Constraints (2013) 18:478–534
DOI 10.1007/s10601-013-9146-2

SURVEY

Iterative and core-guided MaxSAT solving: A survey and assessment

Antonio Morgado · Federico Heras · Mark Liffiton · Jordi Planes · Joao Marques-Silva

Published online: 20 July 2013

Abstract *Maximum Satisfiability* (MaxSAT) is an optimization version of SAT, and many real world applications can be naturally encoded as such. Solving MaxSAT is an important problem from both a theoretical and a practical point of view. In recent years, there has been considerable interest in developing efficient algorithms and several families of algorithms have been proposed. This paper overviews recent approaches to handle MaxSAT and presents a survey of MaxSAT algorithms based on iteratively calling a SAT solver which are particularly effective to solve problems arising in industrial settings. First, classic algorithms based on iteratively calling a SAT solver and updating a *bound* are overviewed. Such algorithms are referred to as *iterative* MaxSAT algorithms. Then, more sophisticated algorithms that additionally take advantage of *unsatisfiable cores* are described, which are referred to as *core-guided* MaxSAT algorithms. Core-guided MaxSAT algorithms use the information provided by unsatisfiable cores to *relax clauses on demand* and to create simpler constraints. Finally, a comprehensive empirical study on non-random benchmarks is

A. Morgado (✉) · F. Heras · J. Marques-Silva
CSI/CASL, University College Dublin, Dublin, Ireland
e-mail: ajrm@ucd.ie

F. Heras
e-mail: fheras@ucd.ie

M. Liffiton
Illinois Wesleyan University, Bloomington, IL, USA
e-mail: mliffito@iwu.edu

J. Planes
Universitat de Lleida, Lleida, Spain
e-mail: jplanes@diei.udl.cat

J. Marques-Silva
IST/INESC-D, Universidade Tecnica de Lisboa, Lisboa, Portugal
e-mail: jpms@ucd.ie

conducted, including not only the surveyed algorithms, but also other state-of-the-art MaxSAT solvers. The results indicate that (i) core-guided MaxSAT algorithms in general abort in less instances than classic solvers based on iteratively calling a SAT solver and that (ii) core-guided MaxSAT algorithms are fairly competitive compared to other approaches.

Keywords MaxSAT · MaxSMT · Boolean optimization · Optimization problems

1 Introduction

The *Satisfiability* problem in propositional logic (SAT) is the task of deciding whether a given propositional formula has a model. *MaxSAT* is an optimization variant of SAT and it can be seen as a generalization of the SAT problem. Given a propositional formula in conjunctive normal form (*CNF*), a conjunction of disjunctions (*clauses*), the objective of the *MaxSAT* problem is to find an *assignment* for the Boolean variables that maximizes the number of satisfied clauses.

In *weighted MaxSAT*, each clause has an associated *weight* and the goal becomes maximizing the sum of the weights of the satisfied clauses. In many problems originating from real world domains, a subset of the clauses must be satisfied (referred to as *hard* clauses), and the remaining (referred to as *soft*) clauses may be satisfied or not. Weights are usually modeled as natural numbers, and hard clauses are modeled by giving each a sufficiently large weight [31].

There are several variants of the MaxSAT problem [31] depending on the distribution of soft and hard clauses. When all clauses are soft and their weight is 1, the problem is referred to as unweighted MaxSAT. When all clauses are soft and some weights are greater than 1, it is referred to as weighted MaxSAT. When all soft clauses have weight 1 and there is a set of hard clauses, it is referred to as *partial MaxSAT*. Finally, when some soft clauses have a weight greater than 1 and there is a set of hard clauses, it is referred to as *weighted partial MaxSAT*. For all MaxSAT variants, this paper considers the *cost* of a given truth assignment (whether an optimum solution or not) to be the sum of the weights of the clauses *not* satisfied by that assignment. The goal of every variant is thus to find an assignment with *minimum* cost.

The remainder of this section is organized as follows. First, example applications of MaxSAT are outlined, followed by an overview of the existing MaxSAT algorithms in the literature. Then, a general description of the MaxSAT algorithms surveyed in this paper is given. The section concludes with the goals and structure of the survey.

1.1 MaxSAT applications

Many important problems can be naturally expressed as MaxSAT. These include academic problems such as *Max-Cut* or *Max-Clique*, as well as problems from many industrial domains. Concrete examples include the following domains: *Routing* problems [127]; different problems of *BioInformatics*, such as *Protein Alignment* [120], *Haplotyping with Pedigrees* [56], *Reasoning over Biological Networks*[58]; *Hardware Debugging*, both on *Design Debugging* [111], as well as on *Circuit Debugging* [34, 81];

Software Debugging (of C code) [66, 67]; *Scheduling* [124]; *Planning* [38, 68, 108, 129]; *Course Timetabling* [17, 18]; *Probabilistic Reasoning* [102]; *Electronic Markets* [112]; *Credential-Based interactions* as a way to minimize the disclosure of private information [12]; *Enumeration of MUSes/MCSes* [27, 78, 107]; *Software Package Upgrades* [13, 15, 16, 80, 123]; *Combinatorial Auctions* [60]; *Quantified Boolean Formulas* [30].

Additionally, MaxSAT algorithms have also been successfully applied as a way to compute the *Binate/Unate Covering* problem [59], where it has been applied to *Haplotype Inference* [57], to *Digital Filter Design* [1], to *FSM Synthesis and Logic Minimization* [59], among others. Analogously, many problems originally formulated in other optimization frameworks can be easily reformulated as MaxSAT including the *Pseudo-Boolean Optimization* framework [3], the *Weighted CSP* framework [73] and the MaxSMT framework [96].

1.2 MaxSAT algorithms

The last two decades have witnessed significant progress in the development of theoretical and practical work on MaxSAT. Early theoretical MaxSAT research provided insights in the complexity of the problem [21, 100].

Early practical works on MaxSAT were based on *stochastic local search* (*SLS*) [65, 115, 116] with the objective of approximating the MaxSAT solution. SLS algorithms randomly compute an initial assignment of the variables, and at each iteration the value of one variable is *flipped* (from *true* to *false* or vice-versa) in a process that attempts to find an assignment satisfying more clauses than the largest number found thus far. SLS algorithms do not guarantee to find the optimum solution and for this reason are referred to as *incomplete algorithms*. Whereas the mentioned SLS algorithms were initially developed for the SAT problem, they can be directly applied to approximate (unweighted) MaxSAT and for the most general Weighted Partial MaxSAT [31]. For example, the most successful SLS algorithms for SAT have been extended for approximating MaxSAT in the Ubcsat system [122]. SLS has been shown to be among the most effective algorithms to solve *randomly* generated problems, but recent work on *semidefinite programming* has been shown to be quite promising for approximating random Max2Sat instances [5, 54]. A recent survey on SLS algorithms can be found in [64].

In the last decade, different *complete (or exact) algorithms* have been proposed for solving MaxSAT to optimality. Some of the existing approaches are based on *reducing* the MaxSAT problem into a well-known optimization problem and then use an off-the-shelf solver for such problem. For example, a natural approach to solve the MaxSAT problem is to model it as a *Integer Linear Program* (*ILP*). The ILP problem can then be solved directly by a dedicated solver such as CPLEX. A ILP problem restricted to 0-1 inequalities (*pseudo-Boolean* constraints) is usually referred to as *Pseudo-Boolean Optimization* problem [23, 110]. Several approaches have been developed to handle the PBO problem. Most of the existing PBO algorithms, either adapt a modern SAT solver to natively handle pseudo-Boolean constraints [3, 32, 117] or directly solve a sequence of SAT problems [45]. In both cases, the aim is to take advantage of modern SAT solvers with powerful *clause learning* and *backjumping techniques* [44, 88, 93].

Another example of reducing MaxSAT includes *Answer Set Programming* (*ASP*) [50], in particular its optimization version *MaxASP* [48]. Recent algorithms for

MaxASP include a branch and bound approach [98] and the adapted versions of some core-guided MaxSAT algorithms presented in this survey [4]. The last relevant reduction include the *Weighted Constraint Satisfaction Problem* (*WCSP*), e.g. [42]. Current approaches for solving WCSP are based on branch and bound algorithms which apply *local consistency* to boost the search [39], as well as stochastic local search approaches [94].

Besides reductions, other approaches for MaxSAT are based on adapting a modern SAT solver by either (i) forcing an order on the selection of variables during the search, that by construction leads to the optimum solution [51, 52, 109], or (ii) by integrating *knowledge compilation*-based lower bounds and exploiting them during the search [103, 106].

A large family of contemporary exact MaxSAT solvers follow a *branch and bound* (*BB*) algorithm [29, 41, 61, 72, 76, 99, 126]. BB algorithms *assign* a variable at each node of a *search tree*, simplify the current formula and compute *lower bounds* and *upper bounds* on the value of any assignment that may be found below that node. Whenever the lower bound matches the upper bound, no solution better than the current one can be found in that branch, and the algorithm can *backtrack*. The branch and bound MaxSAT algorithm proposed in [29] is organized in two phases. In the first phase, a SLS algorithm is used to compute an initial upper bound. Then, in a second phase, the actual branch and bound takes place in which primitive lower bounds and *simplification rules* were applied at each of the nodes of the search tree. Later works added more effective techniques in order to boost the search. Namely, more efficient data-structures, new branching heuristics, new simplification rules and more accurate lower bounds [95, 118, 121, 126]. Modern BB solvers additionally exploit unit propagation [75] to compute powerful lower bounds, as well as new inference rules [61, 72, 76] based on the resolution rule for MaxSAT [28, 71, 72]. Those algorithms have been recently surveyed in [74] and are particularly effective on *random* and *crafted* benchmarks as consistently demonstrated by the MaxSAT Evaluations since 2008 [14].

Another large family of MaxSAT algorithms is based on iteratively calling a SAT solver. This paper presents a survey on those algorithms which are characterized by using a SAT solver to iteratively search for satisfiable subsets of a formula's clauses within cardinality bounds determined differently by each algorithm. Those algorithms are particularly suitable for benchmarks coming from industrial settings, as observed in past MaxSAT Evaluations [14]. For example, in the 2012 MaxSAT Evaluation, all the three top places of industrial categories are occupied by algorithms that are based on iteratively calling a SAT solver, as well as for partial crafted and weighted partial benchmarks. The next subsection briefly introduces this class of MaxSAT algorithms.

1.3 Iterative and core-guided MaxSAT algorithms

Iterative and core-guided MaxSAT Algorithms operate by instrumenting a CNF formula with fresh Boolean variables (called the *relaxation variables*) added to the soft clauses. The soft clauses in a formula relaxed in this way can then be "disabled" with a temporary assignment to their relaxation variables, effectively allowing a SAT solver to search through the space of *clause subsets* to find satisfiable subsets, given that the complete formula is unsatisfiable. Each temporary assignment is associated to a cost represented by the sum of weights of the relaxed soft clauses whose relaxation variable(s) is assigned True, which implicitly defines a satisfying clause subset.

Additionally, these algorithms utilize *cardinality constraints* [45, 110], *pseudo-Boolean constraints* [45, 110] that place a bound k on a set of variables, allowing the sum of weights associated to True variables to be at most k. These are often called *at most constraints* (abbreviated in this paper as AtMostK). The MaxSAT algorithms described in this section use AtMostK constraints over sets of relaxation variables in order to place bounds on the cost of the assignments discovered by the SAT solver. This way, each algorithm can search through a sequence of satisfiable clause subset until the largest such size is determined, yielding the desired, optimum solution with lowest cost. The differences in these algorithms lie primarily in the selection of clauses to relax and in the progression of bounds each uses during its search.

Early theoretical works (e.g. [55, 100, 101]) introduced *binary search* in order to characterize the complexity of the MaxSAT problem and other optimization problems. The idea of the algorithm is to initially *relax all* soft clauses and use AtMostK constraints to refine a lower and an upper *bound* on the solution cost. Similarly, *linear search* can be applied in order to refine only one bound. As such, algorithms based on linear search can be of one of two main variants: those that refine an *upper bound* on the cost of the optimum solution, and those that refine a *lower bound*.

Algorithms that iteratively refine upper bounds will be referred to as *linear search Sat-Unsat* (LIN-SU), denoting that all calls to the SAT solver but the last will declare the given formula as *satisfiable*. Several MaxSAT tools are available that follow a linear search Sat-Unsat strategy, for example SAT4J [25] and QMAXSAT [69]. Note that using a SAT solver following linear Sat-Unsat scheme has also been used in other domains, for example in the PBO solvers PBS [2], MINISAT+ [45] and SAT4J [25].

