{\overline{y}}$ **do**
        $w(c) \leftarrow 2w(c)$
      end for
    else
      $L_{\text{SAT}} \leftarrow L_{\text{SAT}} \cup \{\overline{y}\}, S_{\text{SAT}} \leftarrow S_{\text{SAT}} \cup S_{\overline{y}}, S_{\text{LEFT}} \leftarrow S_{\text{LEFT}} \setminus S_{\overline{y}}$
      for all $c \in S_y$ **do**
        $w(c) \leftarrow 2w(c)$
      end for
    end if
    $L_{\text{LEFT}} \leftarrow L_{\text{LEFT}} \setminus \{y, \overline{y}\}$
  end while
  return $L_{\text{SAT}}, S_{\text{SAT}}$
end procedure

Theorem 4.1 (Johnson)**.** *For MAX-SAT instances with at least $k \geq 1$ literals per clause,* Algorithm 1 *has an approximation ratio of $r_1(n) \geq \frac{2^k-1}{2^k}$.*

For simplicity, the proof is given for the unweighted case. However, the weighted case can be handled similarly by using the total weight $W = \sum_{c \in S} w_c$ for the clauses in S instead of just the respective number of clauses, $|S|$.

Proof. In the beginning, since each clause has at least k literals, the summed weight of the clauses in S_{LEFT} is at most $|S| \cdot 2^{-k}$. Now, in each step of the iteration, a literal y is chosen and the set of clauses containing y is removed from S_{LEFT}. By the greedy property, the weight that is subtracted by removing the clauses from S_{LEFT} is at least the weight that is added to S_{LEFT} by doubling the weight of the wounded ones. Thus, the weight of S_{LEFT} does not increase and is still at most $|S| \cdot 2^{-k}$ at the end of the algorithm. On the other hand, at the end, each clause in S_{LEFT} must have been wounded, and thus doubled in weight, exactly as many times as it contains literals. This means that, after the procedure, each clause in S_{LEFT} has weight 1. Combining these arguments yields

$$|S| = |S_{\text{LEFT}}| + |S_{\text{SAT}}| \leq \frac{|S|}{2^k} + |S_{\text{SAT}}|,$$

and thus

$$|S_{\text{SAT}}| \geq |S|\big(1 - \frac{1}{2^k}\big). \qquad \square$$

This proof does not take into account the optimal solution, but compares the number of satisfied clauses in the solution to the total number of clauses. This measure is called *performance ratio* and is a lower bound for the approximation ratio.

Note that for instances containing unit clauses, this analysis of the algorithm can only promise a performance ratio of $\frac{1}{2}$. However, this bound is not tight. Both [11] and the full version of [40] prove that Johnson's algorithm is indeed a $\frac{2}{3}$-approximation algorithm. The former is using Johnson's original notation while the latter is using the terminology devised for the Slack algorithm in [41].

For an example where the bound from Theorem 4.1 is tight, consider the formula

$$\begin{aligned}\varphi = {} & (\overline{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_4 \vee x_5) \\ & \wedge (\overline{x}_1 \vee x_2 \vee \overline{x}_3) \wedge (x_1 \vee x_6 \vee x_7) \\ & \wedge (\overline{x}_1 \vee \overline{x}_2 \vee x_3) \wedge (x_1 \vee x_8 \vee x_9) \\ & \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3) \wedge (x_1 \vee x_{10} \vee x_{11}).\end{aligned}$$

It can be fully satisfied by any assignment with $x_1 = 0$, $x_4 = 1$, $x_6 = 1$, $x_8 = 1$ and $x_{10} = 1$. But for the first decision of the algorithm, namely $\sum_{c \in S_{x_1}} w(c) \geq \sum_{c \in S_{\overline{x}_1}} w(c)$, equality holds and the algorithm may choose $x_1 = 1$. Then every possible assignment of the variables x_2 and x_3 will leave one clause unsatisfied. So the approximation ratio for such a solution is $r_1(\varphi) = \frac{7}{8} = \frac{2^k-1}{2^k}$.

Johnson's paper states a time complexity of $\mathcal{O}(N \log N)$ with $N = |\varphi|$ being the overall (encoding) length of the formula.

The algorithm can be implemented quite easily by following the structure of the original description of the algorithm closely using a library implementing the used set operations, such as the `Set` library from [13]. A detailed analysis of the complexity is omitted at this point in favor of the analysis of the simplified variant in Section 4.1.2.

4.1.1 A Simplified Variant

With the naive implementation of the algorithm, which simply translates into calls to the `Set` library, the runtime of the program is not even practical for small instances.

To overcome this, we use a more direct approach: Using a more variable-centric notation, we can re-write the algorithm such that an $\mathcal{O}(N)$ time implementation is possible.

Again, let us consider the update procedure of Johnson's algorithm: With the defined start weight for each clause and the update rule to double the weight of all clauses that were wounded, we get

$$\mu_x = \sum_{x \in c} w_c 2^{-|c|}$$

as the metric for variable x, where $|c|$ is the number of unfixed variables of the clause.

Essentially, the algorithm then boils down to iterating over all variables x and deciding on the respective truth assignment solely based on μ_x and $\mu_{\overline{x}}$. The pseudocode for this condensed form can be found in listing Algorithm 2.

What is left to do is to define a suitable data structure which allows the involved operations to be done in an efficient way. The main operation in the algorithm is the summing of clause weights for a given variable. For this, it would be helpful to have a mapping from the variable to the required data for μ_x and $\mu_{\overline{x}}$.

Therefore, for each clause we store its respective weight and current length in a vector. Then, for each variable, a list of references to the clauses that contain the positive literal and a list of references to the clauses that contain

Algorithm 2 Greedy algorithm

```
1: procedure GREEDY(S)                ▷ Maximizing the satisfied clauses in S
2:     for all x ∈ V do
3:         μ_x ← Σ_{c: x∈c, c alive} w_c 2^{-|c|},   μ_x̄ ← Σ_{c: x̄∈c, c alive} w_c 2^{-|c|}
4:         if μ_x ≥ μ_x̄ then
5:             x ← 1
6:         else
7:             x ← 0
8:         end if
9:         Remove x, x̄ from all clauses in S
10:    end for
11: end procedure
```

the negative literal can be kept. As references, the indices of the actual clauses in the array are used. The positive and negative lists are stored in two separate arrays (*pos* and *neg* in Figure 4.1.1) and are accessible from the variable, as their indices provide a mapping from variable to clauses with $\mathcal{O}(1)$ access time.

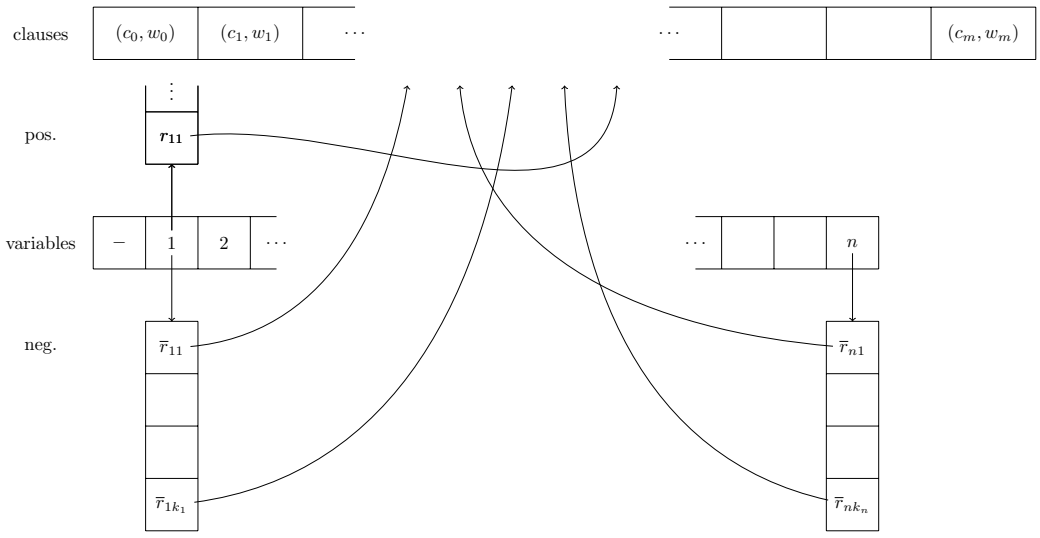

Figure 1: Custom data structure for the greedy algorithm

With this data structure, all that needs to be done in each step of the algorithm is to go through the *pos* and *neg* lists for the current variable, and retrieve the referenced clause data.

After a truth value has been assigned to the variable, the update works in a similar way: All clauses that are now satisfied by the variable setting can be marked as such by simply traversing through the corresponding list and updating the entry in the array containing the clause weights. Similarly, all clauses that were wounded in this step can be updated by iterating over the list of clauses and updating the respective weight or length.

4.1.2 Running time analysis

If N is the overall number of literals the formula φ contains, counting multiple occurrences of each literal, then by using this data structure a time and space complexity of $\mathcal{O}(N)$ can be achieved.

First, consider the construction of the data structure: The construction of the array storing the clause weights and remaining lengths needs $\mathcal{O}(N)$ time and $\mathcal{O}(m)$ space. All N literals are traversed once. For each distinct literal, a reference to the clause it is contained in is added to *pos* or *neg*, the clause lists of the literal. This can be done in $\mathcal{O}(1)$ for each literal. Thus the overall construction can be done in $\mathcal{O}(N)$ time.

During the computation, the steps needed for each of the n variables are:

1. Traverse through both *pos* and *neg* to retrieve the clauses containing x or $\overline{x}$ and calculate μ_x and $\mu_{\overline{x}}$. For each reference in *pos* and *neg*, $\mathcal{O}(1)$ time is needed to access the information from the clause array. Thus, if k_x and $k_{\overline{x}}$ are the lengths of *pos* and *neg*, this needs $\mathcal{O}(k_x + k_{\overline{x}})$ time.

2. Since μ_x and $\mu_{\overline{x}}$ are already calculated, the decision whether to set $x = 0$ for $x = 1$ only takes $\mathcal{O}(1)$.

3. Updating the clause lengths in the clause array for the satisfied and wounded clauses: As with the retrieval, this can be done in $\mathcal{O}(1)$ for each of the $k_x + k_{\overline{x}}$ references.

Summing over all n variables thus yields a runtime of

$$\sum_x \mathcal{O}(k_x + k_{\overline{x}}) = \sum_c \mathcal{O}(|c|) = \mathcal{O}(N).$$

4.2 Slack Algorithm

The Slack algorithm is a randomized variant of Johnson's greedy algorithm and was first proposed in [41] (2011). It builds upon the *canonical randomization (CR)* of a modified Johnson's algorithm. Instead of greedily choosing the

value of x based on μ_x and $\mu_{\overline{x}}$, the decision is made probabilistically with probabilities $q_0 = \frac{\mu_{\overline{x}}}{\mu_x + \mu_{\overline{x}}}$ for $x = 0$ and $q_1 = \frac{\mu_x}{\mu_x + \mu_{\overline{x}}}$ for $x = 1$.

The main difference to a step in Johnson's algorithm is the definition of the measure used to decide which value to assign to a variable. In the original algorithm, this measure was defined as

$$\mu_x = \sum_{c:\ x \in c} w_c 2^{-|c|}.$$

This gives shorter clauses larger weights than longer clauses, as they have a higher probability of becoming unsatisfiable.