Algorithms that iteratively refine lower bounds on the solution cost will be referred to as *linear search Unsat-Sat* (LIN-US), denoting that all calls to a SAT solver but the last will return *unsatisfiable*. There are no known implementations of the linear search Unsat-Sat algorithm specifically for MaxSAT, though CAMUS [77], which computes *minimal unsatisfiable subsets* (*MUS*) of an input formula, does use such an approach in its first phase, solving MaxSAT as a side-effect of its primary goal.

In order to apply a *binary search* [47] strategy, both lower and upper bounds are maintained. Essentially, the binary search algorithm computes the midpoint ν between the upper and lower bounds and calls a SAT solver to test whether there exists a solution whose cost is less than or equal to ν. If the SAT solver returns satisfiable, the upper bound can be updated to the value of the model found. Alternatively, if the solver returns unsatisfiable, the lower bound can be updated to the middle value just tested. Observe that, in the worst case, binary search requires a number of calls to the SAT solver linear in the problem size (in fact the number of soft clauses), whereas linear search may require an exponential number of calls due to the clause weights. A MaxSAT solver that follows the binary search scheme was introduced in [47]. Also, one of the algorithms introduced in [69] alternates binary search and linear search Sat-Unsat at each iteration. In [35] binary search was used to solve an optimization version of the SMT framework. *Bit-based search* [37, 53] aims to find the optimum solution by exploring its binary representation and also requires a linear number of calls to a SAT solver. A *prolog* implementation was proposed in [37] for bit-based search.

The algorithms described so far require the SAT solver to report the *satisfiable* (SAT) or *unsatisfiable* (UNSAT) outcomes and to provide *models* on SAT outcomes. Additionally, such algorithms *relax all soft clauses* before calling the SAT solver for the first time. Those algorithms are referred to as *iterative MaxSAT algorithms*.

More recent algorithms based on iteratively calling a SAT solver take advantage of the information provided by *unsatisfiable cores* [128] to guide the search. Such algorithms are referred to as *core-guided* MaxSAT algorithms. Hence, core-guided MaxSAT additionally requires the SAT solver to be able to produce unsatisfiable cores on UNSAT outcomes. In particular, unsatisfiable cores are used to relax soft clauses *on demand* and/or to create constraints which are *shorter* in the sense that they involve a smaller set of relaxation variables. Existing core-guided MaxSAT algorithms follow an algorithmic scheme similar to linear or binary search.

The seminal core-guided algorithm was introduced in [47] (referred in this paper as MSU1), being restricted to unweighted partial MaxSAT. At each iteration, a relaxation variable is added to each soft clause involved in a newly extracted unsatisfiable core, and a new constraint is added to the formula. Several improvements were introduced later in MSU1.1 [86], and MSU1.2 [85]. The extension to the weighted case was proposed concurrently in [82] and [8] (respectively, WMSU1 and WPM1). Both WMSU1 and WPM1 [47] algorithms may require *more than one* relaxation variable per soft clause and use AtMost1 constraints (instead of general AtMostK constraints).

More recent algorithms relax soft clauses *on demand*, add at most one relaxation variable per soft clause, and use general AtMostK constraints. MSU3 was the first algorithm to follow such organization, and it is based on refining a lower bound. Similarly, PM2 and PM2.1 also refine a lower bound and add additional constraints to the formula based on a heuristic that counts how many previous unsatisfiable cores are *contained* at each new unsatisfiable core. PM2.1 is the first algorithm to take advantage of *disjoint cores* (or *covers*) to add smaller constraints to the formula. Essentially, a disjoint core is an unsatisfiable core that does not share any soft clause with previous unsatisfiable cores. A weighted version of PM2.1 can be found in WPM2 [9]. WPM2 introduced a technique based on the *subset sum* problem to refine the lower bound at each iteration. A simplified version of such technique is borrowed in this paper to extend several iterative and core-guided MaxSAT algorithms, which were originally presented in their unweighted version, to handle weighted clauses.

MSU4 [87] alternates satisfiable and unsatisfiable calls to the SAT solver and also relaxes soft clauses on demand. *Core-guided binary search* [63] relaxes soft clauses on demand and follows a binary search strategy. Finally, *core-guided binary search with disjoint cores* [63] enhances the previous algorithm by maintaining disjoint cores. Note that WMSU4 and both versions of core-guided binary search refine both a lower bound and an upper bound.

Figure 1 presents algorithms and publications in chronological order along with their citation graph. These references are for papers reporting on implementations of iterative and core-guided algorithms examined in this work as well as more recent

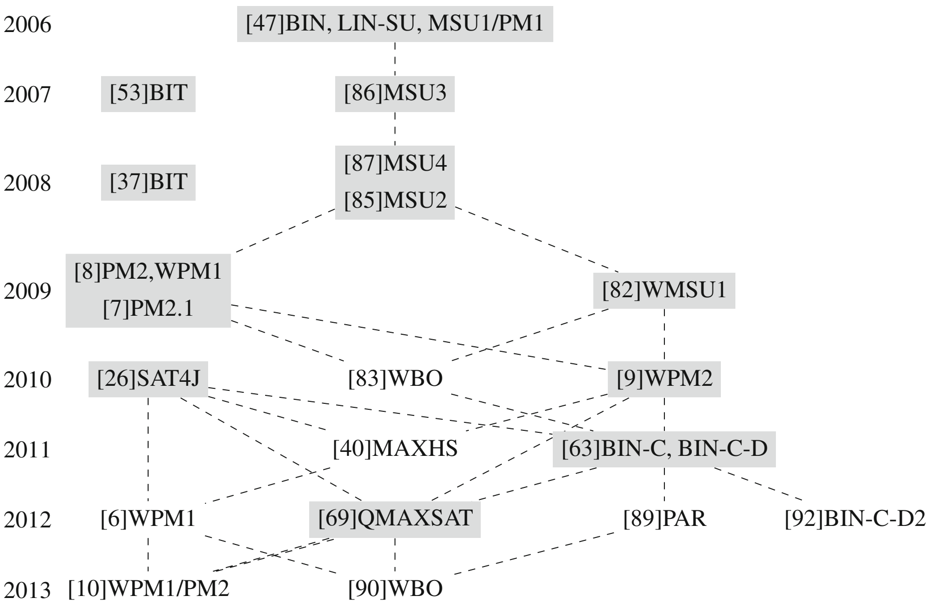

Fig. 1 Chronology and citation map. Algorithms covered in this survey are shaded in *grey*. *Dashed line* paths indicate the citation graph within this set of papers

works not covered by this survey. In the figure, algorithms covered by this work have been shaded in grey. The names of the algorithms are as they have been introduced in the respective papers, except for the algorithm of [47]. Additionally, Table 1 presents all the iterative and core-guided MaxSAT algorithms as studied in the paper, together with the references to where the algorithms were published and a reference to where the respective pseudo-code can be found in the paper. The names of the algorithms in the table are as used in this survey.

Note that several works have been proposed to handle other optimization problems using similar or adapted versions of the iterative and core-guided MaxSAT algorithms presented so far. As mentioned before, linear search Sat-Unsat has been widely used for PBO and an adapted version of MSU1 and MSU3 for MaxASP [4]. Additionally, several works have been recently presented to handle an optimization version of the Satisfiability Modulo Theories (SMT) framework, referred in this paper as MaxSMT. Such works include [22, 36, 97, 114] to name a few. In this paper, it is explained how iterative and core-guided MaxSAT algorithms can be easily adapted to handle the MaxSMT problem.

Contemporary MaxSAT algorithms are based not only on iteratively calling a SAT solver to retrieve models or unsatisfiable cores, but also require the integration of additional sophisticated techniques. Such approaches include an algorithm [40] that uses linear programming technology to handle the cardinality and pseudo-Boolean constraints, more aggressive computation of lower bounds and upper bounds at each disjoint core for the core-guided binary search with disjoint cores algorithm [92], and the integration of several techniques in WMSU1, including Boolean

Table 1 Iterative and core-guided MaxSAT algorithms studied

Iterative MaxSAT algorithms	
Linear Search Unsat-Sat: `LIN-US`	Algorithm 2 (page 13)
Linear Search Sat-Unsat: `LIN-SU` [25, 47, 69]	Algorithm 4 (page 16)
Binary Search: `BIN` [47, 69]	Algorithm 5 (page 17)
Alternating Binary Search with Linear Search Sat-Unsat: `BIN/LIN-SU` [69]	Algorithm 6 (page 18)
Bit-Based Search: `BIT` [37, 53]	Algorithm 7 (page 19)
Core-GuidedMaxSAT algorithms	
`WMSU1/WPM1` [8, 47, 82]	Algorithm 8 (page 21)
`MSU2` [85]	Algorithm 9 (page 24)
`WMSU3` [86]	Algorithm 10 (page 26)
`WMSU4` [87]	Algorithm 11 (page 26)
`PM2` [8]	Algorithm 12 (page 27)
`PM2.1` [7]	Algorithm 13 (page 28)
`WPM2` [9]	Algorithm 14 (page 30)
Core-Guided Binary Search: `BIN-C` [63]	Algorithm 15 (page 31)
Core-Guided Binary Search with Disjoint Cores: `BIN-C-D` [63]	Algorithm 16 (page 34)

multilevel optimization [84] , MaxSAT resolution [62], breaking symmetries [6], and partitioning soft clauses [89].

1.4 Goals and structure

The main goal of this paper is to present a survey of iterative and core-guided MaxSAT algorithms as of 2012. Some of the algorithms were originally introduced in their unweighted version. In this paper, these algorithms are extended to handle weighted partial MaxSAT. In particular, a total of 14 algorithms are described in detail and characterized in terms of different properties. Such algorithms were implemented in the same software platform so that they could be fairly compared in order to better understand which are the most competitive algorithms independently of implementation details. A comprehensive empirical study was conducted including not only the algorithms described in this paper but also other state-of-the-art MaxSAT solvers. The results on non-random benchmarks from MaxSAT Evaluations indicate that (i) core-guided MaxSAT algorithms in general abort in fewer instances than classic algorithms based on iteratively calling a SAT solver and that (ii) core-guided MaxSAT algorithms are fairly competitive compared to the other approaches.

The paper is structured as follows. Section 2 formally defines the MaxSAT problem and related notation, as well as preliminary notation that will be used when describing MaxSAT algorithms (Section 2.3). Section 3 briefly surveys Cardinality and Pseudo-Boolean encodings used by the MaxSAT algorithms based on calling a SAT solver in order to transform the specific constraints into a set of (hard) clauses. *Iterative* MaxSAT algorithms are introduced in Section 4 and *core-guided* algorithms in Section 5. Section 6 presents the MaxSMT problem, which can be seen as a generalization of the MaxSAT problem, and shows how current SMT solvers can be easily turned into MaxSMT solvers implementing any of the algorithms described in this survey. The experimental investigation is shown in Section 7. Finally, Section 8 concludes the paper.

2 Preliminaries

This section presents the necessary definitions and notation related to the SAT and MaxSAT problems, as well as the notation used for describing the MaxSAT algorithms in the remainder of the paper.

2.1 Boolean satisfiability (SAT)

Let $X = \{x_1, x_2, \ldots, x_n\}$ be a set of Boolean variables. A *literal* is either a variable x_i or its negation $\neg x_i$. The variable to which a literal l refers is denoted by $var(l)$. Given a literal l, its negation $\neg l$ is $\neg x_i$ if l is x_i and it is x_i if l is $\neg x_i$. A *clause* c is a disjunction of literals. Hereafter, lower case letters from the start of the alphabet will represent clauses. The *size* of a clause, noted $|c|$, is the number of literals it contains. A formula in *conjunctive normal form* (CNF) φ is a set of clauses. An *assignment* is a set of literals $\mathcal{A} = \{l_1, l_2, \ldots, l_n\}$ such that for all $l_i \in \mathcal{A}$, its variable $var(l_i) = x_i$ is assigned to a value (*true* or *false*). If variable x_i is assigned to *true*, literal x_i is *satisfied* and literal $\neg x_i$ is *unsatisfied* (or *falsified*). Similarly, if variable x_i is assigned to *false*, literal $\neg x_i$ is satisfied and literal x_i is unsatisfied. If all variables in X are assigned, the assignment is called *complete*, otherwise it is called *partial*. An assignment *satisfies* a literal iff it belongs to the assignment, it satisfies a clause iff it satisfies one or more of its literals and it *unsatisfies* a clause iff it contains the negation of all its literals.

A *model* is a complete assignment that satisfies all the clauses in a CNF formula φ. SAT is the problem of deciding whether there exists a model for a given propositional formula. Given an unsatisfiable SAT formula φ, a subset of clauses φ_C whose conjunction is still unsatisfiable is called an *unsatisfiable core* of the original formula. Given an unsatisfiable formula, modern SAT solvers can be instructed to generate an unsatisfiable core [128].

2.2 Maximum satisfiability (MaxSAT)

A *weighted* clause is a pair (c_i, w_i), where c_i is a clause and w_i is the cost of falsifying it, also called its *weight*. Many real problems contain clauses that *must* be satisfied. Such clauses are called *mandatory* (or *hard*) and are associated with a special weight $\top$. Note that any weight $w_i \geq \top$ indicates that the associated clause must be necessarily satisfied. Thus, w_i can be replaced by $\top$ without changing the problem. Consequently, all weights take values in $\{0, \ldots, \top\}$. Non-mandatory clauses are also called *soft* clauses. A formula in *weighted conjunctive normal form* (WCNF) $\varphi = \varphi^H \cup \varphi^S$ is a set of weighted clauses where φ^H is the set of *hard* clauses, and φ^S is the set of *soft* clauses.

A *model* is a complete assignment $\mathcal{A}$ that satisfies all mandatory clauses. The *cost of a model* is the sum of weights of the soft clauses that it falsifies. Given a WCNF formula $\varphi = \varphi^H \cup \varphi^S$, *Weighted Partial* MaxSAT is the problem of finding a model of minimum cost. In other words, the objective is to find an assignment that satisfies all hard clauses in φ^H and minimizes the sum of weights of unsatisfied soft clauses

Function RelaxCls(R, φ, ψ)

```
Input: $(R, \varphi, \psi)$ denotes the set of relaxation variables $R$, and two WCNF formulas $\varphi$ and $\psi$ satisfying $\psi \subseteq \varphi$
begin
    $(R_o, \varphi_o) \leftarrow (R, \varphi)$    /* $R_o$ set of relaxation variables; $\varphi_o$ set of clauses to return */
    foreach $(c, \omega) \in$ Soft $(\psi)$ do
        $R_o \leftarrow R_o \cup \{r\}$    /* $r$ is a fresh relaxation variable */
        $c_R \leftarrow c \cup \{r\}$
        $\varphi_o \leftarrow \varphi_o \setminus \{(c, \omega)\} \cup \{(c_R, \omega)\}$    /* $(c_R, \omega)$ is tagged soft */
    end
    return $(R_o, \varphi_o)$
end
```

in φ^S. If the set of hard clauses is empty ($\varphi^H = \emptyset$), the problem is called *weighted MaxSAT* problem. When all the weights are equal to 1 ($\forall_i : w_i = 1$), then the problem is called *partial MaxSAT* problem. Finally, if both the set of hard clauses is empty ($\varphi^H = \emptyset$), and all the weights are equal to 1 ($\forall_i : w_i = 1$), then the problem is called (unweighted) *MaxSAT* problem.

2.3 Describing MaxSAT algorithms

The algorithms described in this paper are based on *iteratively* calling a SAT solver. At each iteration, the algorithm creates a CNF instance and invokes a SAT solver on it. Depending on the result given by the SAT solver, the algorithm updates the CNF formula accordingly and starts a new iteration. Two types of algorithms are considered in this paper: *Iterative* MaxSAT algorithms that on each iteration update a *bound* depending on the result returned by the SAT solver (satisfiable or unsatisfiable) and *core-guided* MaxSAT algorithms that additionally exploit the information in the *unsatisfiable cores* returned by the SAT solver on unsatisfiable outcomes.

All the algorithms make use of *relaxation variables*. Relaxation variables are new Boolean variables that are added to the soft clauses of the formula. Unless otherwise indicated, the algorithms associate at most one unique relaxation variable with each soft clause. In terms of notation, relaxation variables are maintained in a set R, and the relaxation variable r_i is associated to the clause c_i with weight w_i where $1 \leq i \leq m$. This paper assumes that any weighted formula φ has m soft clauses.

In order to add relaxation variables to soft clauses, the algorithms use function *RelaxCls*(R, φ, ψ), which receives a set of relaxation variables R, a WCNF formula φ, and a set of soft clauses ψ. It returns the pair (R_o, φ_o), where φ_o corresponds to a copy of φ whose soft clauses included in ψ have been augmented with fresh relaxation variables, and R_o corresponds to R augmented with the relaxation variables added in φ_o. Function RelaxCls show the pseudo-code of *RelaxCls*(R, φ, ψ).

Given the set of relaxation variables R, the algorithms add *cardinality/pseudo-Boolean constraints* [47] and translate them to hard clauses. Such constraints usually state that the sum of the weights of the relaxed clauses is bounded above (`AtMostK` constraint with $\sum_{i=1}^{m} w_i r_i \leq K$) or bounded below (`AtLeastK` constraint with $\sum_{i=1}^{m} w_i r_i \geq K$) by a specific value K.

Opt(φ) will refer to the optimum solution of instance φ. The algorithms may use the following variables: λ for a lower bound, μ for an upper bound, ν for the value

Algorithm 1 MaxSAT Wrapper Algorithm

```
Input: $(\varphi, MSAlgorithm, UBHeuristic, LBHeuristic)$
1 $st \leftarrow$ SAT(Hard($\varphi$))
2 if $st = UNSAT$ then return $\emptyset$                              /* No Solution */
3 if $UBHeuristic = NIL \wedge LBHeuristic = NIL$ then return $MSAlgorithm(\varphi)$
4 $(\mathcal{A}, \mu) \leftarrow UBHeuristic(\varphi)$
5 $(\lambda, \varphi_{cores}) \leftarrow LBHeuristic(\varphi)$
6 return $MSAlgorithm(\varphi, \mathcal{A}, \mu, \lambda, \varphi_{cores})$
```

the algorithm is searching (in the current iteration), and ι for one bit of the binary representation of that value. The algorithms also make use of the following functions:

- `Soft`(φ) returns the set of all *soft* clauses in φ.
- `hard`(φ) returns the set of all *hard* clauses in φ.
- $SAT(\varphi)$ makes a call to the SAT solver which returns whether φ is satisfiable (SAT) or unsatisfiable (UNSAT).
  The SAT solver returns a complete assignment $\mathcal{A}$ if φ is satisfiable, otherwise $\mathcal{A}$ is empty. When φ is unsatisfiable, the SAT solver is able to return an unsatisfiable core φ_C.
- $CNF(c)$ returns a set of clauses that encode the pseudo-Boolean constraint c into CNF.
- $Init(\mathcal{A})$, given the assignment $\mathcal{A}$, returns the set of assignments in $\mathcal{A}$ to the initial variables of the input formula φ.

Without loss of generality, all the algorithms presented in this paper assume that the input formula is not empty and that the set of hard clauses of the input formula has a model. In practice, it is expected that some *wrapper* checks these assumptions before calling the actual MaxSAT algorithm. The pseudo-code for such a wrapper can be found in Algorithm 1. The parameters of the wrapper are a WCNF formula (φ), the algorithm to execute (*MSAlgorithm*) and the upper bound and lower bound heuristic to compute (*LBHeuristic* and *UBHeuristic*). Initially, a SAT solver is called with the hard clauses of the formula (line 1). If the SAT solver returns unsatisfiable, it means that the problem has no solution (line 2). Otherwise, the specific MaxSAT solver is called in line 3. If an upper bound or a lower bound are used, these are computed in lines 4 and 5, and the specified MaxSAT algorithm is called with the computed bounds in line 6. The MaxSAT algorithms are described in Sections 4 and 5. Lower and upper bound heuristics are described in detail in [63] and briefly reviewed below.

First, the upper bound μ is described. Each soft clause in φ is extended with a new relaxation variable obtaining a new formula φ'. Then, a SAT solver is called with the resulting formula φ' and no additional constraints. Clearly, a satisfying assignment $\mathcal{A}$ exists for φ'. Then, the sum of weights of soft clauses which are unsatisfied by $\mathcal{A}$ (disregarding relaxation variables) corresponds to a correct upper bound μ.

The lower bound λ is initially set to 0, and an initial working formula is created from the original formula. The working formula is iteratively tested for satisfiability with a SAT solver until a satisfiable instance is reached, in which case the lower bound has been computed. On each unsatisfiable outcome, an unsatisfiable core φ_C is obtained and the working formula is updated by dropping the soft clauses of φ_C.

Additionally, the minimum weight m among the soft clauses in φ_C is added to the lower bound ($\lambda = \lambda + m$).

3 Cardinality and Pseudo-Boolean constraint encodings

During the execution of any of the MaxSAT algorithms described in this paper, different constraints are added to the working formula. Such constraints need to be encoded as (hard) clauses. Depending on the type of constraint different encodings can be used. If the constraint can be represented by a clause then it is directly encoded as such. This is the case when the constraint to encode is of the form $r_1 + \ldots + r_n \geq 1$ (where r_i represents a pseudo-Boolean variable). Then the constraint is encoded as the clause $(x_1 \vee \ldots \vee x_n)$.

In general, the constraints used by the MaxSAT solvers based on iteratively calling a SAT oracle, can be represented as $\sum_{i=1}^{m} w_i \times r_i \# K$ where r_i is a relaxation variable associated to the soft clause i and w_i is its weight. The operator # can assume one of the symbols: $\leq$ for `AtMostK` constraints, $\geq$ for `AtLeastK` constraints. The value K can be equal 1 to form `AtMost1` and `AtLeast1` constraints. Otherwise, they are called `AtMostK` and `AtLeastK` constraints. If all the weights w_i are equal to 1 then the constraint is referred to as a *Cardinality constraint*. Differently, if some of the weights w_i are different than 1, the constraint is called a *Pseudo-Boolean* constraint. If the constraint represents a cardinality constraint, then some of the available encodings in the literature are one of the following encodings: the pairwise and bitwise encodings [104, 105] (but only for `AtMost1` and `AtLeast1`), the ladder encoding [49], sequential counters [119], sorting networks [24, 45], cardinality networks [19], and binary decision diagrams (BDDs) [45]. For pseudo-Boolean constraints available CNF encodings in the literature include BDDs [45], sorting networks [45] and the Warners' encoding [125] . The previous lists of encodings are not exhaustive, as cardinality and pseudo-Boolean encodings are active research fields. A survey on the subject can be found in [110]. The performance of some of the above encodings was analyzed in [91], where in general the conclusions are:

- The pairwise encoding is the best encoding for `AtMost1` and `AtLeast1` cardinality constraints.
- Cardinality networks is the best encoding for `AtMostK` and `AtLeastK` cardinality constraints (but in some very specific cases sequential counters is better).
- For pseudo-Boolean constraints BDDs is the best option but it can result in a very large set of clauses.

The encodings selected for the experimental results in this paper correspond to the best performing encodings analyzed in [91].

4 Iterative algorithms

Iterative algorithms can be classified in two; either being based on a linear search, or being based on binary search (or a conjunction of both). In the following is presented the algorithms based on linear search, and then the algorithms based on binary search. The section ends with a characterization of all the iterative algorithms.

Algorithm 2 `LIN-US`: The Linear Search Unsat-Sat Algorithm

```
Input: $\varphi$
1 $(R, \varphi_W) \leftarrow$ RelaxCls($\emptyset, \varphi,$ Soft($\varphi$))
2 $\lambda \leftarrow 0$
3 while true do
4     $(\text{st}, \mathcal{A}) \leftarrow$ SAT(Cls($\varphi_W$) $\cup$ CNF($\sum_{i=1}^{m} w_i \cdot r_i \leq \lambda$))
5     if st = $SAT$ then return Init($\mathcal{A}$)
6     $\lambda \leftarrow$ RefineBound($\{\omega_i | 1 \leq i \leq m\}, \lambda$)
7 end
```

4.1 Linear search algorithms

This section presents iterative MaxSAT algorithms that are based on linear search. Two algorithms are shown: Linear Search Unsat-Sat and Linear Search Sat-Unsat that differ on the direction in which the search is made.

A straightforward approach for MaxSAT solving using a SAT solver is to iteratively convert the optimization problem into a decision problem: Given a WCNF formula φ, the algorithm checks the satisfiability of φ together with a constraint (expressed in clauses) stating that the solution cost, i.e. the sum of weights of unsatisfied clauses, is bounded by a given K. The algorithm can start from a lower bound λ with the minimum possible cost $(K =)\lambda = 0$, meaning all clauses are satisfied, and increase it up to the optimum solution, or it can start with an upper bound μ with the maximum possible cost $(K =)\mu = \sum_{i=1}^{m} w_i$, allowing all clauses to be falsified, and decrease down to the optimum solution.