The modification for the Slack algorithm is to still give unit clauses a larger weight but to leave the weights of longer clauses untouched:

$$\mu_x = 2 \sum_{\substack{c:\ x \in c \\ c \text{ alive} \\ |c|=1}} w_c + \sum_{\substack{c:\ x \in c \\ c \text{ alive} \\ |c|>1}} w_c, \quad \mu_{\overline{x}} = 2 \sum_{\substack{c:\ \overline{x} \in c \\ c \text{ alive} \\ |c|=1}} w_c + \sum_{\substack{c:\ \overline{x} \in c \\ c \text{ alive} \\ |c|>1}} w_c.$$

The paper in [41] defines additional variables in order to simplify this definition. Let the values w_x and $w_{\overline{x}}$ be the weights of the unit clause x and $\overline{x}$ respectively:

$$w_x = \sum_{\substack{c:\ x \in c \\ c \text{ alive} \\ |c|=1}} w_c, \quad w_{\overline{x}} = \sum_{\substack{c:\ \overline{x} \in c \\ c \text{ alive} \\ |c|=1}} w_c,$$

Also, let *fanin* be the combined weight of all alive non-unit clauses containing x (after removing the already fixed variables) and let *fanout* be the same for $\overline{x}$:

$$\text{fanin} = \sum_{\substack{c:\ x \in c \\ c \text{ alive} \\ |c|>1}} w_c, \quad \text{fanout} = \sum_{\substack{c:\ \overline{x} \in c \\ c \text{ alive} \\ |c|>1}} w_c.$$

With the additional definition of

$$\Delta = 2w_x + \text{fanin} + 2w_{\overline{x}} + \text{fanout},$$

the definitions from above can be rewritten as

$$\mu_x = \text{fanin} + 2w_x, \quad \mu_{\overline{x}} = \text{fanout} + 2w_{\overline{x}},$$

and

$$q_0 = \frac{\text{fanout} + 2w_{\overline{x}}}{\Delta}, \quad q_1 = \frac{\text{fanin} + 2w_x}{\Delta}.$$

Note that for formulas with 2 literals per clause the two definitions of μ_x,

the one from Johnson's algorithm and the one defined for the Slack algorithm, are equivalent.

The distance between the weights for $x = 0$ and $x = 1$ is called *slack* and is defined as

$$\text{slack} = |\mu_x - \mu_{\overline{x}}| = |q_1 - q_0| \cdot \Delta.$$

The basic idea for the algorithm is to carefully increase the slack by some ε by changing the probabilities to

$$p_0 = q_0 - \varepsilon, \quad p_1 = q_1 + \varepsilon$$

where we assume w.l.o.g. $q_0 \leq q_1$.

To find a suitable ε, and with it p_0 and p_1, we consider one step of the algorithm using the CR with the modified weights $\mu_{\overline{x}}$ and μ_x as defined for the Slack algorithm. Increasing the probability from q_1 to p_1 is a good idea if it increases the likelihood of the variable being fixed to the value that would also be assigned to the variable in an optimal assignment. However, if this is not the case, it constitutes a new source of error. We consider the decision for variable x with respect to the expected deviation, or "contradiction", to the optimal assignment π that is added by using the changed probabilities.

Let *sat* be the total weight of newly satisfied clauses and let *unsat* be the total weight of terminally unsatisfied clauses after fixing x. Both are random variables and their expected values are given by

$$\begin{aligned} \mathbb{E}[\text{sat}] &= p_0(w_{\overline{x}} + \text{fanout}) + p_1(w_x + \text{fanin}) \\ \mathbb{E}[\text{unsat}] &= p_0 w_x + p_1 w_{\overline{x}}. \end{aligned}$$

Furthermore, let *wounded* be the weight of clauses which are still alive after fixing x, but for which one of their literals was falsified. Again, this forms a random variable with expectation

$$\mathbb{E}[\text{wounded}] = p_0 \,\text{fanin} + p_1 \,\text{fanout}\,.$$

Now we can consider the "contradiction" when setting x as compared to the optimal assignment π. This can be expressed with the two random variables c and c'. They represent the total weight of all alive clauses right before and after the decision for x is made respectively. The assignment for the variables up to but excluding x is the same as in π, while for the decision for x the probabilities from the Slack algorithm are used. Then $c' - c$ is the "lost potential", or "contradiction", when fixing x. This contradiction can be related to the other previously defined random variables according to the following Lemma.

Lemma 4.1.

$$\mathbb{E}[sat - 3 \cdot unsat - 2(c' - c)] = \mathbb{E}[slack + wounded - 2(c' - c)] - (w_{\overline{x}} + w_x). \quad (1)$$

Proof. The definition of slack can be re-written as

$$\text{slack} = \begin{cases} \text{fanin} + 2w_x - (\text{fanout} + 2w_{\overline{x}}) & \text{if } x = 1 \\ \text{fanout} + 2w_{\overline{x}} - (\text{fanin} + 2w_x) & \text{if } x = 0 \end{cases}$$

by splitting up the absolute value into its cases and substituting μ_x and $\mu_{\overline{x}}$ for their definitions. The expected value of slack is

$$\begin{aligned} \mathbb{E}[\text{slack}] = {} & p_1(\text{fanin} + 2w_x - (\text{fanout} + 2w_{\overline{x}})) \\ & + p_0(\text{fanout} + 2w_{\overline{x}} - (\text{fanin} + 2w_x)). \end{aligned}$$

We now consider a simplification of Equation 1 and show that $\mathbb{E}[\text{sat} - 3 \cdot \text{unsat}] = \mathbb{E}[\text{slack} + \text{wounded}] - (w_{\overline{x}} + w_x)$:

$$\begin{aligned} & \mathbb{E}[\text{sat} - 3 \cdot \text{unsat}] \\ & \quad = p_0(w_{\overline{x}} + \text{fanout}) + p_1(w_x + \text{fanin}) - 3(p_0 w_x + p_1 w_{\overline{x}}) \\ & \quad = p_0(2w_{\overline{x}} + \text{fanout} - 2w_x - \text{fanin}) + p_1(2w_x + \text{fanin} - 2w_{\overline{x}} - \text{fanout}) \\ & \qquad - p_0 w_{\overline{x}} - p_0 w_x + p_0 \,\text{fanin} - p_1 w_x - p_1 w_{\overline{x}} + p_1 \,\text{fanout} \\ & \quad = \mathbb{E}[\text{slack}] - p_0(w_{\overline{x}} + w_x) - p_1(w_x + w_{\overline{x}}) + p_0 \,\text{fanin} + p_1 \,\text{fanout} \\ & \quad = \mathbb{E}[\text{slack}] - (w_{\overline{x}} + w_x) + \mathbb{E}[\text{wounded}] \end{aligned}$$

From this, the Lemma follows. □

Summing over all variables, we obtain the following result. Note that the yet to be determined new probabilities will make the right-hand side in Equation 1 non-negative.

Lemma 4.2.

$$\begin{aligned} \mathbb{E}[Sat] &\geq 3\mathbb{E}[Unsat] - 2(W - Opt) \\ &= W - 3\mathbb{E}[Sat] + 2\,Opt, \end{aligned}$$

where Sat *and* Unsat *are the sums of* sat *and* unsat*, respectively, over all variables in the SAT formula, W is the total weight of all clauses, and* Opt *is the objective value of an optimal solution.*

We now give the analysis for choosing ε and the modified probabilities by closely following the analysis in [41].

Lemma 4.3. *Let p_0 and p_1 be the assignment probabilities for $x = 0$ and $x = 1$ respectively. Then, if $x = 1$ is optimal, the equation*

$$\mathbb{E}[c' - c] \leq p_0 \cdot \text{fanin} - w_{\overline{x}}$$

holds and if $x = 0$ is optimal, then

$$\mathbb{E}[c' - c] \leq p_1 \cdot \text{fanout} - w_x$$

holds.

Proof. Assume that $x = 1$ is optimal. We consider the cases for fixing $x = 0$ and $x = 1$:

- Assume that the variable x is fixed to 0. Consider the alive clauses (after fixing x) that would have been satisfied by $x = 1$ but become contradictory when setting the variables as in the optimal assignment otherwise. Denote the weight of these clauses with fanin^c. Clearly, the inequality $\text{fanin}^c \leq \text{fanin}$ holds.
- Assume that the variable x is fixed to 1. No new contradiction is introduced, i.e. $\text{fanout}^c = 0$.

The contradictory unit clauses $\overline{x}$ are removed in both cases. Thus

$$\mathbb{E}[c' - c] = p_0(\text{fanin}^c - w_{\overline{x}}) - p_1 w_{\overline{x}} \leq p_0 \,\text{fanin} - w_{\overline{x}}$$

The case for $x = 0$ being optimal can be shown analogously. □

Lemma 4.4.

1. $q_1 \,\text{fanout} - q_0 \,\text{fanin} = 2q_0 w_x - 2q_1 w_{\overline{x}}$.
2. *Let* $\text{Slack} = |2w_x + \text{fanin} - (2w_{\overline{x}} + \text{fanout})|$, $p_0 = q_0 - \varepsilon$, $p_1 = q_1 + \varepsilon$ *and assume that* $q_0 \leq q_1$ *and* $x = 1$ *is optimal. Then*

$$\begin{aligned}&\mathbb{E}[\text{wounded} - 2(c' - c)] - (w_x + w_{\overline{x}})\\ &\geq -\frac{\text{Slack}}{\Delta}(w_x + w_{\overline{x}}) + \varepsilon(\text{fanout} + \text{fanin}).\end{aligned}$$

If $x = 0$ is optimal, then

$$\begin{aligned}&\mathbb{E}[\text{wounded} - 2(c' - c)] - (w_x + w_{\overline{x}})\\ &\geq \frac{\text{Slack}}{\Delta}(w_x + w_{\overline{x}}) - \varepsilon(\text{fanout} + \text{fanin}).\end{aligned}$$

Proof.

1. $q_1 \,\text{fanout} - q_0 \,\text{fanin} = 2q_0 w_x - 2q_1 w_{\overline{x}}$ is equivalent to $q_1(2w_{\overline{x}} + \text{fanout}) = q_0(\text{fanin} + 2w_x)$. In the last equation, both sides are equivalent to $q_0 q_1 \Delta$.

2. If $x = 1$ is optimal then

   $$\mathbb{E}[\text{wounded} - 2(c' - c)] - (w_x + w_{\overline{x}}) \geq p_1 \,\text{fanout} - p_0 \,\text{fanin} + w_{\overline{x}} - w_x$$

   holds by using Lemma 4.3 and the equation for the expected value of wounded. Then, using the first statement of this lemma, it can be shown that

   $$p_1 \,\text{fanout} - p_0 \,\text{fanin} + w_{\overline{x}} - w_x = -\frac{\text{Slack}}{\Delta}(w_x + w_{\overline{x}}) + \varepsilon(\text{fanout} + \text{fanin}).$$

   With this, the claim follows. □

For $q_0 \leq q_1$ and $p_0 = q_0 - \varepsilon$, $p_1 = q_1 + \varepsilon$ the expected slack can be re-formulated as

$$\mathbb{E}[\text{slack}] = \frac{Slack^2}{\Delta} + 2\varepsilon \,\text{Slack},$$

using $\text{Slack} = (q_1 - q_0)\Delta$. Then assuming $x = 1$ is optimal we get the inequality

$$\begin{aligned} &\mathbb{E}[\text{slack} + \text{wounded} - 2(c' - c)] - (w_{\overline{x}} + w_x) \\ &\quad \geq \frac{Slack^2}{\Delta} + 2\varepsilon \,\text{Slack} - \frac{\text{Slack}}{\Delta}(w_x + w_{\overline{x}}) + \varepsilon(\text{fanout} + \text{fanin}). \end{aligned} \tag{2}$$

If setting $\varepsilon = 0$, and thus considering the original probabilities, then the right hand side of Equation 2 is negative only for $\text{Slack} \in (0, w_x + w_{\overline{x}})$. We choose this condition to decide whether to adjust the probabilities in the algorithm. For the case where the probabilities need to be adjusted, i.e. $\varepsilon > 0$, a solution that makes the right hand side disappear is

$$\varepsilon_1 = \frac{-\frac{\text{Slack}^2}{\Delta} + \frac{\text{Slack}}{\Delta}(w_x + w_{\overline{x}})}{2\,\text{Slack} + \text{fanout} + \text{fanin}}.$$

A similar analysis can be done for the case where $x = 0$ is optimal. There, the right hand side of the inequality for $\mathbb{E}[\text{slack} + \text{wounded} - 2(c' - c)] - (w_{\overline{x}} + w_x)$ is non-negative for all $\varepsilon \geq 0$, provided $\text{fanout} + \text{fanin} \leq 2\,\text{Slack}$ holds. This yields a correction term ε_2 which is not lower than ε_1. Thus ε_1 can be used as the correction term for the case where $x = 1$ as well as where $x = 0$ holds.

It can be shown that the modified probabilities are well defined, i.e $0 \leq \varepsilon_1 \leq q_0$.

Utilizing these considerations, the Slack algorithm can be formulated as in the listing for Algorithm 3.

Algorithm 3 Slack algorithm

procedure SLACK(S) ▷ Maximizing the weight of sat. clauses in S
  for all $x \in V$ **do**
    Set $w_x, w_{\overline{x}}, \text{fanin}, \text{fanout}$ and Δ as defined above.
    $q_0 \leftarrow \frac{2w_{\overline{x}}+\text{fanout}}{\Delta}, \quad q_1 \leftarrow \frac{2w_x+\text{fanin}}{\Delta}$
    $\text{slack} \leftarrow |q_0 - q_1| \cdot \Delta$
    $\varepsilon_1 \leftarrow \frac{-\frac{\text{slack}^2}{\Delta}+\frac{\text{slack}}{\Delta}(w_x+w_{\overline{x}})}{2\,\text{slack}+\text{fanout}+\text{fanin}}$
    if slack $= 0$ or slack $\geq w_x + w_{\overline{x}}$ **then**
      $p_0 \leftarrow q_0, \quad p_1 \leftarrow q_1$
    else
      if $q_0 < q_1$ **then**
        $p_0 = q_0 - \varepsilon_1, \quad p_1 = q_1 + \varepsilon_1$
      else
        $p_0 = q_0 + \varepsilon_1, \quad p_1 = q_1 - \varepsilon_1$
      end if
    end if
    $x \leftarrow \begin{cases} 0 & \text{with prob. } p_0 \\ 1 & \text{with prob. } p_1 \end{cases}$
  end for
end procedure

Theorem 4.2 (Poloczek and Schnitger [41])**.** *Let W be the total weight of all clauses and let Opt be the total weight of all satisfied clauses in an optimal solution. The Slack algorithm satisfies clauses with an expected total clause weight of*

$$\mathbb{E}[Sat] \geq \frac{2\,Opt + W}{4} \geq \frac{3}{4}\,Opt.$$

This is tight: No approximation ratio $r > \frac{3}{4}$ can be achieved.

We will now illustrate the rationale for the tightness result. For this purpose we use an example which is particularly bad for the Slack algorithm: Let n be sufficiently large. We define $2n$ variables $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$. For the formula we consider the clauses $(\overline{x}_i \vee y_j) \wedge (\overline{y}_j \vee x_i)$, for all $i, j = 1, \ldots, n$ representing the logical equivalence $x_i \leftrightarrow y_j$.

This formula can be completely satisfied with an optimal value of $2n^2$ by assigning $x_i = 1$ and $y_j = 1$ for all i, j.

We now consider the assignment of the Slack algorithm when first processing all variables x_i and then all y_j variables. Since slack $= 0$ holds for all x_i, the probabilities are $\text{prob}[x_i = 1] = \text{prob}[x_i = 0] = \frac{1}{2}$. We therefore expect to only set half of the variables x_i to 1. No matter the assignment of the variables y_j, at least $\frac{1}{4}$ of the total weight will not be satisfied:

Suppose k of the variables x_i have been set to 0. Now consider y_j for some j: There are k terms of the form $(0 \to y_j) \wedge (y_j \to 0)$ and $n - k$ terms of the form $(1 \to y_j) \wedge (y_j \to 1)$. The first group demands $y_j = 0$ while the second group demands $y_j = 1$ in order for the formula to be fully satisfied. For an optimal assignment, given the variables x_i fixed above, the algorithm should choose $y_j = 0$ for $k \geq \frac{n}{2}$ and $y_j = 1$ for $n - k \geq \frac{n}{2}$. In any case, there will be $\min(k, n - k)$ unsatisfied clauses for each y_j. If $y_j = 1$ then k clauses of the from $y_j \to 0$ are unsatisfied and if $y_j = 0$ then $n - k$ clauses of the form $1 \to y_j$ are unsatisfied. Thus, the total number of unsatisfied clauses is nk out of $2n^2$. When choosing n large enough, k will be close to $\frac{n}{2}$ and the overall ratio is $\frac{1}{4}$.

4.2.1 Running time analysis

The Slack Algorithm can use the same data structure that was used above for Johnson's Algorithm. While the calculations needed in order to decide on the truth assignment for each variable are slightly more complex, this calculation is still possible in $\mathcal{O}(1)$ time. Thus, we again get a time and space complexity of $\mathcal{O}(N)$, where N is the length of the overall formula.

4.3 Simulated Annealing

In this section we describe simulated annealing, both as a general optimization method and as a heuristic for MAX-SAT.

A short overview of the history of the method is given first, followed by the method itself in its general form based on [15]. We then consider convergence results for the method. Finally, with the general terms established, simulated annealing as applied to MAX-SAT is considered. The algorithm as proposed in [45] is described, including an explanation of how the algorithm fits into the larger simulated annealing landscape.

4.3.1 Overview

Simulated annealing has its conceptual origins in thermodynamics. There, *annealing* describes the process of first heating a material to a high temperature and then slowly cooling the matter in order to bring it to a desired equilibrium state. At the initial high temperature, the molecules can still move around relatively freely to find a thermal equilibrium. But as the material is cooling down, the possible movements of the molecules become more and more restricted. Finally, the material solidifies into a robust state of minimal energy between the particles.

This physical process is translated into an algorithm for general optimization problems, where a given energy function is to be minimized. The result is a general metaheuristic for various optimization problems. It combines a descent algorithm with probabilistic uphill steps that allow to escape local minima.

First formulated in [32] and [10], we can also see the method as an iterated version of the Metropolis algorithm from [37] or the generalized version known as the Metropolis-Hastings algorithm from [28].

The Metropolis algorithm employs a modified Monte Carlo scheme to find an equilibrium state for a fixed temperature using the Boltzmann distribution, which was generalized to arbitrary distributions by Hastings.

Simulated annealing repeatedly applies the Metropolis-Hastings algorithm, starting with a high temperature that is gradually lowered after each iteration. In the beginning the high temperature allows for more steps away from the current local minimum, just like in the physical process, whereas such moves become increasingly unlikely as the temperature cools down. When the temperature reaches 0, only moves towards the (local) minimum are accepted.

At its core, simulated annealing employs two stochastic processes. The first is the generation of the states in the system: The initial state for the algorithm is generated randomly and in each step random perturbations are applied to the state in order to move around the state space in search of a global minimum. These perturbations are applied with certain acceptance probabilities P.

This is where the second stochastic process comes into play. The perturbed states that have lower energy than the originating state are assigned a probability of 1, whereas the probabilities for perturbed states with a higher energy (i.e. that are further away from the optimum) are set to a value that depends on the current temperature and the energy difference between the states.

For each temperature, the perturbation and acceptance processes are iterated until the current state has reached a certain equilibrium condition.

This termination condition can take several different forms: It ranges from limiting the number of accepted or unaccepted state transitions to fixing the number of iterations, e.g. considering each component of the state exactly once.

Starting with $T_{\max}$, the procedure described above is repeated for each temperature T until $T_{\min}$ is reached. The temperature is updated according to some cooling-schedule $T(k)$ in each step k.

The pseudocode for the general simulated annealing method can be found in the listing for Algorithm 4.

Algorithm 4 Simulated Annealing

```
1: procedure SIMULATED ANNEALING
2:     x ← random initial state
3:     T ← T_max,   k ← 0
4:     while T ≥ T_min do
5:         repeat
6:             Δx ← random perturbation of the state
7:             ΔE = E(x + Δx) − E(x)
8:             x ← x + Δx with probability P(ΔE, T)
9:         until Equilibrium reached for temperature T
10:        T ← T(k),   k ← k + 1
11:    end while
12: end procedure
```

When using the Boltzmann distribution with $P(\Delta E, T) = e^{-\frac{\Delta E}{T}}$ for $\Delta E > 0$ and $P(\Delta E, T) = 1$ if $\Delta E < 0$ in Algorithm 4, the resulting algorithm is called *Boltzmann Annealing*. This is the original *simulated annealing* that evolved from the Metropolis algorithm.

4.3.2 Convergence results

Given certain conditions, simulated annealing has been proven to converge towards a global optimum in [22]. Most importantly, the cooling schedule has to satisfy a logarithmic cooldown as described by the equation

$$T(k) \geq \frac{C}{\log(k+1)},$$

where k is the iteration in the cooling schedule and C is some constant.

Hajek [26] showed conditions for C which make the algorithm converge in probability to a global optimum.

Unfortunately, while these results prove that, conceptually, the algorithm is able to find a global optimum, they require cooling schedules that are not feasible in practice as it would be faster to solve the problem exactly.

Therefore, faster cooling schedules without convergence guarantees such as $T(k) = aT(k-1)$ with $0.85 \leq a \leq 0.96$ are often used in practice.

4.3.3 Simulated Annealing for MAX-SAT

The application of simulated annealing to MAX-SAT was first addressed in 1996, in an attempt to solve certain SAT instances that were hard to solve for traditional algorithms [45]. In the paper, Spears describes a simulated annealing variant with a deterministic state generator, a logistic acceptance function, and an exponential decay rate as the cooling schedule.

The state space for MAX-SAT consists of all possible assignments of the variables.

Generating the neighbor states is rather intuitive: Given an assignment of the variables, new states are sequentially generated by flipping one variable after another, at each step testing if the flip is accepted. That way, each variable is flipped and tested exactly once before the temperature is updated.

The energy function maps truth assignments to the total weight of the clauses satisfied by the respective assignment:

$$E(\mathbf{x}) = \sum_{c \in S_{\mathrm{SAT}}(\mathbf{x})} w_c$$

where $\mathbf{x}$ is the current truth assignment of the variables in V.

The increase or decrease in energy, ΔE, can be computed as the difference $E(\mathbf{x} + \Delta\mathbf{x}) - E(\mathbf{x})$ where $\Delta\mathbf{x}$ is the flip of one variable. But this is computationally expensive. Fortunately, it can also be computed directly: Let $\Delta_x E$ denote the increase or decrease in energy (i.e. the change in weight of satisfied clauses) when flipping the truth value of variable x, without loss of generality, from **true** to **false**. Only clauses in which x was the only satisfied variable become unsatisfied when flipping and only unsatisfied clauses containing $\overline{x}$ become satisfied. Thus keeping a list of positive and negative clauses, like before for the greedy algorithm, allows the computation to be done more efficiently. With this mapping and the number of satisfied literals for each clause, the change in energy can be expressed as

$$\Delta_x E = \sum_{c \in S_{\mathrm{UNSAT}}(\mathbf{x})} w_c - \sum_{\substack{c \in S_{\mathrm{SAT}}(\mathbf{x}) \\ c \setminus \{x\} \notin S_{\mathrm{SAT}}(\mathbf{x})}} w_c$$

A similar data structure is described in [45].

For the acceptance probabilities, the logistic function

$$P(\Delta E, T) = \frac{1}{1 + e^{-\frac{\Delta E}{T}}}$$

is used along with the cooling schedule

$$T(k) = T_{\max} e^{-kr(i)},$$

where k is the number of complete sweeps over the variables and $r(i) = \frac{1}{i \cdot |V|}$ is the decay rate with the number of retries i and the number of variables $|V|$.

The algorithm takes an additional parameter to specify the number of retries. Each retry starts a separate run of the algorithm with an individual random assignment of the variables. The main difference, however, is the decay rate, which is slowed down by the retry number i. Spears describes this definition of the decay rate $r(i)$, depending on the number of tries, as a reasonable choice. This is a form of *reannealing* described in [29], where the temperature decreases exponentially.

The full simulated annealing algorithm for MAX-SAT can then be put together as listed in Algorithm 5.

Algorithm 5 Simulated Annealing

procedure SIMULATED ANNEALING$_{\text{MAX-SAT}}$(S)
  for $i = 1 \ldots \max$ tries **do**
    $\mathbf{x} \leftarrow$ random initial assignment
    $T \leftarrow T_{\max}$
    while $T \geq T_{\min}$ **do**
      $T \leftarrow T_{\max} e^{-kr(i)}$
      for $x \in V$ **do**
        $\Delta_x E \leftarrow \sum_{c \in S_{\text{UNSAT}}(\mathbf{x})} w_c - \sum_{\substack{c \in S_{\text{SAT}}(\mathbf{x}) \\ x \text{ only sat. lit.}}} w_c$
        Flip x in $\mathbf{x}$ with probability $\frac{1}{1+e^{-\frac{\Delta_x E}{T}}}$
      end for
    end while
  end for
end procedure

4.3.4 Running time analysis

Given the cooling schedule $T(k) = T_{\max} e^{-kr(i)}$, the number of temperatures that need to be run through is

$$k \geq \frac{1}{r(i)} \ln \frac{T_{\max}}{T_{\min}} = in \ln \frac{T_{\max}}{T_{\min}},$$

where i is the current try and n is the total number of variables. We consider the parameters $T_{\max}$, $T_{\min}$ and the number of tries to be constant. Then the number of temperatures that need to be considered is $\mathcal{O}(n)$.