The *Linear Search Unsat-Sat* algorithm starts with $\lambda = 0$, looking for an optimum solution by checking unsatisfiable instances (all values of λ less than the optimum solution) until a satisfiable instance is identified. That first satisfiable instance must lie at the boundary between the unsatisfiable and the satisfiable, and thus it must be the optimum solution. At every step, the value λ, which is a lower bound on the optimum solution, is increased to the next possible cost (depending on the weights) until the solution is found. The pseudo-code of *Linear Search Unsat-Sat* is shown in Algorithm 2. Initially, all soft clauses are relaxed (line 1) and the lower bound is initialized to 0 (line 2). Then, the main loop (line 3) iterates while the SAT solver returns unsatisfiable (line 5). At each iteration, the SAT solver is called with the clauses of the current working formula and the encoding of the `AtMostK` constraint $\sum_{i=1}^{m} w_i \cdot r_i \leq \lambda$ (line 4). Essentially, the `AtMostK` constraint requires that, for any satisfying assignment, the sum of weights of the clauses whose relaxation variables are assigned to true is lower or equal to the lower bound λ. If the SAT solver returns unsatisfiable, the lower bound λ is increased (line 6); otherwise the algorithm terminates (line 5) and returns the satisfying assignment $\mathcal{A}$ restricted to the original variables.

Example 1 Let $\varphi = \varphi^S \cup \varphi^H$ be a partial MaxSAT instance with 6 variables, a set of 7 soft clauses φ^S, where $\varphi^S = \{(x_1, 1), (x_2, 1), (x_3, 1), (x_4, 1), (x_5, 1), (x_6, 1), (\neg x_6, 1)\}$, and a set of 5 hard clauses φ^H, where $\varphi^H = \{(\neg x_1 \vee \neg x_2, \top), (\neg x_2 \vee \neg x_3, \top), (\neg x_3 \vee \neg x_4, \top), (\neg x_4 \vee \neg x_5, \top), (\neg x_5 \vee \neg x_1, \top)\}$. From here on, in order to make the examples easier to follow, the weights are removed; i.e. assume the weights of every soft

Table 2 Running example for the linear Unsat-Sat algorithm

	$\varphi_W = \{(x_1 \vee r_1), (x_2 \vee r_2), (x_3 \vee r_3), (x_4 \vee r_4), (x_5 \vee r_5), (x_6 \vee r_6), (\neg x_6 \vee r_7)\} \cup \varphi^H$ $\lambda = 0$;
#1	Constraint to include: $CNF(r_1 + r_2 + r_3 + r_4 + r_5 + r_6 + r_7 \leq 0)$ st = UNSAT; $\lambda = 1$;
#2	Constraint to include: $CNF(r_1 + r_2 + r_3 + r_4 + r_5 + r_6 + r_7 \leq 1)$ st = UNSAT; $\lambda = 2$;
#3	Constraint to include: $CNF(r_1 + r_2 + r_3 + r_4 + r_5 + r_6 + r_7 \leq 2)$ st = UNSAT; $\lambda = 3$;
#4	Constraint to include: $CNF(r_1 + r_2 + r_3 + r_4 + r_5 + r_6 + r_7 \leq 3)$ st = UNSAT; $\lambda = 4$;
#5	Constraint to include: $CNF(r_1 + r_2 + r_3 + r_4 + r_5 + r_6 + r_7 \leq 4)$ st = SAT; $Init(\mathcal{A}) = \{x_3 = x_5 = 1; x_1 = x_2 = x_4 = x_6 = 0\}$

#*i* represents the *i*-th iteration of the algorithm

clause is set to 1 and of every hard clause is set to $\top$. If instance φ is solved with the Linear Search Unsat-Sat algorithm (Algorithm 2), the sequence of iterations is as shown in Table 2. The first column of the table shows the number of the iteration, where the first row is the initial iteration (before any call to the SAT solver). The second column shows the status of some of the structures of the algorithm, namely for the initial iteration both the working formula φ_W and the lower bound λ are presented. Since the Linear Search Unsat-Sat algorithm starts by relaxing all soft clauses in the working formula, then in the initial iteration φ_W is a union of all the hard clauses (φ^H) together with the set of soft clauses φ^S already relaxed. For the remaining iterations, Table 2 shows the constraint that together with the working formula is tested for satisfiability with the SAT solver. Variable *st* gives the outcome of the SAT solver call. This value is unsatisfiable (UNSAT) for iterations #1 to #4 and the lower bound is increased by 1 each time. For iteration #5, the SAT solver reports the formula to be satisfiable (SAT) and the satisfying assignment is shown in the last row of the table (restricted to the original variables). In the remaining of the paper, examples of the execution for several algorithms are presented in a similar fashion to this example.

Observe that in the pseudo-code, λ is updated by `RefineBound`$\{\omega_i | 1 \leq i \leq m\}, \lambda$ instead of just by 1. Function `RefineBound()` takes as parameters (i) a vector of weights and (ii) a bound, and it returns a refinement of the given bound. The possible refinements depend on the distribution of the weights. The following refinements are considered:

- $\lambda \leftarrow \lambda + 1$
- $\lambda \leftarrow SubSetSum(\{\omega_i | 1 \leq i \leq m\}, \lambda)$ (as suggested in [9])

For unweighted MaxSAT instances (i.e., all weights equal to 1), the bound refinement cannot be better than $\lambda + 1$. However, for weighted MaxSAT the bound

Algorithm 3 The Linear Sat-Unsat Algorithm

```
Input: $\varphi$
1 $(R, \varphi_W) \leftarrow$ RelaxCls($\emptyset$, $\varphi$, Soft($\varphi$))
2 $(\mu, last\mathcal{A}) \leftarrow (1 + \sum_{i=1}^{m} w_i, \emptyset)$
3 while true do
4     $(\mathrm{st}, \mathcal{A}) \leftarrow$ SAT(Cls($\varphi_W$) $\cup$ CNF($\sum_{i=1}^{m} w_i \cdot r_i \leq \mu - 1$))
5     if st = UNSAT then return Init($last\mathcal{A}$)
6     $(last\mathcal{A}, \mu) \leftarrow (\mathcal{A}, \mu - 1)$
7 end
```

refinement using the *subset sum* could save a considerable number of iterations [9]. Given a set of integers and an integer k, the *subset sum* problem asks if there is any subset of integers such that their sum equals k. The subset sum problem is a well-known NP-hard problem which can be solved by a *pseudo-polynomial* algorithm, for example, a *dynamic programming algorithm* [9]. The bound refinement $SubSetSum(\{\omega_i | 1 \leq i \leq m\}, \lambda)$ receives the current lower bound λ and the set of weights of the soft clauses, and it returns the next value for λ such that the subset sum is true.

Example 2 Let φ be a weighted formula with four soft clauses with weights 1, 2, 3, and 100 and a set of hard clauses. The only possible values for the optimum solution are in $0, \ldots, 6$ and in $100, \ldots, 106$. So, there is no need to assign λ to any of the values in $7, \ldots, 99$ for the Linear Search Unsat-Sat algorithm. If the algorithm is run on this instance and the bound refinement is just $\lambda + 1$, it will iterate over all the values $0, \ldots, 106$ in the worst case (i.e., $Opt(\varphi) = 106$). Alternatively, using the bound refinement based on the subset sum, it will iterate only over the values in $0, \ldots, 6$ and $100, \ldots, 106$ in the worst case.

Unless otherwise indicated, function `RefineBound()` will be used in the remaining algorithms that refine a lower bound.

Algorithm 3 shows the pseudo-code for *Linear Search Sat-Unsat*, which searches in the opposite direction than Linear Search Unsat-Sat. Here, in every iteration but the last, the instance is satisfiable. The optimum solution is found immediately before the first *unsatisfiable* instance. Starting from the sum of the weights of the soft clauses, an upper bound μ is decreased until the MaxSAT solution is found. Initially, all soft clauses are relaxed (line 1), and the upper bound is initialized to the sum of the weights plus one (line 2). The main loop (line 3) iterates until an unsatisfiable formula is found (line 5). At each iteration, the SAT solver is called with the clauses of the working formula and the encoding of the `AtMostK` constraint $\sum_{i=1}^{m} w_i \cdot r_i \leq \mu - 1$ (line 4). Otherwise, the upper bound is updated and the current assignment is stored (line 6). The algorithm terminates whenever an unsatisfiable formula is found (line 5).

An improved version of Algorithm 3 is presented in Algorithm 4 [25]. The advantage of this algorithm is that the value of μ may be decreased by a value greater than 1, since the assignment $\mathcal{A}$ in line 6 provides a possibly stronger upper bound $Opt(\varphi) \leq \sum_{i=1}^{m} w_i \leq \mu$, and $(\mu - \sum_{i=1}^{m} w_i) \geq 1$. Hence, Algorithm 4 represents the actual Linear Search Sat-Unsat that will be evaluated in the empirical investigation.

Algorithm 4 `LIN-SU`: The Linear Sat-Unsat Algorithm with Assignment Leap [25]

```
   Input: $\varphi$
1  $(R, \varphi_W) \leftarrow$ RelaxCls($\emptyset, \varphi$, Soft($\varphi$))
2  $(\mu, last\mathcal{A}) \leftarrow (1 + \sum_{i=1}^{m} w_i, \emptyset)$
3  while true do
4      (st, $\mathcal{A}$) $\leftarrow$ SAT(Cls($\varphi_W$)$\cup$ CNF($\sum_{i=1}^{m} w_i \cdot r_i \leq \mu - 1$))
5      if st = UNSAT then return Init($last\mathcal{A}$)
6      $(last\mathcal{A}, \mu) \leftarrow (\mathcal{A}, \sum_{i=1}^{m} w_i \cdot (1 - \mathcal{A}\langle c_i \setminus \{r_i\}\rangle))$
7  end
```

Table 3 Running example for the linear Search Sat-Unsat algorithm with Assignment Leap

	$\varphi_W = \{(x_1 \vee r_1), (x_2 \vee r_2), (x_3 \vee r_3), (x_4 \vee r_4), (x_5 \vee r_5), (x_6 \vee r_6), (\neg x_6 \vee r_7)\} \cup \varphi^H$ $\mu = 8$
#1	Constraint to include: $CNF(r_1+r_2+r_3+r_4+r_5+r_6+r_7 \leq 7)$ st = SAT; $\mathcal{A} = \{r_7 = 0; r_1 = r_2 = r_3 = r_4 = r_5 = r_6 = 1\} \cup Init(\mathcal{A})$ $\mu = 6$
#2	Constraint to include: $CNF(r_1+r_2+r_3+r_4+r_5+r_6+r_7 \leq 5)$ st = SAT; $\mathcal{A} = \{r_5 = r_7 = 0; r_1 = r_2 = r_3 = r_4 = r_6 = 1\} \cup Init(\mathcal{A})$ $\mu = 5$
#3	Constraint to include: $CNF(r_1+r_2+r_3+r_4+r_5+r_6+r_7 \leq 4)$ st = SAT; $\mathcal{A} = \{r_3 = r_5 = r_7 = 0; r_1 = r_2 = r_4 = r_6 = 1\} \cup Init(\mathcal{A})$ $\mu = 4$
#4	Constraint to include: $CNF(r_1+r_2+r_3+r_4+r_5+r_6+r_7 \leq 3)$ st = UNSAT; $\mathcal{A} = \emptyset$
	$Init(last\mathcal{A}) = \{x_3 = x_5 = 1; x_1 = x_2 = x_4 = x_6 = 0\}$

#*i* represents the *i*-th iteration of the algorithm

Example 3 Consider that the instance φ of Example 1 is given to Linear Search Sat-Unsat with Assignment Leap (Algorithm 4). Table 3 presents the iterations obtained from running Algorithm 4 on φ. In the first iteration, the call to the SAT solver returns a satisfying assignment $\mathcal{A}$ with 5 relaxation variables assigned to true. Given that each soft clause has weight 1, the upper bound can be safely updated to 5. This makes Algorithm 4 skip two iterations that Algorithm 3 would make.

4.2 Binary search based algorithms

This section presents iterative MaxSAT algorithms based on binary search. First is presented the binary search algorithm, followed by an algorithm that alternates between binary search and linear search. Then is presented an algorithm based on binary search in the binary representation of the value of the optimum cost.