The calculation of $\Delta_x E$ for each variable x needs to iterate over all clauses that x and $\overline{x}$ occur in. Together with the loop over all variables, the cost per temperature amounts to

$$\sum_x \mathcal{O}(k_x + k_{\overline{x}}) = \sum_c \mathcal{O}(|c|) = \mathcal{O}(N),$$

with $N = |\varphi|$ again being the overall length of the formula.

Thus, the overall time complexity is $\mathcal{O}(nN) = \mathcal{O}(N^2)$.

4.4 Open-WBO

So far, only approximation algorithms for which optimality can only be guaranteed when the solution satisfies all clauses have been discussed here. There is, however, a whole range of algorithms for MAX-SAT which are able to solve the optimization problem exactly. For such algorithms, it is not conducive to consider their approximation ratios as the approximation ratios of exact algorithms are always 1. However, it is possible to compare these algorithms among each other using the time they need to arrive at the solution or on the number of instances they are able to solve before a given timeout is reached.

The latter is used as the metric in an event affiliated with the International Conference on Theory and Applications of Satisfiability Testing [46], called *MAX-SAT Evaluation* [2]. In this event (exact) MAX-SAT solvers are benchmarked against each other on submitted test instances. The goal for the submitted solvers in the evaluation is to solve as many instances from the test sets as possible before a given timeout occurs. In 2017, the two main tracks of the competition were the unweighted and the weighted track. Each track contains crafted and industrial instances as well as partial instances with hard and soft clauses. Instances in the "crafted" category come from MAX-SAT encodings of theoretical problems such as MAX-CUT, while

"industrial" instances encode problems such as circuit debugging problems. More information on the MAX-SAT Evaluation contest from 2017 can be found at `http://mse17.cs.helsinki.fi/index.html`.

In the track for unweighted instances, the solver that solved the largest number of instances before the timeout of one hour was the solver Open-WBO with the version `Open-WBO-RES`. It managed to solve 652 out of 880 instances (74%).

In the following, an overview of the tool Open-WBO is given from a practical perspective. This includes an outline of the used algorithms. Afterwards, the underlying algorithm of `Open-WBO-RES` for unweighted instances is described in more detail in order to contrast the approximation algorithms and metaheuristic discussed before with an exact optimization algorithm.

Open-WBO consists of different versions to solve MAX-SAT and its variants, `Open-WBO-RES` being only one of them. `Open-WBO-RES` is based on the *unsatisfiability-based* algorithms MSU3 [35] and OLL [38] for unweighted and weighted instances respectively. These algorithms iteratively refine a lower bound for the number of unsatisfiable clauses until the bound is large enough for the generated sub-problem to be satisfied. Another version is `Open-WBO-LSU` which employs a linear search algorithm that successively refines an upper bound for the number of unsatisfiable clauses until an optimal solution is found.

Both versions work by constructing and solving a sequence of SAT instances. These instances are solved by calling a SAT solver to find solutions to the respective sub-problems.

Due to the modular design of Open-WBO, any SAT solver implementing or extending the MiniSAT interface [16] can be used to solve the sub-problems. One benefit of this design choice is that it allows making use of performance improvements of SAT solvers to speed up Open-WBO without any additional work for Open-WBO itself. For the MAX-SAT Evaluation of 2017, the SAT solver `Glucose4.1` from `http://www.labri.fr/perso/lsimon/glucose/` was used. For unweighted MAX-SAT instances, `Open-WBO-RES` incorporates additional, partition-based techniques, which are heuristically enabled on a per-instance basis. A more detailed overview of Open-WBO and the other algorithms participating in the evaluation is given in [2].

4.4.1 Methods for solving MAX-SAT

Before discussing MSU3 in detail, we give a short overview of methods for solving MAX-SAT.

Solvers for MAX-SAT utilizing *branch and bound* compute a lower bound

and apply inference rules to simplify the formula in order to restrict the search space, e.g. [33, 1]. Another method for solving MAX-SAT instances is *pseudo boolean* (PB) optimization, e.g. [8]. With this method, blocking variables are added to clauses to allow for the overall formula to be satisfied, even if the clauses themselves are not satisfied with the current assignment of the original variables. Additionally to the blocking variables, a cost function that is to be minimized is introduced. The cost function consists of the sum of all blocking variables that are set to true. *Unsatisfiability*-based algorithms for MAX-SAT [19, 35] work by identifying minimal unsatisfiable sub-formulas, extracted from the proof traces of conflict-driven clause learning SAT solvers (CDCL). MSU3 belongs to this category and is described in the following section.

4.4.2 MSU3

As a member of the class of unsatisfiability-based algorithms, MSU3 makes use of minimal unsatisfiable sub-formulas in its calculations. The algorithm incorporates this by repeatedly constructing specific SAT formulas, for which a SAT solver provides the answer whether or not the formula is satisfiable. All but the last answer from the solver are thus that the provided formula is not satisfiable. Additionally, a minimal set of clauses that cannot be satisfied is returned. An outline of the algorithm from [35] is as follows:

1. Identify all (disjoint) unsatisfiable cores, that is minimal unsatisfiable clause sets.

2. Add one new, or *fresh*, variable as a positive literal to each clause in the unsatisfiable core. These variables are called *blocking variables*.

3. The number of unsatisfiable clauses is at least one for each core identified in step 1. This bound is used in the (linear) search for the exact number.

   Let k denote the current lower bound, namely the number of unsatisfiable cores. The formula is extended with the blocking variables b_i and the cardinality constraint $\sum b_i \leq k$ and is tested for satisfiability with k increasing in every step:

   First, the PB constraint that exactly k blocking variables are set to 1, and thus allowing k clauses to not be satisfied, is added to the formula. As long as the overall formula (with the PB constraint) is not satisfiable, the parameter for the number of unsatisfied clauses k is increased and the PB constraint is updated with the new parameter. Then the formula is submitted to the SAT solver once again. If, during an unsuccessful

try the unsatisfiable core contains a clause without blocking variable, a fresh blocking variable is added to the clause in the formula that is to be solved.

4. Finally, after step 3, the maximum number of clauses that can be satisfied is the number of total clauses in the formula, minus the current value of the parameter k.

The missing piece in the outline above is how to add the PB constraint to ensure that the number of blocking variables set to **true** is correctly bounded. This is addressed in the next section.

4.4.3 CNF Encoding

In step 3 of the algorithm above, the linear search for the exact number of unsatisfied clauses needs to maintain the constraint $\sum b_i = k$ on the boolean blocking variables b_i. One method for encoding this constraint as a CNF formula is the *Totalizer encoding* [7]. It was extended for Open-WBO to allow incremental updates to the constraint as they appear in the algorithm [36]. We now describe the basic method for generating a CNF formula for such a constraint.

Given the literals $l_1, \ldots, l_n$, the goal is to maintain the constraint that exactly k of those literals are satisfied. In the encoding, the number of satisfied literals is represented by boolean variables encoding a unary number, i.e. the number k between 1 and n is represented by the variables $y_1, \ldots, y_n$ with $y_1 = \cdots = y_k = 1$ and $y_{k+1} = \cdots = y_n = 0$. To ensure that the variables y_i correspond to the correct number of satisfied literals, a tree structure of additional variables and clauses is added, which encodes this cardinality relation. The leaves of the tree are the literals $l_1, \ldots, l_n$ that need to satisfy the constraint, while the root of the tree represents the cardinality of the constraint, k. All non-leaf nodes $N_{i_1} = (y_1^{(i_1)}, \ldots, y_{n_{i_1}}^{(i_1)}; n_{i_1})$ represent the number of satisfied literals in the leaves below it. The boolean variables $y_1^{(i_1)}, \ldots, y_{n_{i_1}}^{(i_1)}$ encode the number of those literals as a unary number as explained above where n_{i_1} is the maximum number of literals that can be represented in the node. The number n_{i_1} is chosen to be the total number of leaves of the subtree with root N_{i_1}. As an inner node, N_{i_1} has two child nodes that represent the literal counts k_2 and k_3 respectively. Therefore, N_{i_1} needs to encode the number k_1 with $k_1 = k_2 + k_3$. This relation is enforced

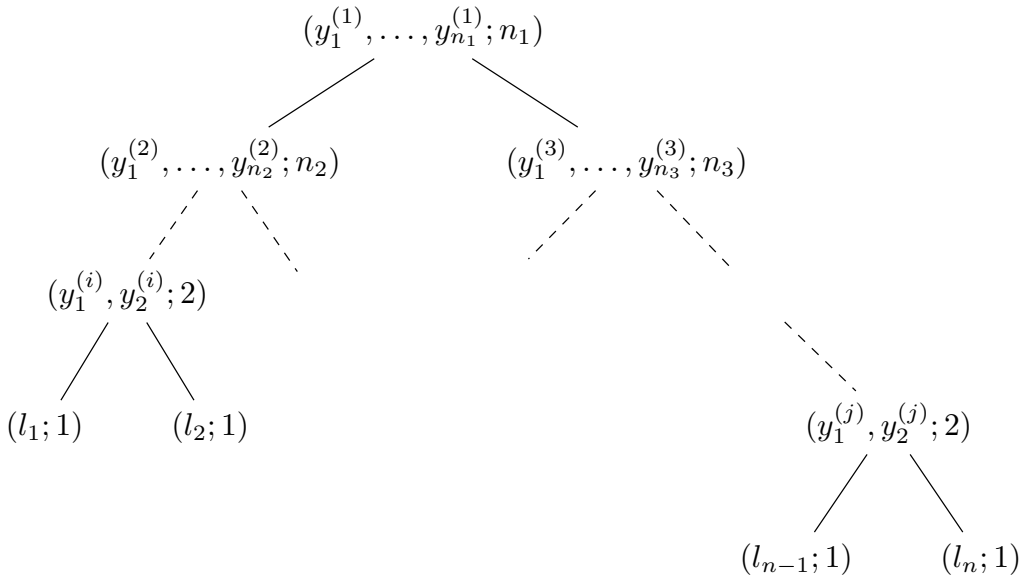

Figure 2: Tree representation of the Totalizer for $l_1 + \cdots + l_n \leq k$

by connecting the nodes using the formula

$$\bigwedge_{\substack{0 \leq k_2 \leq n_2 \\ 0 \leq k_3 \leq n_3 \\ 0 \leq k_1 \leq n_1 \\ k_2 + k_3 = k_1}} \overline{y}_{k_2}^{(i_2)} \vee \overline{y}_{k_3}^{(i_3)} \vee y_{k_1}^{(i_1)}$$

with $y_0^{(i_1)} = y_0^{(i_2)} = y_0^{(i_3)} = 1$ to allow that one sibling contains no true literals. Strictly speaking, the formula encodes the relation $k_1 \geq k_2 + k_3$. But with

$$\bigwedge_{j=k+1,\ldots,n} \overline{y}_j$$

the constraint that at most k literals are true can be enforced. Finally, these formulas together with the unit clauses of the variables y_i from the root node taken in conjunction represent the desired cardinality constraint.

Summing up, the literals $l_1, \ldots, l_n$ required to satisfy the cardinality constraint $l_1 + \cdots + l_n \leq k$ are placed separately in the leaves of the binary tree. For each internal node of the tree, the clauses that are added ensure that the node counts at least as many **true** literals as its two child nodes. These constraints, starting at the leaves and going up to the root, ensure that the root counts all satisfied literals. This is visualized in Figure 2.

5 Computational study

This section reports on the computational study which has been a major part of this thesis. The goal is to evaluate the practical behaviour of the algorithms described in Section 4. In the following, the algorithms will be identified by their names as used in the command line tool that is elaborated in Section 5.2.1. These identifiers are `annealing`, `greedy` and `slack` for the respective algorithm. Additionally, `greedy-s` as a variant of `greedy` has been included. For details see Section 5.1.2.