Linear search-based algorithms for MaxSAT can take a number of iterations that grows linearly with $\sum_{i=1}^{m} w_i$, and so are *exponential* in the size of the problem

Algorithm 5 BIN: The Binary Search Algorithm [47]

```
Input: $\varphi$
1 $(R, \varphi_W) \leftarrow$ RelaxCls($\emptyset, \varphi,$ Soft($\varphi$))
2 $(\lambda, \mu, last\mathcal{A}) \leftarrow (-1, 1 + \sum_{i=1}^{m} w_i, \emptyset)$
3 while $\lambda + 1 < \mu$ do
4     $\nu \leftarrow \lfloor \frac{\mu+\lambda}{2} \rfloor$
5     $(\mathrm{st}, \mathcal{A}) \leftarrow$ SAT(Cls($\varphi_W$) $\cup$CNF($\sum_{i=1}^{m} w_i \cdot r_i \leq \nu$))        /* Optionally include $\sum_{i=1}^{m} w_i \cdot r_i > \lambda$ */
6     if st = SAT then $(last\mathcal{A}, \mu) \leftarrow (\mathcal{A}, \sum_{i=1}^{m} w_i \cdot (1 - \mathcal{A}\langle c_i \setminus \{r_i\}\rangle))$
7     else $\lambda \leftarrow$ RefineBound($\{w_i \mid 1 \leq i \leq m\}, \nu$)$-1$
8 end
9 return Init($last\mathcal{A}$)
```

instance[1]. Since we are searching for a value on a monotonic scale (the set of solution costs), *binary search* can be used instead. Two bounds (upper and lower) can be maintained, and binary search can be used to locate the optimum solution, again as the cost at the boundary between satisfiable and unsatisfiable.

Binary Search [47] (See Algorithm 5) maintains both a lower bound λ and an upper bound μ. Initially, all soft clauses are relaxed (line 1), and the lower and upper bounds are initialized, respectively, to -1 and to one plus the sum of the weights of the soft clauses (line 2). At each iteration, the middle value ν is computed (line 4), an AtMostK constraint is added to the working formula, requiring the sum of the weights of relaxed soft clauses to be lower or equal to ν, and the SAT solver is called on the resulting CNF formula (line 5). If the formula is satisfiable (line 6), then the optimum solution must be lower than ν, and the upper bound is updated. If the formula is unsatisfiable, then the optimum solution must be larger than ν and the lower bound is updated (line 7). The algorithm terminates when $\lambda + 1 = \mu$. The number of iterations of binary search grows with $\log(\sum_{i=1}^{m} w_i)$, and so it is *linear* in the size of the problem instance.

Example 4 Consider the execution of Binary Search (Algorithm 5) on the instance φ of Example 1. The sequence of steps of the algorithm is shown in Table 4, where it is assumed that RefinedBound() always returns $\nu + 1$.

Algorithm 6 shows the pseudo-code for the algorithm introduced in [69], which implements a mix of the two previous algorithms and will be referred to as BIN/LIN-SU. The authors (in [69]) motivate this algorithm by stating, "Generally speaking, binary search is better than linear search. Practically, however, there are instances for which linear search is better than binary search. [...] [The algorithm] alternates binary search and linear search in order to benefit from both searches." The algorithm can be in one of two possible *execution modes*: linear or binary search. Initially, all soft clauses are relaxed (line 1), and the lower and upper bounds are initialized to -1 and to the sum of the weights plus one, respectively (line 2). Additionally, the execution mode is also initialized to binary search (line 2). At each iteration, the SAT solver is called with the current working formula and an AtMostK constraint (line 5). Depending on the current execution mode, the AtMostK constraint is bounded with the upper bound (Linear Search Sat-Unsat

[1]Here, size denotes the number of bits used to represent the instance.

Table 4 Running example for the Binary Search Algorithm

	$\varphi_W = \{(x_1 \vee r_1), (x_2 \vee r_2), (x_3 \vee r_3), (x_4 \vee r_4), (x_5 \vee r_5), (x_6 \vee r_6), (\neg x_6 \vee r_7)\} \cup \varphi^H$ $\mu = 8, \quad \lambda = -1;$
#1	$\nu = 3$ Constraint to include: $CNF(r_1 + r_2 + r_3 + r_4 + r_5 + r_6 + r_7 \leq 3)$ $st =$ UNSAT; $\mu = 8, \quad \lambda = 3;$
#2	$\nu = 5$ Constraint to include: $CNF(r_1 + r_2 + r_3 + r_4 + r_5 + r_6 + r_7 \leq 5)$ $st =$ SAT; $\mathcal{A} = \{r_3 = r_5 = r_7 = 0; r_1 = r_2 = r_4 = r_6 = 1\} \cup Init(\mathcal{A})$ $\mu = 4, \quad \lambda = 3;$
	$Init(last\mathcal{A}) = \{x_3 = x_5 = 1; x_1 = x_2 = x_4 = x_6 = 0\}$

#*i* represents the *i*-th iteration of the algorithm

mode) or with the middle value between the lower and upper the bound (Binary Search mode). If the SAT solver returns satisfiable, the upper bound is updated. If the SAT solver returns unsatisfiable, the lower bound is updated to the middle value (line 7). As shown, in each iteration, the execution mode is swapped (line 8).

One additional approach related to binary search, exploits the fact that the solution is a natural number, which can be represented as a sequence of bits. The optimum solution can thus be found by "narrowing in" on the boundary between satisfiable and unsatisfiable instances bit-by-bit, such that each bit's optimum value is determined before moving to the next. The algorithm starts from the most significant bit (MSB) and moves, iteration by iteration, to the least significant bit, at which point it has found the exact solution. Algorithm 7 presents the pseudo-code for *Bit-Based Search*. Initially, all soft clauses are relaxed (line 1). The sum of the weights of the soft clauses is an upper bound on the optimum solution. Such an upper bound is used to decide how many bits are needed to represent the solution and thus which bit will be the MSB (line 2). The index of the current bit of interest is stored in ι, whereas ν is the actual solution value being constructed (line 3).

The main loop iterates until it has reached the least significant bit when $\iota = 0$ (line 4). Inside the loop, a call to the SAT solver is made stating that the sum of weights of the relaxed soft clauses should be lower than ν (line 5). If the SAT solver returns unsatisfiable, the search continues to the next most significant bit (line 13), and the value ν is increased by 2^ι (line 14), given that the solution is clearly greater than the current value of ν. If the SAT solver returns satisfiable, then the sum of weights of unsatisfied soft clauses by the current assignment is computed and the associated set of bits representing that value is created using a set of constants $s_0, \ldots, s_k$ (line 8). The index to the current bit ι is decreased to the next index $j < \iota$ such that the associated constant s_j is assigned to 1 (line 9). If no such bit exists, ι is set to -1 so that the algorithm terminates (line 9). Otherwise, the value ν is updated according to the set of constants assigned to 1.

Example 5 Consider the execution of Bit-Based Search (Algorithm 7) on the the instance φ of Example 1. The sequence of main steps of the algorithm is shown in Table 5.

Algorithm 6 BIN/LIN-SU: Alternating Binary Search and Linear Search Sat-Unsat [69]

```
    Input: $\varphi$
1   $(R, \varphi_W) \leftarrow$ RelaxCls($\emptyset, \varphi,$ Cls(Soft($\varphi$)))
2   $(\lambda, \mu, last\mathcal{A}, mode) \leftarrow (-1, 1 + \sum_{i=1}^{m} w_i, \emptyset, Binary)$
3   while $\lambda + 1 < \mu$ do
4       $\nu \leftarrow (mode = Binary)\ ?\ \lfloor \frac{\mu+\lambda}{2} \rfloor :\ \mu - 1$
5       $(\text{st}, \mathcal{A}) \leftarrow$ SAT(Cls($\varphi_W$) $\cup$ CNF($\sum_{i=1}^{m} w_i \cdot r_i \leq \nu$))
6       if st = $SAT$ then $(last\mathcal{A}, \mu) \leftarrow (\mathcal{A}, \sum_{i=1}^{m} w_i \cdot (1 - \mathcal{A}\langle c_i \setminus \{r_i\}\rangle))$
7       else $\lambda \leftarrow (mode = Binary)\ ?$ RefineBound($\{w_i \mid 1 \leq i \leq m\}, \nu$)$-1 :\ \nu$
8       $mode \leftarrow (mode = Binary)\ ?\ Linear :\ Binary$
9   end
10  return Init($last\mathcal{A}$)
```

Algorithm 7 BIT: The Bit-Based Search Algorithm [37, 53]

```
    Input: $\varphi$
1   $(R, \varphi_W) \leftarrow$ RelaxCls($\emptyset, \varphi,$ Soft($\varphi$))
2   $(last\mathcal{A}, k) \leftarrow (\emptyset, \lfloor \log_2(\sum_{i=1}^{m} w_i) \rfloor)$
3   $(\iota, \nu) \leftarrow (k, 2^k)$
4   while $\iota \geq 0$ do
5       $(\text{st}, \mathcal{A}) \leftarrow$ SAT( Cls($\varphi_W$) $\cup$ CNF ($\sum_{i=1}^{m} w_i \cdot r_i < \nu$))
6       if st = $SAT$ then
7           $last\mathcal{A} \leftarrow \mathcal{A}$
8           let $s_0, \ldots, s_k$ be constants such that $\sum_{i=1}^{m} w_i \cdot [1 - \mathcal{A}\langle c_i \setminus \{r_i\}\rangle] = \sum_{j=0}^{k} 2^j \cdot s_j$
9               $\iota \leftarrow \max(\{j \mid j < \iota$ and $s_j = 1\} \cup \{-1\})$
10              if $\iota \geq 0$ then $\nu \leftarrow \sum_{j=\iota}^{k} 2^j \cdot s_j$
11          end
12      else
13          $\iota \leftarrow \iota - 1$
14          $\nu \leftarrow \nu + 2^{\iota}$
15      end
16  end
17  return Init($last\mathcal{A}$)
```

4.3 Characterization of iterative algorithms

Table 6 presents a characterization of iterative MaxSAT algorithms. The first column enumerates several characteristics. The remaining columns refer to the following iterative MaxSAT algorithms: LIN-US is Linear Search Unsat-Sat, LIN-SU is Linear Search Sat-Unsat, BIN is Binary Search, BIN/LIN-SU alternates Binary Search with Linear Search Sat-Unsat and BIT is Bit-Based Search. The considered characteristics are:

- Progression: indicates if the algorithm refines a lower bound (LB) or an upper bound (UB).
- # Relax. Vars./Clause: The number of relaxation variables per soft clause.

Table 5 Running example for the Bit-Based Search Algorithm

	$\varphi_W = \{(x_1 \vee r_1), (x_2 \vee r_2), (x_3 \vee r_3), (x_4 \vee r_4), (x_5 \vee r_5), (x_6 \vee r_6), (\neg x_6 \vee r_7)\} \cup \varphi^H$ $k = 2$ $\iota = 2$ $\nu = 2^2 = 4$
#1	Constraint to include: $CNF(r_1 + r_2 + r_3 + r_4 + r_5 + r_6 + r_7 < 2^2)$ st = UNSAT $\iota = 1$ $\nu = 2^2 + 2^1 = 6$
#2	Constraint to include: $CNF(r_1 + r_2 + r_3 + r_4 + r_5 + r_6 + r_7 < 2^2 + 2^1)$ st = SAT; $\mathcal{A} = \{r_1 = r_2 = r_4 = r_6 = 1, r_3 = r_5 = r_7 = 0\} \cup Init(\mathcal{A})$ $\{s_0 = s_1 = 0;\ s_2 = 1\}$ $\iota = -1$ $Init(last\mathcal{A}) = \{x_3 = x_5 = 1; x_1 = x_2 = x_4 = x_6 = 0\}$

#*i* represents the *i*-th iteration of the algorithm

- Total # Relax. Vars.: The total number of relaxation variables added to the formula throughout the algorithm.
- # Const./Iteration: The number of constraints added at each iteration.
- Total # Const.: The total number of constraints added to the formula throughout the algorithm.
- Calls SAT Oracle: The theoretical worst case number of calls to a SAT oracle (solver).
- Weighted: If the algorithm handles weighted MaxSAT.

Recall that m is the total number of soft clauses in $\varphi = \varphi^S \cup \varphi^H$ (i.e., $m = |\varphi^S|$) and W be the sum of weights of soft clauses $W = \sum_{i=1}^{m} w_i$. Because all algorithms start by relaxing all soft clauses, all algorithms use exactly m relaxation variables, precisely one for each soft clause. All 5 algorithms maintain one `AtMostK` constraint at each iteration and in the last call to the SAT solver. Also, all 5 algorithms have been presented in their weighted MaxSAT version. The main differences between the algorithms are their progression and the worst case number of calls to a SAT solver. Linear Search Unsat-Sat refines a lower bound, whereas Linear Search Sat-Unsat refines an upper bound. Binary Search, Bit-Based Search and `BIN/LIN-SU` refine a lower and upper bound (LB+UB). Linear Search Unsat-Sat and Sat-Unsat require a worst case exponential number of calls to the SAT solver on the problem instance size whereas Binary Search, Bit-Based Search and `BIN/LIN-SU` require a linear number of calls (worst case).