Our main point of interest lies in the quality of the solution as measured by the approximation ratio. This creates a problem because the approximation ratio depends on the number of satisfied clauses in the optimal solution, which is usually not available. The weaker performance ratio, where the total number of clauses is used as an upper bound for the optimal solution, will therefore be considered instead. Recall that the performance ratio is a lower bound for the approximation ratio.

Another metric that will be considered is the time needed by the algorithms to arrive at their final result. The times are expected to differ the most between the greedy algorithms and simulated annealing, as the greedy algorithms are sharing a common data structure and are thus expected to have similar runtimes.

Before we report about the results of the experiments we will comment on some implementation details. After that, the test environment is described. This includes usage of the solver, input and output formats and the machine specification on which the instances are solved. Finally, the benchmark test sets, taken from the MAX-SAT Evaluations in 2016 and 2017, are described together with the results of the respective experiments.

5.1 Implementation details

In addition to the theoretical discussions of the algorithms in Section 4, a few practical considerations are necessary when conducting the computational experiments.

5.1.1 Simulated annealing

For simulated annealing, reasonable default values for the parameters that show up in the algorithm are suggested in [45]. In our experiments we set $T_{\max} = 0.3$ for the initial temperature and $T_{\min} = 0.01$ for the minimal temperature, at which the system is considered cooled down. The number of retries is set to 1, since early tests showed that the restricting factor in

solving MAX-SAT instances with simulated annealing is the computation time. Additionally, the algorithm seemed to yield good results even with only one attempt.

Our implementation of simulated annealing also allows for a timeout to be specified, as tests during the initial implementation quickly revealed a much higher computation time than for the other algorithms. This timeout considers only the actual computation time of the algorithm because reading and preprocessing the instance cannot be interrupted. In the following, the time limit is set to $120s$ as is used in [42]. The actual time reported by the algorithm can exceed the specified timeout since the algorithm checks between iterations whether the specified time has passed and, if so, breaks from the loop and subsequently all loops it is nested in. Additionally, some extra time is needed for postprocessing after the actual computation to extract the solution from the data structure used in the specific algorithm. This can further increase the total time even after the timeout has passed, especially for larger instances.

5.1.2 Greedy and slack algorithm

The performance guarantee for the Slack algorithm (Theorem 4.2) states an expected approximation ratio of $\frac{3}{4}$. Since this is a probabilistic result, individual values can be lower than that, as can be the case for the minimal values in the summary tables presented in Section 5.4. For better comparison with the probabilistic approximation guarantee for the Slack algorithm, the reported performance ratios for this algorithm should be considered in aggregated form. This means considering the mean ratio for a set of instances or the average of multiple runs of the same instance instead of considering each value individually.

In [42], which served as a starting point for the present evaluation, a variant of Johnson's algorithm was considered instead of the original algorithm described in Section 4.1. To allow for better comparison with the results from [42], we implemented that variant as well. It is based on the deviation from Johnson's algorithm in the derivation of the Slack algorithm, which doubles the weight of unit clauses and leaves the weights of all other clauses as they are. As with the Slack algorithm before, the implementation of this variant uses the same greedy framework and simply replaces the local decision for each variable, given the relevant clauses. To distinguish the variant of the greedy algorithm from the original greedy algorithm, the variant will be labeled `greedy-s` in the following, where `s` stands for *simple* or *slack*.

5.2 Test Environment

Before discussing specific test instances and reporting on the experimental results for them, we describe the environment in which the instances are solved. This includes the user interface to the solver implemented for the computational study with the respective options for each algorithm, input and output formats, as well as the runtime environment with the hardware used and the runtime configuration options.

5.2.1 Solver User Interface

The solver for the computational study of this thesis is implemented as the command line program `maxsat-exe` and is written in the functional programming language Haskell (`https://www.haskell.org/`) using the compiler `GHC 8.0.2`. It reads an instance definition from a file, applies the specified algorithm, and prints the solution, i.e. the variable assignment, to the screen. Optionally, a file can be specified upon command invocation to which the same summary information is written in CSV format. The command line tool provides a simple yet useful interface to the implemented algorithms, including a help text outlining the correct usage of the program.

```
# ./maxsat-exe --help
maxsat -- A MAX-SAT solver

Usage: maxsat-exe COMMAND [-v|--verbose] [-s|--stats] [-z|--summary ARG]
                  [-w|--weighted] FILES

Available options:
  -v,--verbose             Whether to print verbose log messages
  -s,--stats               Whether to collect and print statistics
  -z,--summary ARG         If specified, a summary of the simulation will
                           be written to the file
  -w,--weighted            Whether to run instances with specified weights.
                           Has no effect for unweighted instances
  FILES                    Input file in Dimacs format
  -h,--help                Show this help text

Available commands:
  greedy                   Solve using greedy algorithm
  slack                    Solve using slack algorithm
  annealing                Solve using simulated annealing
  greedy-s                 Solve using the Poloczek variant of the greedy
                           algorithm
```

The `annealing` algorithm offers additional parameters that are specific to this algorithm. These include the previously mentioned tuning parameters as

well as the timeout parameter. The corresponding help text reads as follows:

```
# ./maxsat-exe annealing --help
Usage: maxsat-exe annealing [-t|--tries ARG] [-l|--min-temp ARG]
                            [-u|--max-temp ARG] [--timeout ARG]
  Solve using simulated annealing

Available options:
  -t,--tries ARG           tries (default: 1)
  -l,--min-temp ARG        min-temp (default: 1.0e-2)
  -u,--max-temp ARG        max-temp (default: 0.3)
  --timeout ARG            If specified, the timeout in seconds
  -h,--help                Show this help text
```

5.2.2 Dimacs-Format

The command line tool uses the DIMACS format for input and output. It is the mandatory input and output format for the participating solvers in the annual MAX-SAT Evaluation and is adopted here for consistency. Unweighted instances are specified in the DIMACS CNF format whereas the DIMACS WCNF format is used for weighted problem definitions. The full specification is available at `http://www.maxsat.udl.cat/16/`.

DIMACS CNF and WCNF are line-based text file formats that consist of three elements: *Parameters*, *clauses* and optional *comment* lines.

The format for unweighted instances is discussed first: The instance file may start with one or more comment lines. Each comment line starts with `c`, usually followed by a description of the instance or additional information. These comment lines are ignored by the solver.

The line specifying the parameters is non-optional and has to be in the format

`p cnf <nbVars> <nbClauses>.`

It starts with `p`, followed by the instance type, the number of variables `<nbVars>`, and the total number of clauses `<nbClauses>`. In the example above, the instance type is `cnf`, identifying the instance as an *unweighted* CNF formula.

After the parameters, the actual clauses of the instance are listed. In that list of clauses, variables are encoded as integers from 1 to `<nbVars>`, with negated variables represented as the respective negative integer. Each clause has to be written on a separate line, wherein the positive and negative variables are separated by spaces and postfixed with `0` to signal the end of the clause.

For example, consider the following formula in 5 variables and 4 clauses

in conjunctive normal form:

$$\varphi = (x_1 \vee x_2) \wedge (x_2 \vee \overline{x}_3) \wedge (\overline{x}_3 \vee x_4) \wedge (x_1 \vee \overline{x}_2 \vee x_5).$$

This formula can be represented as the following input file in the DIMACS format:

```
c
c Test Instance 1
c
p cnf 5 4
1 2 0
2 -3 0
-3 4 0
1 -2 5 0
```

The format for *weighted* instances is similar. Here, the parameter line has the format

```
p wcnf <nbVars> <nbClauses> <maxWght>
```

with the instance type `wcnf` and an additional parameter `<maxWght>`, which specifies the maximum weight of a single clause.

The clauses themselves are written as in the unweighted case, but with the (integer) weight of the clause at the beginning of the line, followed by the list of literals.

As an example, the previous formula augmented with weights 8, 8, 8 and 12 would be written as follows:

```
c
c Weighted Test Instance 1
c
p wcnf 5 4 12
8 1 2 0
8 2 -3 0
8 -3 4 0
12 1 -2 5 0
```

The output format for printing messages to either the screen or a log file is similar to the input format. Just like the input file format it is line-based, with similar prefixes distinguishing different messages. There are 4 distinct message types:

- The *objective value*, defined as the number of unsatisfied clauses, is prefixed with `o`. There can be more than one such line in the output log, with only the last one being valid. This way, the algorithm can print objective values for intermediate solutions and update them whenever a better solution is found.

- The *type* of the *solution* is prefixed with `s`, and can either be
  - `OPTIMUM FOUND` if the last line with an objective value is that of an optimal solution,
  - `SATISFIABLE` if the algorithm found a variable assignment satisfying all hard clauses of the instance, or
  - `UNKNOWN` otherwise.

  If no such message is printed, `UNKNOWN` can be assumed implicitly.
- The *variable assignment* for the printed objective value is shown as a spaces-separated list of satisfied literals, prefixed with `v`.
- Finally, all other log messages and information that are printed are interpreted as *comments* and are prefixed with `c`.

We give an example of what such a log might look like:

```
c maxsat solver
c
c Version:    2c5c8075
c Opts:       AlgGreedy
c Started at: 2018-01-09 19:22:00.723561 UTC
c
c Reading file sbox_4.wcnf
c Elapsed Time: 651.32 us
c Number of clauses: 387, number of variables: 147
c Running algorithm
o 35
v 1 2 3 4 5 6 7 -8 9 -10 11 12 13 14 15 -16 -17 -18 -19 -20 -21 -22 -23 24
 25 26 27 28 -29 -30 -31 -32 -33 -34 -35 36 37 38 39 -40 -41 -42 -43 -44
 -45 -46 -47 -48 -49 -50 -51 -52 -53 -54 -55 -56 -57 -58 59 60 61 62 -63
 -64 -65 -66 -67 -68 -69 -70 -71 -72 -73 -74 -75 76 77 78 79 -80 -81 -82
 -83 -84 -85 -86 -87 88 -89 -90 -91 -92 -93 -94 -95 -96 -97 -98 -99 -100
 -101 -102 -103 -104 -105 -106 -107 -108 -109 -110 -111 -112 -113 -114
 -115 116 117 -118 119 -120 -121 -122 123 -124 -125 -126 -127 -128 -129
 -130 -131 -132 -133 -134 -135 -136 -137 -138 -139 -140 -141 -142 -143 -144
 -145 -146 -147
c Ratio: 0.91 (352 of 387 satisfied)
c Elapsed Time: 1.88 ms
```

5.2.3 Machine Specification

The tests were conducted on a Debian Linux workstation located at the Institute of Optimization and Discrete Mathematics at Graz University of Technology, Austria. The machine is running the kernel `Linux 4.9.0-5-amd64`

`#1 SMP Debian 4.9.65-3+deb9u2 (2018-01-04) x86_64` on an `Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz` processor with 32 cores, 128GB of memory and a 1TB HDD (SATA 6.0Gb/s) storage space with 7200 RPM and 128MB cache.

5.2.4 Batch system

Running all instances with the different algorithms and collecting their results is in itself a task to be considered carefully. With 911 instances from the 2016 test sets and 880 instances from 2017 in combination with 4 algorithms, naively processing the more than 7000 jobs sequentially would require a significant amount of time. Luckily, the available computer with its 32 cores and 128GB of memory allows for a good amount of parallelization.

The instances of each category (as defined below) are solved in parallel by all implemented algorithms, with one algorithm at a time. Recall that those are `greedy`, `slack`, `annealing`, and `greedy-s`. The solver is restricted to 1 CPU core for each instance as to not influence other jobs during parallel execution. This is done via the runtime options `+RTS -N1 -RTS`. The script coordinating the benchmarks runs at most 30 jobs at the same time. This utilizes 30 of the 32 cores of the machine at once, with 2 cores to spare for overhead and an interactive shell session to monitor the progress.

Unless specified otherwise, all instances are interpreted as unweighted, run with the statistics flag `-s` enabled, the verbose flag `-v` disabled, and the output piped into an instance-specific log file. The timeout is set to 120 seconds with the option `--timeout 120`. Recall that this only affects `annealing`, and only the actual optimization process without pre- or postprocessing. All other parameters are not changed.

5.3 Benchmarks

Below we describe the benchmark test sets from the MAX-SAT Evaluations in 2016 and 2017 [3, 2] that are used to evaluate the implemented algorithms.

5.3.1 MAX-SAT Evaluation 2016

From the evaluation in 2016 [3], three test sets with unweighted instances are considered: The `ms_random` test set consisting of randomly generated instances, `ms_crafted` containing instances specifically designed to test the performance of MAX-SAT solvers, and `ms_industrial` containing instances used to solve real-world problems. Let us provide some more details on the chosen three sets of test instances.