5 Core-guided algorithms

This section describes several algorithms for MaxSAT that are guided by the discovery of unsatisfiable cores [128]. As such, the algorithms assume that the SAT solver used is able to return an unsatisfiable core whenever the input instance to the SAT solver is unsatisfiable. On the pseudo-code presented in this section, the function $SAT(\varphi)$ returns a triple $(st, \varphi_C, \mathcal{A})$, where st represents the satisfiability of

Table 6 Characteristics of Iterative MaxSAT Algorithms

Characteristic	LIN-US	LIN-SU	BIN	BIN/LIN-SU	BIT
Progression	LB	UB	LB + UB	LB+UB	LB + UB
# Relax. Vars./Clause	1	1	1	1	1
Total # Relax. Vars.	m	m	m	m	m
# Const./Iteration	1	1	1	1	1
Total # Const.	1	1	1	1	1
Calls SAT oracle	$O(W)$	$O(W)$	$O(log(W))$	$O(log(W))$	$O(log(W))$
Weighted	Yes	Yes	Yes	Yes	Yes

m is the number of *soft* clauses of φ and W is the sum of weights of soft clauses $W = \sum_{i=1}^{m} w_i$

Algorithm 8 The WMSU1 Algorithm [8, 47, 82]

```
Input: $\varphi$
1  $\varphi_W \leftarrow \varphi$
2  while true do
3      $(\text{st}, \varphi_C, \mathcal{A}) \leftarrow$ SAT(Cls($\varphi_W$))
4      if st = $SAT$ then return Init($\mathcal{A}$)
5      $(R, min_{\varphi_C}) \leftarrow (\emptyset, \min\{w \mid (c,w) \in$ Soft($\varphi_C$)$\})$
6      foreach $(c,w) \in$ Soft($\varphi_C$) do
7          let $r$ be a new relaxation variable and $R_c$ be the set of relaxation variables of $c$
8              $(R, c_r, w_r) \leftarrow (R \cup \{r\}, c \cup \{r\}, min_{\varphi_C})$
9              $\varphi_W \leftarrow \varphi_W \setminus \{(c,w)\} \cup \{(c_r, w_r)\}$    /* Optionally include CNF($r + \sum_{r_j \in R_c} r_j \leq 1$) */
10             if $w > min_{\varphi_C}$ then $\varphi_W \leftarrow \varphi_W \cup \{(c, w - min_{\varphi_C})\}$
11         end
12     end
13     $\varphi_W \leftarrow \varphi_W \cup$ CNF($\sum_{r \in R} r \leq 1$)
14 end
```

φ (*SAT* or *UNSAT*), φ_C represents the set of clauses in the core if φ is unsatisfiable, and $\mathcal{A}$ represents an assignment if φ is satisfiable.

In order to facilitate the presentation of the core-guided algorithms, the algorithms have been divided in three sections. First is presented the algorithms that consider possibly more than one relaxation variable per soft clause relaxed. These correspond to the WMSU1 and MSU2 algorithms.

Then, the algorithms that add at most one relaxation variable per soft clause are introduced. These algorithms were further divided in the set of algorithms are not based on binary search and the set of algorithms based on binary search. The section ends with a characterization of all the algorithms.

5.1 Algorithms based on multiple relaxation variables per soft clause

This section presents the WMSU1 and the MSU2 algorithms. As core-guided MaxSAT algorithms, both algorithms take advantage of unsatisfiable core to relax soft clauses belonging to cores. These algorithms are characterized by adding more than one relaxation variable to a soft clause, in the case that such clause belongs to more than one of the computed cores. The difference between the two algorithms is the way the relaxation variables and cardinality constraints are used by each algorithm.

The first core-guided algorithm in the literature is due to Fu & Malik [47]. It is similar to the *Linear Search Unsat-Sat* algorithm, but it limits the clauses it relaxes (thus reducing the search space) by using unsatisfiable cores. The algorithm is based on iteratively calling a SAT solver on a working formula φ_W, initially set to be the input formula φ. If φ_W is satisfiable, then the current satisfying assignment (provided by the SAT solver) is an optimum assignment, and the number of iterations required so far corresponds to the MaxSAT solution.

If φ_W is unsatisfiable, then the SAT solver provides a reason for the unsatisfiability of the formula, namely an unsatisfiable core φ_C. In this case, the algorithm proceeds by relaxing in φ_W each of the soft clauses of φ_C, and adding a new cardinality constraint to φ_W stating that one and only one of the new relaxation variables (created due to the current φ_C), must the assigned true (all others must be assigned false). Observe that each soft clause that belongs to more than one of the cores found gets relaxed more than once.

The original algorithm proposed in [47] used the *one-hot* constraint (the cardinality constraint is an equality $= 1$, i.e., one and only one variable in the constraint will be assigned true) which is quadratic on the input size, and was restricted to unweighted MaxSAT. In more recent works, the one-hot constraint is replaced by an `AtMost1` constraint with more efficient encodings such as the *bitwise* encoding [85] or the *regular* encoding [8].

The extension of the Fu & Malik algorithm [47] to handle weighted MaxSAT was presented simultaneously in [8, 82]. In [8], the resulting algorithm is referred to as `WPM1`, while in [82] the resulting algorithm is referred as `WMSU1`.[2]

The pseudo-code (for the weighted version) of `WMSU1` is presented in Algorithm 8. The algorithm progresses through unsatisfiable instances until a satisfiable instance is found. The working formula φ_W is maintained between iterations (line 1). Each time the working formula is found to be unsatisfiable (line 4), an unsatisfiable core φ_C is obtained (line 5), and each soft clause in the core is augmented with a fresh relaxation variable (line 8). Then, a new hard constraint is added to the formula to limit the number of the relaxation variables to be assigned true to at most one (line 13).

`WMSU1` was improved in [86] by adding an additional cardinality constraints to φ_W for each soft clause with more than one relaxation variable. Since on these clauses at most one of the relaxation variables gets assigned true, then the cardinality constraint allows at most one of these relaxation variables to be assigned true (optional constraint in line 9).

The introduction of weights in `WMSU1` requires *clause replication*. Let min_{φ_C} be the minimum weight of the soft clauses in the current unsatisfiable core φ_C (line 5). The weighted version of `WMSU1` proceeds (line 6) by replicating each soft clause in φ_C and extending each replicated clause with an additional relaxation variable (line 8). Further, each replicated clause is assigned weight min_{φ_C} (line 8). Finally, the weight of each soft clause in φ_C is decremented by min_{φ_C} (line 10). Note that in the unweighted version of `WMSU1` each soft clause has a weight of 1, and as such line 10 is never applied (since min_{φ_C} is always 1).

[2]In the remainder of this work, we will abuse the naming and refer to the algorithm of Fu & Malik [47], to the `WPM1` algorithm of [8] and to the `WMSU1` algorithm of [8], by the same name `WMSU1`.

Table 7 Running example for the WMSU1 Algorithm

	$\varphi_W = \varphi^H \cup \{(x_1), (x_2), (x_3), (x_4), (x_5), (x_6), (\neg x_6)\}$
#1	st = UNSAT; $Soft(\varphi_C) = \{(x_6), (\neg x_6)\}$; New relaxation variables r_1, r_2; Resulting $\varphi_W = \varphi^H \cup \{(x_1), (x_2), (x_3), (x_4), (x_5), (x_6 \vee r_1), (\neg x_6 \vee r_2)\}$ $\cup\ CNF(\mathbf{r_1 + r_2 \leq 1})$
#2	st = UNSAT; $Soft(\varphi_C) = \{(x_1), (x_2)\}$; New relaxation variables r_3, r_4; Resulting $\varphi_W = \varphi^H \cup \{(x_1 \vee r_3), (x_2 \vee r_4), (x_3), (x_4), (x_5), (x_6 \vee r_1), (\neg x_6 \vee r_2)\}$ $\cup\ CNF(r_1 + r_2 \leq 1) \cup\ CNF(\mathbf{r_3 + r_4 \leq 1})$
#3	st = UNSAT; $Soft(\varphi_C) = \{(x_3), (x_4)\}$; New relaxation variables r_5, r_6; Resulting $\varphi_W = \varphi^H \cup \{(x_1 \vee r_3), (x_2 \vee r_4), (x_3 \vee r_5), (x_4 \vee r_6), (x_5), (x_6 \vee r_1), (\neg x_6 \vee r_2)\}$ $\cup\ CNF(r_1 + r_2 \leq 1) \cup CNF(r_3 + r_4 \leq 1) \cup CNF(\mathbf{r_5 + r_6 \leq 1})$
#4	st = UNSAT; $Soft(\varphi_C) = \{(x_1 \vee r_3), (x_2 \vee r_4), (x_3 \vee r_5), (x_4 \vee r_6), (x_5)\}$; New relaxation variables $r_7, r_8, r_9, r_{10}, r_{11}$; Resulting $\varphi_W = \varphi^H \cup \{(x_1 \vee r_3 \vee r_7), (x_2 \vee r_4 \vee r_8), (x_3 \vee r_5 \vee r_9), (x_4 \vee r_6 \vee r_{10}), (x_5 \vee r_{11}),$ $(x_6 \vee r_1), (\neg x_6 \vee r_2)\} \cup CNF(r_1 + r_2 \leq 1) \cup\ CNF(r_3 + r_4 \leq 1)$ $\cup CNF(r_5 + r_6 \leq 1) \cup\ CNF(\mathbf{r_7 + r_8 + r_9 + r_{10} + r_{11} \leq 1})$
#5	st = SAT; $Init(\mathcal{A}) = \{x_3 = x_5 = 1; x_1 = x_2 = x_4 = x_6 = 0\}$

#i represents the i–th iteration of the algorithm

Example 6 Consider that the instance φ of Example 1 is solved with algorithm WMSU1 (Algorithm 8). The execution sequence of WMSU1 would be as in Table 7.

Example 7 Let $\varphi_C = \{(x_1 \vee x_2, 5), (\neg x_1 \vee x_2, 6), (x_1 \vee \neg x_2, 7), (\neg x_1 \vee \neg x_2, 8)\}$ be an unsatisfiable core with just soft clauses found by the SAT solver. The minimum weight among such clauses is $min_{\varphi_C} = 5$. Clause replication will create the following set of clauses with an additional relaxation variable: $\{(x_1 \vee x_2 \vee r_1, 5), (\neg x_1 \vee x_2 \vee r_2, 5), (x_1 \vee \neg x_2 \vee r_3, 5), (\neg x_1 \vee \neg x_2 \vee r_4, 5)\}$ and the weight of the original soft clauses will be decremented by min_{φ_C} resulting in $\{(\neg x_1 \vee x_2, 1), (x_1 \vee \neg x_2, 2), (\neg x_1 \vee \neg x_2, 3)\}$.

Observe that WMSU1 is the only MaxSAT algorithm that adds more than one relaxation variable per soft clause. Whereas the remaining MaxSAT algorithms described in this paper add and remove AtMostK constraints at each iteration, WMSU1 is the only algorithm that just adds AtMost1 and AtLeast1 constraints that *become* part of the working formula. For this reason, WMSU1 is said to *transform* each problem instance into an equivalent one at each iteration in [8].

The MSU2 algorithm [85] aims to reduce the number of relaxation variables added by working directly with the *auxiliary variables* of the *bitwise encoding* [105]. In MSU2, the bitwise encoding operates on the soft clauses of each identified unsatisfiable core. This essentially allows it to eliminate the relaxation variables by working directly with the auxiliary variables used by the encoding. For an unsatisfiable core with k soft

Algorithm 9 The MSU2 Algorithm [85]

```
   Input: $\varphi$
1
2  function BinRelax ($\varphi_I$, $\varphi_W$)
3      $(\varphi_o, k, n) \leftarrow (\emptyset, |\varphi_I|, 1)$
4      if $k \neq 1$ then $n \leftarrow \lceil \log_2 k \rceil$
5      let $r_0, \ldots, r_{n-1}$ be fresh relaxation variables
6          foreach $i \in [0, k-1]$ do
7              let $c_i$ is $i$-th clause of $\varphi_I$
8                  foreach $c_R \in$ getAssocCls($\varphi_W, c_i$) do
9                      foreach $j \in [0, n-1]$ do
10                         if binary representation of $i$ has value 1 in position $j$ then $\varphi_o \leftarrow \varphi_o \cup \{c_R \cup \{r_j\}\}$
11                         else $\varphi_o \leftarrow \varphi_o \cup \{c_R \cup \{\neg r_j\}\}$
12                     end
13                 end
14             end
15         end
16     end
17     return $\varphi_o$                                    // Clauses in $\varphi_o$ marked soft
18 end
19
20 $\varphi_W \leftarrow \varphi$
21 while true do
22     $(\text{st}, \varphi_C, \mathcal{A}) \leftarrow$ SAT(Cls($\varphi_W$))
23     if st = $SAT$ or Soft($\varphi_C$)= $\emptyset$ then return Init($\mathcal{A}$)
24     $(\varphi_I, \varphi_R) \leftarrow (\emptyset, \emptyset)$
25     foreach $c_R \in Soft(\varphi_C)$ do
26         $c \leftarrow$ getInitCl($\varphi, c_R$)          /* getInitCl($\varphi, c_R$)= $c \in \varphi$ such that $c_R = (c \cup \bigcup\{r\})$ */
27         $\varphi_I \leftarrow \varphi_I \cup \{c\}$
28         $\varphi_R \leftarrow \varphi_R \cup$ getAssocCls($\varphi_W, c$)   /* getAssocCls($\varphi_W, c$)= $\{c_R \mid c_R = (c \cup \bigcup\{r\}) \in \varphi_W\}$ */
29     end
30     $\varphi_W \leftarrow \varphi_W \setminus \varphi_R \cup$ BinRelax($\varphi_I, \varphi_W$)
31 end
```

clauses, the bitwise encoding requires n auxiliary variables, where $n = 1$ if $k = 1$ and $n = \log_2 \lceil k \rceil$ if $k > 1$. Moreover, the encoding requires $O(\log k)$ new clauses for each original clause in the unsatisfiable core. Assume a clause c_{ij}, relaxed from an original clause c_i, is included in an identified unsatisfiable core. Then all clauses generated from c_i need to be re-relaxed. For this reason, the pseudo-code assumes an auxiliary function, *getAssocCls*(φ_W, c) that, given a set of clauses φ_W and an initial clause c, returns the subset of clauses of φ_W that originated from c. Note that MSU2 does not use the function RelaxCls, and instead uses the *BinRelax* function for relaxing the soft clauses as explained below.

The pseudo-code of MSU2 is shown in Algorithm 9. In the main loop, the algorithm iteratively calls the SAT solver with the current working formula (line 22). Whenever the SAT solver returns satisfiable, the algorithm terminates and returns the solution (line 23). Otherwise, the algorithm traverses the set of soft clauses in the unsatisfiable core and creates two sets φ_I and φ_R. For each soft clause c_R in the unsatisfiable core (line 25), the original clause c associated with c_R is retrieved (line 26) and added to φ_I (line 27). Moreover, all soft clauses originating from c are added to φ_R (line 28). Then, the working formula is updated by removing the clauses in φ_R and adding the new set of soft clauses provided by the bitwise encoding (line 30).

Function *BinRelax* receives the set of original soft clauses φ_I and the working formula and returns a set of new soft clauses φ_o. Initially, φ_o is empty (line 3). For

Table 8 Running example for the `MSU2` Algorithm

	$\varphi_W = \{(x_1), (x_2), (x_3), (x_4), (x_5), (x_6), (\neg x_6)\} \cup \varphi^H$
#1	$st = \text{UNSAT}$; $\mathit{Soft}(\varphi_C) = \{(x_6), (\neg x_6)\}$; New relaxation variable r_1; Resulting $\varphi_W = \{(x_1), (x_2), (x_3), (x_4), (x_5), (x_6 \vee \neg r_1), (\neg x_6 \vee r_1)\} \cup \varphi^H$
#2	$st = \text{UNSAT}$; $\mathit{Soft}(\varphi_C) = \{(x_1), (x_2)\}$; New relaxation variable r_2; Resulting $\varphi_W = \{(x_1 \vee \neg r_2), (x_2 \vee r_2), (x_3), (x_4), (x_5), (x_6 \vee \neg r_1), (\neg x_6 \vee r_1)\} \cup \varphi^H$
#3	$st = \text{UNSAT}$; $\mathit{Soft}(\varphi_C) = \{(x_3), (x_4)\}$; New relaxation variable r_3; Resulting $\varphi_W = \{(x_1 \vee \neg r_2), (x_2 \vee r_2), (x_3 \vee \neg r_3), (x_4 \vee r_3), (x_5), (x_6 \vee \neg r_1), (\neg x_6 \vee r_1)\} \cup \varphi^H$
#4	$st = \text{UNSAT}$; $\mathit{Soft}(\varphi_C) = \{(x_1 \vee \neg r_2), (x_2 \vee r_2), (x_3 \vee \neg r_3), (x_4 \vee r_3), (x_5)\}$; New relaxation variables r_4, r_5, r_6; Resulting $\varphi_W = \{(x_1 \vee \neg r_2 \vee \neg r_4), (x_1 \vee \neg r_2 \vee \neg r_5), (x_1 \vee \neg r_2 \vee \neg r_6),$ $(x_2 \vee r_2 \vee r_4), (x_2 \vee r_2 \vee \neg r_5), (x_2 \vee r_2 \vee \neg r_6),$ $(x_3 \vee \neg r_3 \vee \neg r_4), (x_3 \vee \neg r_3 \vee r_5), (x_3 \vee \neg r_3 \vee \neg r_6),$ $(x_4 \vee r_3 \vee r_4), (x_4 \vee r_3 \vee r_5), (x_4 \vee r_3 \vee \neg r_6),$ $(x_5 \vee \neg r_4), (x_5 \vee \neg r_5), (x_5 \vee r_6),$ $(x_6 \vee \neg r_1), (\neg x_6 \vee r_1)\} \cup \varphi^H$
#5	$st = \text{SAT}$; $\mathit{Init}(\mathcal{A}) = \{x_3 = x_5 = 1; x_1 = x_2 = x_4 = x_6 = 0\}$

#i represents the i–th iteration of the algorithm

an unsatisfiable core with k original soft clauses (i.e., $k = |\varphi_I|$), the bitwise encoding requires n auxiliary variables, where $n = 1$ if $k = 1$ and $n = \log_2\lceil k \rceil$ if $k > 1$ (lines 3 and 4). Observe that those auxiliary variables are at the same time the relaxation variables associated with soft clauses. For this reason, the pseudo-code refers to them simply as relaxation variables (line 5). For each original soft clause c_i in φ_I (line 7), the algorithm iterates over each soft clause c_R originating from c_i. Then, c_R is replicated n times. Each replicated clause j with $0 \leq j \leq (n-1)$ is extended with the relaxation variable r_j if the binary representation of i has value 1 in position j (line 10), otherwise it is extended with $\neg r_j$ (line 11). The replicated clauses are added to 17).

Example 8 Consider that the instance φ of Example 1 is solved with algorithm `MSU2` (Algorithm 9). Table 8 shows the execution of the algorithm.

5.2 Non-binary search based algorithms

This section presents five core-guided MaxSAT algorithms whose underlying search is not based in binary search. Unlike the two algorithms of the previous section, the algorithms presented in this section consider adding to the soft clauses at most one relaxation variable. The differences between the solvers include the direction in which the search proceeds, which bound(s) are considered, among others.

`WMSU3` [86] aims to use a smaller number of relaxation variables. This is achieved by adding relaxation variables *on demand* and *only one relaxation variable is added per soft clause*, in order to keep the number of new variables low. As it is shown in

Algorithm 10 The WMSU3 Algorithm [86]

```
Input: $\varphi$
1 $(R, \varphi_W, \lambda) \leftarrow (\emptyset, \varphi, 0)$
2 while true do
3     $(\text{st}, \varphi_C, \mathcal{A}) \leftarrow$ SAT(Cls($\varphi_W$) $\cup$ CNF($\sum_{\{i:r_i \in R\}} w_i \cdot r_i \leq \lambda$))
4     if st = $SAT$ then return $\mathcal{A}$
5     $(R, \varphi_W) \leftarrow$ RelaxCls($R, \varphi_W$,Soft($\varphi_C \cap \varphi$))
6     $\lambda \leftarrow$RefineBound($\{w_i \mid r_i \in R\}, \lambda$)
7 end
```

Algorithm 11 The WMSU4 Algorithm [87]

```
Input: $\varphi$
1  $(\varphi_W, R, \lambda, \mu, last\mathcal{A}) \leftarrow (\varphi, \emptyset, -1, 1 + \sum_{i=1}^{m} w_i, \emptyset)$
2  while $\mu > \lambda + 1$ do
3      $(\text{st}, \varphi_C, \mathcal{A}) \leftarrow$ SAT(Cls($\varphi_W$) $\cup$ CNF($\sum_{\{i:r_i \in R\}} w_i \cdot r_i \leq \mu - 1$))
4      if st = $SAT$ then $(last\mathcal{A}, \mu) \leftarrow (\mathcal{A}, \sum_{\{i:r_i \in R\}} w_i \cdot (1 - \mathcal{A}\langle c_i \setminus \{r_i\}\rangle))$
5      else
6          $(I, \varphi_W) \leftarrow$ RelaxCls($\emptyset, \varphi_W$, Soft($\varphi_C \cap \varphi$))
7          if $I = \emptyset$ then $\lambda \leftarrow \mu - 1$
8          else $(R, \lambda) \leftarrow (R \cup I$, RefineBound($\{w_i \mid r_i \in R \cup I\}, \lambda$))     /* Optionally CNF($\sum_{r \in I} r \geq 1$) */
9      end
10 end
11 return $last\mathcal{A}$
```

Algorithm 10, WMSU3 initializes the lower bound λ to 0 (line 1). Then, WMSU3 iterates (line 2) through unsatisfiable instances until a satisfiable instance is found (line 4). However, in contrast to WMSU1, it only adds *one relaxation variable per soft clause* (line 5) and only one *at most constraint* is maintained (line 3). For each unsatisfiable core found, each soft clause not yet relaxed gets a new relaxation variable (line 5) and the lower bound λ is updated to the next value. Observe that this algorithm is very similar to *Linear Search Unsat-Sat* (Algorithm 2) and that the main difference is the relaxation of soft clauses. In the latter, the relaxation of variables is done to all soft clauses prior to the main loop, whereas in WMSU3 it is only done depending on the unsatisfiable cores found.

Similarly to the previous algorithm, WMSU4 [87] (Algorithm 11) relaxes each soft clause at most once. However, WMSU4 maintains both an upper bound μ and a lower bound λ. On the calls to the SAT solver that return satisfiable, μ is updated according to the assignment $\mathcal{A}$ (line 4). On the unsatisfiable ones, every soft clause of the core which is not yet relaxed, is extended with a fresh relaxation variable (line 6) and λ is increased (line 8). If all soft clauses of an unsatisfiable core found have been relaxed, then the algorithm exits the main loop (line 7). In the original description of the algorithm [87] an additional cardinality constraint is added every time a core is found $\sum_{r \in I} r \geq 1$ (line 8).

Example 9 For the WMSU4, a different example is presented instead of the running example. Consider the formula $\varphi = \varphi^S \cup \varphi^H$, where $\varphi^S = \{(x_1, 1), (x_2, 1), (x_3, 1), (x_4, 1)\}$ and $\varphi^H = \{(\neg x_1 \vee \neg x_2, \top), (\neg x_1 \vee \neg x_3, \top), (\neg x_1 \vee \neg x_4, \top), (\neg x_2 \vee \neg x_3 \vee \neg x_4, \top)\}$. A possible set of iterations for WMSU4 with φ is shown in Table 9.