- `ms_random`: 454 instances in 10 categories.

  In the test set of randomly generated instances, the number of variables ranges from 70 to 300 and the number of clauses from 700 to 2,600. The ratios between the number of clauses and variables, which give a rough overview of the structures of the instances, lie between 4.0 and 21.7.

- `ms_crafted`: 402 instances in 11 categories.

  In this test set, the number of variables lies between 27 and 11,264 with a median of 140. The number of clauses ranges from 48 to 39,424 (median 1,400) and the ratio of clauses to variables lies between 1.1 and 57.0.

- `ms_industrial`: 55 instances in 2 categories.

  The industrial test set is the smallest test set used in the MAX-SAT Evaluation of 2016 but features the largest instances. The number of variables starts at 45,552 but may be as high as 4,426,323 (median 400,085) and the number of clauses is between 140,056 and 15,983,633 (median 1,221,020). With 4,426,323 variables and 15,983,633 clauses `mem_ctrl-problem.dimacs_27.filtered.cnf` is the largest file with around 616.2 MB. The clause to variable ratio is smaller than for the test sets before, namely between 1.1922 and 4.7597.

5.3.2 MAX-SAT Evaluation 2017

In the evaluation of 2017, the organization of the test instances was revised. The test set `ms_crafted` and `ms_industrial` of the *partial* track from the previous years were merged, while dropping the random instances from `ms_random` altogether. Other non-random instances were added instead. For new instances, the editors of the evaluation imposed the requirement that each new category of test instances needs to be described briefly in the format as is suggested for IEEE Proceedings and which can be found at `https://www.ieee.org/conferences_events/conferences/publishing/templates.html`. Both the editors' discussion as well as the actual instance descriptions can be found in [2].

Overall, 880 instances in 36 categories were used in the evaluation and our experiments. The number of variables ranges from 27 to 2,785,108 with a median of 19,778, while there are between 48 and 13,901,121 clauses per instance (median 108,706). The ratio between clauses and variables ranges further than for the test sets before. It starts at 0.1 and goes up to 1276.0, with median 4.7.

In addition to the results of the implemented algorithms, the results of `Open-WBO-RES`, retrieved from `http://mse17.cs.helsinki.fi/rankings.html`, have been included in the following discussion of computational results. From the solver log files in DIMACS format, the variable assignment of the solution, the objective value, and the type of the solution (optimal, satisfiable, or unknown) are extracted and stored in a CSV file for later processing.

The MAX-SAT Evaluation 2017 used the StarExec computer cluster (`https://www.starexec.org/`) whose 192 computation nodes consist of `Intel(R) Xeon(R) CPU E5-2609 0 @ 2.40GHz` processors with 128GB of main memory each. Each individual solver run was restricted to 1 CPU core with a time limit of 1 hour and a memory limit of 32GB.

Including the results of an exact solver allows to compare the results of the approximation algorithms to the actual solution of the optimization problem. With these optimal solutions, the approximation ratios can be considered in addition to the weaker performance ratios.

The instances in the "complete, unweighted" track from the MAX-SAT Evaluation 2017 are unweighted but they include soft and hard instances, which the approximation algorithms are unable to differentiate. Recall that for such instances some clauses are marked as hard and thus have to be satisfied in order for a solution to be feasible. However, in the DIMACS format such instances are represented as weighted instances where the hard clauses have a weight equal to the maximum weight parameter of the instance. As long as that maximum weight is guaranteed to be greater than the sum of all soft clauses, maximizing the weight of satisfied clauses should lead to satisfying the hard clauses first. Of course, for the extreme case where every clause is hard, the problem is reduced to a conventional SAT instance, for which it is NP-hard to find a solution.

As a simple heuristic for such unweighted partial instances, those instances are interpreted as weighted (non-partial) instances and solved as such. Then, for the performance and approximation ratios the instances are again interpreted as unweighted and only the respective number of clauses is taken into account.

5.4 Results of the computational study

The statistical analyses of the solver runtimes and performance ratios were conducted in R (`https://www.r-project.org/`). The presented tables contain summaries of the respective variables grouped by algorithm. We state *minimum*, *maximum* as well as *median*, *first*, and *third quartile* to provide insight into the distribution of the data. In the tables the quartiles are denoted by *q1* and *q3*. Also included are *mean* and *standard deviation*, abbreviated

as *sd*, as measures for location and dispersion.

5.4.1 MAX-SAT Evaluation 2016

The results of the computational study for the benchmarks from the MAX-SAT Evaluation 2016 are presented separately for the three test sets `ms_random`, `ms_crafted` and `ms_industrial`. We give a table containing the statistical summary of the runtimes of the implemented algorithms in the `maxsat-exe` tool, as well as a table containing the statistical summary of the respective performance ratios for each of the three test sets. These are Tables 1 to 6.

After the results per test set are described, we consider the performance ratios, again for each of algorithms, split into their respective categories.

Finally, because `annealing` exhibits behaviour that differs from that of the other algorithms, the results for `annealing` are described in more detail.

	min	q1	median	q3	max	mean	sd
greedy	0.001	0.002	0.003	0.003	0.013	0.003	0.001
slack	0.001	0.002	0.003	0.003	0.011	0.003	0.001
annealing	0.027	0.106	0.192	0.307	0.837	0.234	0.176
greedy-s	0.001	0.002	0.002	0.003	0.008	0.003	0.001

Table 1: Solver runtime for `ms_random`, $n = 454$

	min	q1	median	q3	max	mean	sd
greedy	0.793	0.839	0.887	0.954	0.989	0.896	0.059
slack	0.745	0.782	0.822	0.909	0.967	0.847	0.069
annealing	0.801	0.855	0.902	0.970	0.998	0.913	0.062
greedy-s	0.793	0.839	0.887	0.950	0.984	0.895	0.058

Table 2: Performance ratio for `ms_random`, $n = 454$

`ms_random` Looking at the times the algorithms needed to find approximate solutions for the instances of the `ms_random` test set in Table 1, it is clear that with these instance sizes neither of the algorithms had any problems finding such an approximation quickly. Even `annealing` took at most a little under a second to find a variable assignment. The greedy algorithms `greedy`, `slack` and `greedy-s` generally only show small differences in solver time. This is to be expected because all three algorithms only differ in how to fix a variable (local decision), but not in the overall structure of the algorithm.

On the performance side, we can see in Table 2 that the greedy algorithms `greedy` and `greedy-s` are far above the theoretical worst-case of $\frac{2}{3}$, even

if we are considering the performance ratio instead of the approximation ratio. The algorithm with the highest mean performance ratio is simulated annealing. Surprisingly, the Slack algorithm comes in last, with Johnson's greedy algorithm being second. The order of `greedy` and `slack` is particularly interesting because this is the opposite of what would be expected from the theoretical results. It is, however, in accordance with the experimental findings in [42], where similar results were observed for the MAX-SAT and SAT evaluation test sets from the years 2014 and 2015. Comparing the MAX-SAT instances from 2014 and 2015 with those from 2016 is justified since the test sets stayed mostly the same, with only few additions from 2014 to 2015 and few removals from 2015 to 2016.

	min	q1	median	q3	max	mean	sd
greedy	0.000	0.002	0.002	0.004	0.131	0.004	0.008
slack	0.000	0.002	0.002	0.003	0.115	0.003	0.007
annealing	0.005	0.162	0.196	0.314	120.361	0.902	8.550
greedy-s	0.000	0.002	0.002	0.003	0.116	0.003	0.007

Table 3: Solver runtime for `ms_crafted`, $n = 402$

	min	q1	median	q3	max	mean	sd
greedy	0.756	0.820	0.831	0.840	0.990	0.828	0.027
slack	0.732	0.770	0.778	0.786	0.966	0.779	0.022
annealing	0.756	0.842	0.858	0.865	0.993	0.851	0.032
greedy-s	0.756	0.819	0.831	0.839	0.963	0.827	0.026

Table 4: Performance ratio for `ms_crafted`, $n = 402$

`ms_crafted` The results of the crafted instances are similar to those for `ms_random` before. `annealing` is still able to solve most instances within the given time but the maximum solver time of over $120s$ indicates that the timeout has been reached at least once (Table 3). As can be seen in Table 4, this does not impact the performance ratios much. The average performance ratios of the algorithms are ranked as before, with `annealing` fairing best, `greedy` and `greedy-s` lying close together thereafter and `slack` following behind with a considerable gap.

`ms_industrial` The industrial instances are perhaps the most interesting ones, not only because they have real-world applications but because the results differ from the previous test sets. The most striking thing to notice is

	min	q1	median	q3	max	mean	sd
greedy	0.464	3.647	5.626	17.222	59.593	11.671	13.101
slack	0.404	4.488	7.103	19.480	64.527	12.736	13.528
annealing	120.567	124.961	127.901	136.719	179.478	133.660	13.638
greedy-s	0.522	3.766	5.838	15.552	57.909	11.489	12.648

Table 5: Solver runtime for `ms_industrial`, $n = 55$

	min	q1	median	q3	max	mean	sd
greedy	0.939	0.969	0.981	0.989	0.994	0.978	0.013
slack	0.879	0.914	0.919	0.925	0.934	0.916	0.013
annealing	0.763	0.812	0.823	0.842	0.938	0.827	0.033
greedy-s	0.949	0.971	0.982	0.991	0.994	0.980	0.011

Table 6: Performance ratio for `ms_industrial`, $n = 55$

that `annealing` comes in on the last place here. The solver times in Table 5 are, however, able to shed some light on this result. Even the minimum solver time for an instance of the `ms_industrial` test set with `annealing` is above $120s$, meaning that `annealing` was not able to solve a single instance before the timeout occurred.

The results for the greedy algorithms are consistent with the results before. However, the ratios for these algorithms are a bit higher in general (Table 6). With average performance ratios of just below 98%, `greedy` and `greedy-s` are a good fit for those instances.

Results per category The tables with the mean performance ratios for each category and for each test set in Appendix A show that different categories of instances can have quite different performance ratios. Of course, this can be explained by their different structures and the actual numbers of satisfiable clauses in each instance. What the categories do have in common, however, is that they all yield very similar rankings between the algorithms. For all categories except for the ones in `ms_industrial`, `annealing` yields the highest ratios, followed by both greedy variants `greedy` and `greedy-s` close together, and `slack` trailing a bit behind. The poorer performance of `annealing` for the industrial instances is again easily explained by the fact that for those, `annealing` has a mean solver time greater than $120s$, meaning that the solver times out before finishing its computations.

A closer look at the results for `annealing` We are interested in if and by how much the performance ratios of `annealing` improve when the timeout

	annealing120	annealing600	greedy
mean	0.829	0.839	0.974
sd	0.034	0.042	0.026

Table 7: Performance ratios when extending the timeout for `annealing` from 120s to 600s, $n = 57$

is increased. To answer this question, the instances from all test sets for which `annealing` does not finish within the timeout of $120s$ are solved again with a new timeout of $600s$.

The results are presented side by side in Table 7 for an easy comparison. Although the performance ratios of the instances in question are improved by about one percent there still is a big gap when compared to `greedy`. This is because most of the instances that ran into the timeout of $120s$ also ran into the timeout of $600s$. Almost all of these instances come from the `ms_industrial` test set.

It helps to take a closer look at the convergence behavior of simulated annealing in order to get a better understanding of the algorithm. Figure 3 shows a typical energy graph of the simulated annealing process for one of the instances of the `ms_crafted` test set with 5,120 variables and 16,640 clauses. The number of satisfied clauses is plotted against the temperature during the course of the algorithm. As an initial state for the algorithm, a random assignment of the variables is chosen. This assignment already satisfies about 14,000 of the 16,640 clauses. From there, the algorithm tries to find better assignments but is also allowed to choose a worse assignment with some probability depending on the current temperature. This is reflected in the graph by a funnel that slowly gets narrower as the temperature decreases. At around temperature 0.10, the system has cooled off and only small changes in the number of satisfied clauses are visible.

One aspect that stands out is that the initial random assignment is already a good approximation. After a few steps, the algorithm shows linear improvement of the number of satisfied clauses. It keeps this tendency for about half of the simulation, after which the curve flattens until it reaches its final approximation. Tuning the parameter for the minimum temperature could save some runtime, with the drawback of a possibly slightly worse performance ratio. It might be worth investigating whether a more sophisticated simulated annealing routine might lead to improved results for the industrial instances. Another option might be to try out other metaheuristics such as *Tabu search* [23].

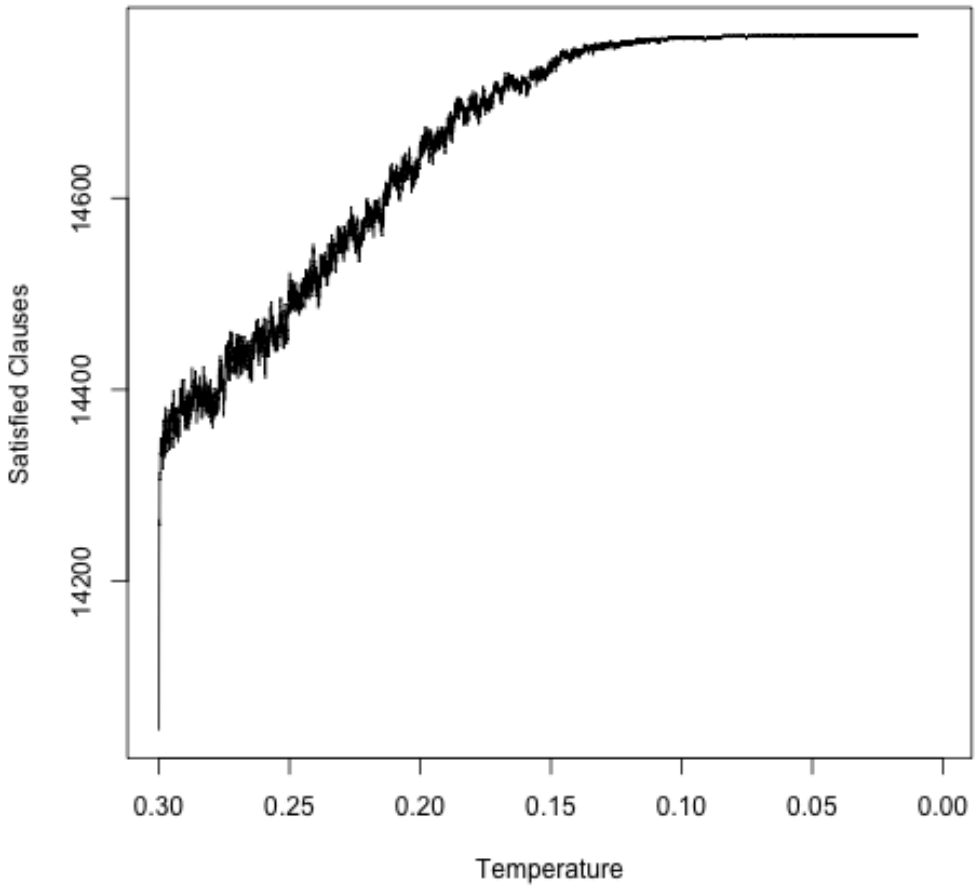

Figure 3: Energy graph of the annealing process of instance `ms_crafted/set-covering/scpcyc-scpcyc10_maxsat`

5.4.2 MAX-SAT Evaluation 2017

As for the results of the computational study of the benchmarks from the MAX-SAT Evaluation 2016, we give the runtimes of the implemented algorithms together with the respective performance ratios in Tables 8 and 9. Furthermore, the results from the solver `Open-WBO-RES` are combined with the results of the implemented algorithms `greedy`, `greedy-s`, `slack`, and `annealing` to list the approximation ratios in Table 10.

The results are similar to those for the 2016 evaluation. Overall, the average performance ratios are better than the theoretical results would

	min	q1	median	q3	max	mean	sd
greedy	0.000	0.052	0.462	2.257	52.089	2.070	4.888
slack	0.000	0.054	0.471	2.388	50.836	2.184	5.107
annealing	0.006	40.175	120.479	122.195	197.516	91.233	51.747
greedy-s	0.000	0.046	0.429	2.184	51.151	2.008	4.821

Table 8: Solver runtime (total) for `MSE17`, $n = 880$

	min	q1	median	q3	max	mean	sd
greedy	0.628	0.939	0.979	0.994	1.000	0.952	0.066
slack	0.607	0.904	0.944	0.970	1.000	0.924	0.066
annealing	0.621	0.852	0.950	0.985	1.000	0.915	0.084
greedy-s	0.629	0.948	0.978	0.994	1.000	0.950	0.069

Table 9: Performance ratio (total) for `MSE17`, $n = 880$

suggest, with all of them being above 90%. Both `greedy` and `greedy-s` are again slightly ahead of `slack`. However, this time `annealing` comes in last, but with only a small gap to `slack`. Looking at the solver runtimes in Table 8, we see that the solver times for `annealing` are quite close to the timeout of $120s$. Precisely 626 out of 880 instances did not finish before the timeout.

For the approximation ratios the results from `Open-WBO-RES` are used as a reference for the optimal value whenever `Open-WBO-RES` was able to find an optimal solution (in 652 out of 880 cases). The approximation ratio is only calculated for instances where the approximation algorithms found a solution that satisfies all hard clauses. This is reflected in the column "n" in Table 10.

	min	q1	median	q3	max	mean	sd	n
greedy	0.702	0.962	0.983	0.998	1.000	0.961	0.064	140
slack	0.803	0.915	0.971	0.998	1.000	0.953	0.050	132
annealing	0.687	0.915	0.989	1.000	1.000	0.944	0.080	158
greedy-s	0.702	0.963	0.986	0.998	1.000	0.959	0.067	133

Table 10: Approximation ratio (total) for `MSE17`

Out of the 652 instances for which `Open-WBO-RES` was able to find an optimal solution, `annealing` also found an optimal solution for 11 instances.

The implemented algorithms `greedy`, `greedy-s` and `annealing` were able to find a solution satisfying the hard clauses in 13 out of 36 categories. Only `slack` solved instances from 2 fewer categories than the rest. Generally, this suggests that interpreting instances with hard clauses as weighted instances as described can be a viable heuristic if the structure of the problem is adequately simple.

6 Conclusion

This thesis evaluated the practical performance of two approximation algorithms for MAX-SAT, *Johnson's greedy* algorithm and the *Slack* algorithm of Poloczek and Schnitger, as well as the metaheuristic *simulated annealing* (MAX-SAT version from Spears).

We found that both Johnson's greedy algorithm and the Slack algorithm have better average performance ratios on a wide range of test instances than what can be proven in theory. Simulated annealing yielded the best results for the instances on which it was able to finish its computation before a specified timeout occurred but should be avoided for larger instances that would require more computation time than the timeout permits.

The better theoretical performance guarantee of the Slack algorithm over Johnson's greedy algorithm did not carry over to the practical evaluation. Johnson's algorithm consistently managed to outperform the Slack algorithm.

All evaluated algorithms were also able to find feasible solutions to some of the partial instances. However, the instances for which feasible solutions were found are concentrated to few categories. This suggests that these categories share a common structure particularly suited for the evaluated algorithms.

Future work could include, but is not restricted to, the following:

- More algorithms could be incorporated into the evaluation. The algorithms implemented for this thesis use simple data structures and do not make use of techniques such as linear or semidefinite programming. In particular, the algorithm from Asano and Williamson [5], with an approximation ratio of 0.7877 [4], would be of interest due to its good theoretical guarantees.

- The authors of [44] use a derandomization of the Slack algorithm with two iterations over the variables to achieve a $\frac{3}{4}$-approximation algorithm, which the authors titled *2Pass*. In their evaluation, 2Pass outperformed both Johnson's greedy algorithm as well as the Slack algorithm [42]. Again, this algorithm could be included in future evaluations in order to reproduce these results.

- The structure of the partial instances could be investigated. The instances could be searched for some common criteria that might be able to explain their suitability for the tested algorithms. Such criteria could then be used to design heuristic approaches for MAX-SAT.

References

[1] T. Alsinet, F. Manya, and J. Planes. "Improved branch and bound algorithms for Max-SAT". In: (2003).

[2] "MaxSAT Evaluation 2017: Solver and Benchmark Descriptions". In: *Volume B-2017-2 of Department of Computer Science Series of Publications B*. Ed. by C. Ansotegui et al. University Of Helsinki, 2017.

[3] J. Argelich et al. *MAX-SAT 2016: Eleventh Max-SAT evaluation*. `http://www.maxsat.udl.cat/16/`. Accessed: 2018-01-07.

[4] T. Asano. "An Improved Analysis of Goemans and Williamson's LP-Relaxation for MAX SAT". In: *Fundamentals of Computation Theory: 14th International Symposium, FCT 2003, Malmö, Sweden, August 12-15, 2003. Proceedings*. Ed. by A. Lingas and B. J. Nilsson. Preliminary version. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 2–14. ISBN: 978-3-540-45077-1. DOI: `10.1007/978-3-540-45077-1_2`.

[5] T. Asano and D. P. Williamson. "Improved Approximation Algorithms for Max Sat". In: *Journal of Algorithms* 42.1 (2002), pp. 173–202. DOI: `10.1006/jagm.2001.1202`.

[6] A. Avidor, I. Berkovitch, and U. Zwick. "Improved Approximation Algorithms for MAX NAE-SAT and MAX SAT". In: *Approximation and Online Algorithms*. Approximation and Online Algorithms. Springer Berlin Heidelberg, 2006, pp. 27–40. DOI: `10.1007/11671411_3`.

[7] O. Bailleux and Y. Boufkhad. "Efficient CNF Encoding of Boolean Cardinality Constraints". In: *Principles and Practice of Constraint Programming – CP 2003*. Ed. by F. Rossi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 108–122. ISBN: 978-3-540-45193-8.

[8] E. Boros and P. L. Hammer. "Pseudo-boolean Optimization". In: *Discrete Appl. Math.* 123.1-3 (Nov. 2002), pp. 155–225. ISSN: 0166-218X. DOI: `10.1016/S0166-218X(01)00341-9`.

[9] N. Buchbinder et al. "A Tight Linear Time (1/2)-Approximation for Unconstrained Submodular Maximization". In: *2012 IEEE 53rd Annual Symposium on Foundations of Computer Science*. Oct. 2012, 649—658. DOI: `10.1109/focs.2012.73`.

[10] V. Černý. "Thermodynamical Approach To the Traveling Salesman Problem: An Efficient Simulation Algorithm". In: *Journal of Optimization Theory and Applications* 45.1 (1985), pp. 41–51. ISSN: 1573-2878. DOI: `10.1007/BF00940812`.

[11] J. Chen, D. K. Friesen, and H. Zheng. "Tight Bound on Johnson's Algorithm for Maximum Satisfiability". In: *Journal of Computer and System Sciences* 58.3 (1999), pp. 622–640. DOI: 10.1006/jcss.1998.1612.

[12] Y. Chen et al. "Automated Design Debugging With Maximum Satisfiability". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 29.11 (2010), pp. 1804–1817. DOI: 10.1109/tcad.2010.2061270.

[13] *containers: Assorted concrete container types*. http://hackage.haskell.org/package/containers-0.5.7.1. Version 0.5.7.1. 2016.

[14] S. A. Cook. "The complexity of theorem-proving procedures". In: *Proceedings of the third annual ACM symposium on Theory of computing - STOC '71*. May 1971, pp. 151–158. DOI: 10.1145/800157.805047.

[15] K.-L. Du and M. N. S. Swamy. "Simulated Annealing". In: *Search and Optimization by Metaheuristics: Techniques and Algorithms Inspired by Nature*. Cham: Springer International Publishing, 2016, pp. 29–36. ISBN: 978-3-319-41192-7. DOI: 10.1007/978-3-319-41192-7_2.

[16] N. Eén and N. Sörensson. "An Extensible SAT-solver". In: *Theory and Applications of Satisfiability Testing*. Ed. by E. Giunchiglia and A. Tacchella. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 502–518. ISBN: 978-3-540-24605-3.

[17] U. Feige and M. Goemans. "Approximating the value of two power proof systems, with applications to MAX 2SAT and MAX DICUT". In: *Proceedings Third Israel Symposium on the Theory of Computing and Systems*. 1995, pp. 182–189. DOI: 10.1109/istcs.1995.377033.

[18] U. Feige and L. Lovász. "Two-prover one-round proof systems". In: *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing - STOC '92*. 1992. DOI: 10.1145/129712.129783.

[19] Z. Fu and S. Malik. "On Solving the Partial MAX-SAT Problem". In: *Lecture Notes in Computer Science*. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 252–265. DOI: 10.1007/11814948_25.

[20] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979. ISBN: 0716710447.

[21] M. Garey, D. Johnson, and L. Stockmeyer. "Some Simplified Np-Complete Graph Problems". In: *Theoretical Computer Science* 1.3 (1976), pp. 237–267. DOI: 10.1016/0304-3975(76)90059-1.

[22] S. Geman and D. Geman. "Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-6.6 (1984), pp. 721–741. DOI: 10.1109/tpami.1984.4767596.

[23] F. Glover, E. Taillard, and E. Taillard. "A user's guide to tabu search". In: *Annals of Operations Research* 41.1 (1993), pp. 1–28. ISSN: 1572-9338. DOI: 10.1007/BF02078647.

[24] M. X. Goemans and D. P. Williamson. "Improved Approximation Algorithms for Maximum Cut and Satisfiability Problems Using Semidefinite Programming". In: *Journal of the ACM* 42.6 (1995), pp. 1115–1145. DOI: 10.1145/227683.227684.

[25] M. X. Goemans and D. P. Williamson. "New $\frac{3}{4}$-Approximation Algorithms for the Maximum Satisfiability Problem". In: *SIAM Journal on Discrete Mathematics* 7.4 (1994), pp. 656–666. DOI: 10.1137/s0895480192243516.

[26] B. Hajek. "Cooling Schedules for Optimal Annealing". In: *Mathematics of Operations Research* 13.2 (1988), pp. 311–329. DOI: 10.1287/moor.13.2.311.

[27] J. Håstad. "Some Optimal Inapproximability Results". In: *Journal of the ACM* 48.4 (2001), pp. 798–859. DOI: 10.1145/502090.502098.

[28] W. K. Hastings. "Monte Carlo Sampling Methods Using Markov Chains and Their Applications". In: *Biometrika* 57.1 (1970), pp. 97–109. DOI: 10.1093/biomet/57.1.97.

[29] L. Ingber. "Very Fast Simulated Re-Annealing". In: *Mathematical and Computer Modelling* 12.8 (1989), pp. 967–973. DOI: 10.1016/0895-7177(89)90202-1.

[30] D. S. Johnson. "Approximation Algorithms for Combinatorial Problems". In: *Journal of Computer and System Sciences* 9.3 (1974), pp. 256–278. DOI: 10.1016/s0022-0000(74)80044-9.

[31] H. Karloff and U. Zwick. "A 7/8-approximation algorithm for MAX 3SAT?" In: *Proceedings 38th Annual Symposium on Foundations of Computer Science*. 1997, pp. 406–415. DOI: 10.1109/sfcs.1997.646129.

[32] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. "Optimization By Simulated Annealing". In: *Science* 220.4598 (1983), pp. 671–680. DOI: 10.1126/science.220.4598.671.

[33] C. M. Li, F. Manyà, and J. Planes. "Exploiting Unit Propagation to Compute Lower Bounds in Branch and Bound Max-SAT Solvers". In: *Principles and Practice of Constraint Programming - CP 2005*. Ed. by P. van Beek. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 403–414. ISBN: 978-3-540-32050-0.

[34] P.-C. Lin and S. P. Khatri. "Application of Max-Sat-Based Atpg To Optimal Cancer Therapy Design". In: *BMC Genomics* 13.Suppl 6 (2012), S5. DOI: 10.1186/1471-2164-13-s6-s5.

[35] J. Marques-Silva and J. Planes. "On Using Unsatisfiability for Solving Maximum Satisfiability". In: *CoRR* (2007). arXiv: 0712.1097 [cs.AI].

[36] R. Martins et al. "Incremental Cardinality Constraints for MaxSAT". In: *Lecture Notes in Computer Science*. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 531–548. DOI: 10.1007/978-3-319-10428-7_39.

[37] N. Metropolis et al. "Equation of State Calculations By Fast Computing Machines". In: *The Journal of Chemical Physics* 21.6 (1953), pp. 1087–1092. DOI: 10.1063/1.1699114.

[38] A. Morgado, C. Dodaro, and J. Marques-Silva. "Core-Guided MaxSAT with Soft Cardinality Constraints". In: *Principles and Practice of Constraint Programming*. Ed. by B. O'Sullivan. Cham: Springer International Publishing, 2014, pp. 564–573. ISBN: 978-3-319-10428-7.

[39] C. Papadimitriou. *Computational Complexity*. Theoretical computer science. Addison-Wesley, 1994. ISBN: 9780201530827.

[40] M. Poloczek. "Bounds on Greedy Algorithms for MAX SAT". In: *Algorithms - ESA 2011*. Algorithms - ESA 2011. Springer Berlin Heidelberg, 2011, pp. 37–48. DOI: 10.1007/978-3-642-23719-5_4.

[41] M. Poloczek and G. Schnitger. "Randomized Variants of Johnson's Algorithm for MAX SAT". In: *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*. Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms. Society for Industrial and Applied Mathematics, 2011, pp. 656–663. DOI: 10.1137/1.9781611973082.51.

[42] M. Poloczek and D. P. Williamson. "An Experimental Evaluation of Fast Approximation Algorithms for the Maximum Satisfiability Problem". In: *Experimental Algorithms: 15th International Symposium, SEA 2016, St. Petersburg, Russia, June 5-8, 2016, Proceedings*. Ed. by A. V. Goldberg and A. S. Kulikov. Cham: Springer International Publishing, 2016,

pp. 246–261. ISBN: 978-3-319-38851-9. DOI: 10.1007/978-3-319-38851-9_17.

[43] M. Poloczek, D. P. Williamson, and A. van Zuylen. "On Some Recent Approximation Algorithms for MAX SAT". In: *LATIN 2014: Theoretical Informatics: 11th Latin American Symposium, Montevideo, Uruguay, March 31–April 4, 2014. Proceedings*. Ed. by A. Pardo and A. Viola. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 598–609. ISBN: 978-3-642-54423-1. DOI: 10.1007/978-3-642-54423-1_52.

[44] M. Poloczek et al. "Greedy Algorithms for the Maximum Satisfiability Problem: Simple Algorithms and Inapproximability Bounds". In: *SIAM Journal on Computing* 46.3 (2017), pp. 1029–1061. DOI: 10.1137/15m1053369.

[45] W. Spears. "Simulated annealing for hard satisfiability problems". In: *Cliques, Coloring, and Satisfiability*. Cliques, Coloring, and Satisfiability. American Mathematical Society, 1996, pp. 533–557. DOI: 10.1090/dimacs/026/26.

[46] *Theory and Applications of Satisfiability Testing – SAT 2017*. Lecture Notes in Computer Science. Springer International Publishing, 2017. DOI: 10.1007/978-3-319-66263-3.

[47] I. Wegener. *Complexity Theory*. Springer-Verlag, 2005. DOI: 10.1007/3-540-27477-4.

[48] D. P. Williamson and D. B. Shmoys. *The Design of Approximation Algorithms*. 1st. New York, NY, USA: Cambridge University Press, 2011. ISBN: 0521195276, 9780521195270.

[49] M. Yannakakis. "On the Approximation of Maximum Satisfiability". In: *Journal of Algorithms* 17.3 (1994), pp. 475–502. DOI: 10.1006/jagm.1994.1045.

[50] Y. Zhang et al. "Protein Interaction Inference as a MAX-SAT Problem". In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Workshops*. 2005. DOI: 10.1109/cvpr.2005.515.

A Results per Category

	cat	annealing	greedy	greedy-s	slack	n
1	abrame-habet/max2sat/120v	0.832	0.819	0.819	0.766	45
2	abrame-habet/max2sat/140v	0.844	0.830	0.830	0.774	45
3	abrame-habet/max2sat/160v	0.852	0.838	0.838	0.778	45
4	abrame-habet/max2sat/180v	0.865	0.849	0.849	0.789	44
5	abrame-habet/max2sat/200v	0.882	0.864	0.864	0.799	49
6	abrame-habet/max3sat/110v	0.972	0.954	0.952	0.909	50
7	abrame-habet/max3sat/70v	0.948	0.935	0.934	0.896	45
8	abrame-habet/max3sat/90v	0.959	0.943	0.940	0.902	49
9	highgirth/3sat	0.991	0.960	0.954	0.915	50
10	highgirth/4sat	0.995	0.982	0.978	0.955	32

Table 11: Mean performance ratios for `ms_random`

	cat	annealing	greedy	greedy-s	slack	n
1	bipartite/maxcut-140-630-0.7	0.862	0.837	0.837	0.780	50
2	bipartite/maxcut-140-630-0.8	0.863	0.836	0.836	0.781	50
3	maxcut/abrame-habet/v140	0.842	0.820	0.820	0.774	45
4	maxcut/abrame-habet/v160	0.849	0.824	0.824	0.776	45
5	maxcut/abrame-habet/v180	0.856	0.832	0.832	0.778	45
6	maxcut/abrame-habet/v200	0.863	0.838	0.838	0.781	45
7	maxcut/abrame-habet/v220	0.868	0.840	0.840	0.787	45
8	maxcut/dimacs-mod	0.802	0.793	0.793	0.767	62
9	maxcut/spinglass	0.894	0.863	0.863	0.776	5
10	set-covering/scpclr	0.981	0.972	0.934	0.934	4
11	set-covering/scpcyc	0.873	0.849	0.776	0.788	6

Table 12: Mean performance ratios for `ms_crafted`

	cat	annealing	greedy	greedy-s	slack	n
1	circuit-debugging-problems	0.810	0.977	0.976	0.913	3
2	sean-safarpour	0.828	0.978	0.980	0.916	52

Table 13: Mean performance ratios for `ms_industrial`

	cat	annealing	greedy	greedy-s	slack	n
1	aes-key-recovery	0.988	0.999	0.999	0.971	35
2	aes	0.953	0.982	0.985	0.985	7
3	atcoss-mesat	0.874	0.994	0.992	0.977	18
4	atcoss-sugar	0.913	0.986	0.988	0.955	19
5	bcp-fir	0.956	0.875	0.876	0.967	32
6	bcp-hipp	0.998	0.998	0.998	0.997	35
7	bcp-msp	0.943	0.931	0.924	0.910	35
8	bcp-mtg	0.986	0.975	0.974	0.901	30
9	bcp-syn	0.928	0.872	0.854	0.853	35
10	circuit-debugging	0.810	0.977	0.976	0.913	3
11	circuit-trace	0.879	0.979	0.984	0.916	4
12	close-solutions	0.857	0.962	0.968	0.933	35
13	des	0.781	0.942	0.954	0.892	35
14	extension-enforcement	0.853	0.904	0.886	0.899	35
15	fault-diagnosis	0.768	0.995	0.995	0.956	35
16	frb	0.975	0.974	0.974	0.974	25
17	gen-hyper-tw	0.883	0.993	0.992	0.982	35
18	haplotype-assembly	0.900	0.875	0.875	0.900	6
19	hs-timetabling	0.891	0.942	0.943	0.926	2
20	job-shop	0.914	0.993	0.993	0.901	3
21	kbtree	0.789	0.784	0.788	0.770	24
22	maxclique	0.954	0.946	0.946	0.946	35
23	maxcut	0.821	0.811	0.811	0.774	30
24	maxone	0.922	0.902	0.896	0.860	25
25	mbd	0.974	0.961	0.962	0.917	35
26	min-fill	0.946	0.976	0.979	0.967	27
27	packup	0.977	0.966	0.966	0.953	35
28	pbo-mqc-nencdr	0.986	0.976	0.976	0.934	25
29	pbo-routing	0.989	0.985	0.986	0.910	15
30	protein-ins	0.998	1.000	1.000	1.000	12
31	reversi	0.976	0.995	0.994	0.941	35
32	scheduling	0.861	0.996	0.996	0.969	5
33	sean-safarpour	0.829	0.977	0.978	0.915	35
34	set-covering	0.916	0.898	0.839	0.848	10
35	tpr-multiple-path	0.887	0.991	0.990	0.909	35
36	treewidth-computation	0.968	0.999	1.000	0.944	33

Table 14: Mean performance ratios for `mse17`