Table 9 Running example for the WMSU4 Algorithm

	$\varphi_W = \{(x_1), (x_2), (x_3), (x_4)\} \cup \varphi^H$ $\mu = 5; \lambda = -1;$
#1	st = UNSAT; $Soft(\varphi_C) = \{(x_2), (x_3), (x_4)\};$ $\lambda = 0;$ New relaxation variables $r_1, r_2, r_3;$ Resulting $\varphi_W = \{(x_1), (x_2 \vee r_1), (x_3 \vee r_2), (x_4 \vee r_3)\} \cup \varphi^H$
#2	Constraint to include: $CNF(r_1 + r_2 + r_3 \leq 4)$ st = SAT; $\mathcal{A} = \{x_1 = r_1 = r_2 = r_3 = 1; x_2 = x_3 = x_4 = 0\}$ $\mu = 3;$
#3	Constraint to include: $CNF(r_1 + r_2 + r_3 \leq 2)$ st = UNSAT; $Soft(\varphi_C) = \{(x_1), (x_2 \vee r_1), (x_3 \vee r_2), (x_4 \vee r_3)\};$ $\lambda = 1;$ New relaxation variable $r_4;$ Resulting $\varphi_W = \{(x_1 \vee r_4), (x_2 \vee r_1), (x_3 \vee r_2), (x_4 \vee r_3)\} \cup \varphi^H$
#4	Constraint to include: $CNF(r_1 + r_2 + r_3 + r_4 \leq 2)$ st = SAT; $\mathcal{A} = \{x_1 = x_2 = r_3 = r_4 = 0; x_3 = x_4 = r_1 = r_2 = 1\}$ $\mu = 2;$

#*i* represents the *i*−th iteration of the algorithm

Algorithm 12 The PM2 Algorithm [8]

```
Input: $\varphi$
1  $(R, \varphi_W) \leftarrow$ RelaxCls($\emptyset, \varphi,$ Soft($\varphi$))
2  $(\lambda, \mathcal{C}, AL) \leftarrow (0, \emptyset, \emptyset)$
3  while true do
4      $(\text{st}, \varphi_C, \mathcal{A}) \leftarrow$ SAT(Cls($\varphi_W$) $\cup$ CNF( $AL \cup \sum_{r \in R} r \leq \lambda$))
5      if st = $SAT$ or Soft($\varphi_C$)= $\emptyset$ then return Init($\mathcal{A}$)
6      $R_C \leftarrow \{r \mid (c \cup \{r\}) \in$ Soft($\varphi_C$) and $r \in R\}$
7      $\mathcal{C} \leftarrow \mathcal{C} \cup \{R_C\}$
8      $k \leftarrow |\{R \in \mathcal{C} \mid R \subseteq R_C\}|$
9      $AL \leftarrow AL \cup \sum_{r \in R_C} r \geq k$
10     $\lambda \leftarrow \lambda + 1$
11 end
```

Observe that on this particular example, WMSU4 changes between satisfiable and unsatisfiable iterations.

The original PM2 [8] is similar to a Linear Search Unsat-Sat approach. The main difference is that PM2 adds an additional AtLeastK cardinality constraint to the working formula at each iteration. PM2 and subsequent algorithms record information regarding the unsatisfiable cores found so far. The information maintained for each unsatisfiable core is the set of relaxation variables associated with each soft clause. No information about hard clauses is recorded.

In particular, PM2 (Algorithm 12) maintains a record of each unsatisfiable core found so far in a structure $\mathcal{C}$ that contains the set of relaxation variables for each core. $\mathcal{C}$ and the set of AtLeastK constraints AL are initially empty and the lower bound λ is initialized to 0 (line 2). All soft clauses are relaxed (line 1) before entering the

Algorithm 13 The PM2.1 Algorithm [7]

```
Input: $\varphi$
1  $(R, \varphi_W) \leftarrow$ RelaxCls($\emptyset, \varphi,$ Soft($\varphi$))
2  $(\mathcal{C}, AL, AM) \leftarrow (\emptyset, \emptyset, \{(r \leq 0) \mid r \in R\})$
3  while true do
4      $(\text{st}, \varphi_C, \mathcal{A}) \leftarrow$ SAT(Cls($\varphi_W$) $\cup$ CNF( $AL \cup AM$))
5      if st $= SAT$ or Soft($\varphi_C$)$= \emptyset$ then return Init($\mathcal{A}$)
6      $R_C \leftarrow \{r \mid (c \cup \{r\}) \in$ Soft($\varphi_C$) and $r \in R\}$
7      $\mathcal{C} \leftarrow \mathcal{C} \cup \{R_C\}$
8      $k \leftarrow |\{R \in \mathcal{C} \mid R \subseteq R_C\}|$
9      $AL \leftarrow AL \cup \sum_{r \in R_C} r \geq k$
10     $AM \leftarrow \emptyset$
11     foreach $R_{cover} \in$ CoversOf($\mathcal{C}$) do
12         $k \leftarrow |\{R \in \mathcal{C} \mid R \subseteq R_{cover}\}|$
13         $AM \leftarrow AM \cup \sum_{r \in R_{cover}} r \leq k$
14     end
15 end
```

main loop (line 3). The SAT solver is called at each iteration with the current working formula, the set of AtLeastK constraints AL, and an AtMostK constraint bounded to the current lower bound λ (line 4). For each new core, the relaxation variables associated with each soft clause are stored in R_C (line 6), and the information of the new core is added to the $\mathcal{C}$ structure (line 7). Then, the recorded cores $\mathcal{C}$ are traversed to check whether their soft clauses are included in the new core. Finally, an AtLeastK cardinality constraint is added to the AL set (line 9), meaning that the number of variables in the set of relaxation variables of the new core R_C that need to be one is at least the number of cores k included in such a new core, including itself (line 8). For the sake of clarity, the set of AtLeastK constraints AL is maintained as an independent set but indeed each new AtLeastK constraint *becomes* part of the working formula. However, the AtMostK constraint (line 4) is actually added and removed at each iteration.

PM2.1 [7] is an extension of PM2 that maintains *sets of covers*. Basically, each unsatisfiable core will result in an AtLeastK cardinality constraint, and every *cover* in an AtMostK cardinality constraint. Let $\mathcal{C}$ be a set of unsatisfiable cores. Each core $R_1 \in \mathcal{C}$ is defined by the set of relaxation variables associated with the soft clauses. A set of relaxation variables R_2 is a cover of $\mathcal{C}$ if it is a minimal set such that, for each $R_1 \in \mathcal{C}$, if $R_1 \cap R_2 \neq \emptyset$, then $R_1 \subseteq R_2$. The expression $CoversOf(\mathcal{C})$ denotes the set of covers of $\mathcal{C}$.

Algorithm 13 corresponds to PM2.1, and is similar to PM2 (lines 1 to 9). The main difference is that an additional set of AtMostK constraints AM is maintained which is initially empty (line 1). At each iteration, the SAT solver is called with the current working formula and the sets AL and AM (line 3). Then, the set of AtLeastK constraints AL is augmented in the same way as in PM2 (lines 6 to 9). Finally, the set of AtMostK constraints AM is prepared for the next iteration. First, the set AM is emptied. Then, for each cover in $\mathcal{C}$, AM is augmented with an additional AtMostK constraint. Function $CoversOf(\mathcal{C})$ divides the set of cores stored in $\mathcal{C}$ into a set of covers. This function traverses the set of cores $\mathcal{C}$ and *joins* together the relaxation

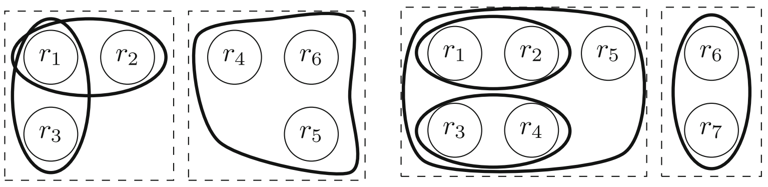

Fig. 2 Representation of clauses, cores and covers in Example 10i and Example 11ii

variables of those cores in $\mathcal{C}$ sharing some relaxation variable (line 11). The resulting sets of relaxation variables are the so-called covers. Then, for each set of relaxation variables R_{cover} of a cover, a new `AtMostK` constraint bounded to k is added (line 13), where k is the total number of cores that were joined to obtain such cover (line 12).

Table 10 Running example for the `PM2.1` Algorithm

	$\varphi_W = \{(x_1 \vee r_1), (x_2 \vee r_2), (x_3 \vee r_3), (x_4 \vee r_4), (x_5 \vee r_5), (x_6 \vee r_6), (\neg x_6 \vee r_7)\} \cup \varphi^H$
	$\mathcal{C} = \emptyset\,; \quad AL = \emptyset$
	$AM = (r_1 \leq 0), (r_2 \leq 0), (r_3 \leq 0), (r_4 \leq 0), (r_5 \leq 0), (r_6 \leq 0), (r_7 \leq 0)$
#1	$st = \text{UNSAT}; \quad Soft(\varphi_C) = \{(x_6 \vee r_6), (\neg x_6 \vee r_7)\}$
	$R_C = \{r_6, r_7\}$
	$\mathcal{C} = \{\{\mathbf{r_6}, \mathbf{r_7}\}\}$
	$k = 1$
	$AL = \{(\mathbf{r_6} + \mathbf{r_7} \geq \mathbf{1})\}$
	$AM = \{(r_1 \leq 0), (r_2 \leq 0), (r_3 \leq 0), (r_4 \leq 0), (r_5 \leq 0), (\mathbf{r_6} + \mathbf{r_7} \leq \mathbf{1})\}$
#2	$st = \text{UNSAT}; \quad Soft(\varphi_C) = \{(x_1 \vee r_1), (x_2 \vee r_2)\}$
	$R_C = \{r_1, r_2\}$
	$\mathcal{C} = \{\{r_6, r_7\}, \{\mathbf{r_1}, \mathbf{r_2}\}\}$
	$k = 1$
	$AL = \{(r_6 + r_7 \geq 1), (\mathbf{r_1} + \mathbf{r_2} \geq \mathbf{1})\}$
	$AM = \{(r_3 \leq 0), (r_4 \leq 0), (r_5 \leq 0), (r_6 + r_7 \leq 1), (\mathbf{r_1} + \mathbf{r_2} \leq \mathbf{1})\}$
#3	$st = \text{UNSAT}; \quad Soft(\varphi_C) = \{(x_3 \vee r_3), (x_4 \vee r_4)\}$
	$R_C = \{r_3, r_4\}$
	$\mathcal{C} = \{\{r_1, r_2\}, \{r_6, r_7\}, \{\mathbf{r_3}, \mathbf{r_4}\}\}$
	$k = 1$
	$AL = \{(r_1 + r_2 \geq 1), (r_6 + r_7 \geq 1), (\mathbf{r_3} + \mathbf{r_4} \geq \mathbf{1})\}$
	$AM = \{(r_1 + r_2 \leq 1), (r_5 \leq 0), (r_6 + r_7 \leq 1), (\mathbf{r_3} + \mathbf{r_4} \leq \mathbf{1})\}$
#4	$st = \text{UNSAT}; \quad Soft(\varphi_C) = \{(x_1 \vee r_1), (x_2 \vee r_2), (x_3 \vee r_3), (x_4 \vee r_4), (x_5 \vee r_5)\}$
	$R_C = \{r_1, r_2, r_3, r_4, r_5\}$
	$\mathcal{C} = \{\{r_1, r_2\}, \{r_3, r_4\}, \{r_6, r_7\}, \{\mathbf{r_1}, \mathbf{r_2}, \mathbf{r_3}, \mathbf{r_4}, \mathbf{r_5}\}\}$
	$k = 3$
	$AL = \{(r_1 + r_2 \geq 1), (r_3 + r_4 \geq 1), (r_6 + r_7 \geq 1), (\mathbf{r_1} + \mathbf{r_2} + \mathbf{r_3} + \mathbf{r_4} + \mathbf{r_5} \geq \mathbf{1})\}$
	$AM = \{(r_6 + r_7 \leq 1), (\mathbf{r_1} + \mathbf{r_2} + \mathbf{r_3} + \mathbf{r_4} + \mathbf{r_5} \leq \mathbf{3})\}$
#5	$st = \text{SAT}; \; Init(\mathcal{A}) = \{x_3 = x_5 = 1; x_1 = x_2 = x_4 = x_6 = 0\}$

#*i* represents the *i*−th iteration of the algorithm

Algorithm 14 The WPM2 Algorithm [9]

```
Input: $\varphi$
function Newbound ($AL$, $B$)
    $\kappa \leftarrow$ Bound($AL$, $B$)        /* Bound()= $\sum\{k' \mid \sum_{i\in B'} w_i \cdot r_i \leq k' \in AM \wedge B' \subseteq B\}$ */
    repeat
        $\kappa \leftarrow$ SubSetSum( $\{w_i \mid i \in B\}, \kappa$)
    until SAT( CNF($AL \cup \{\sum_{i\in B} w_i \cdot r_i = \kappa\}$ ))
    return $\kappa$
end
$(R, \varphi_W) \leftarrow$ RelaxCls($R, \varphi$,Soft($\varphi$))
$SC \leftarrow \{\{i\} \mid (c_i, w_i) \in$ Soft($\varphi$)$\}$        /* Set covers */
$AL \leftarrow \emptyset$        /* AtLeast constraints */
$AM \leftarrow \{w_i \cdot r_i \leq 0 \mid (c_i \cup \{r_i\}, w_i) \in$ Soft($\varphi_W$) and $r_i \in R\}$        /* AtMost constraints */
while true do
    (st, $\varphi_C, \mathcal{A}$) $\leftarrow$SAT(Cls($\varphi_W$) $\cup$ CNF($AL \cup AM$))
    if st = $SAT$ or Soft($\varphi_C$)= $\emptyset$ then return Init($\mathcal{A}$)
    $A \leftarrow \{i \mid (c_i \vee r_i) \in$ Soft($\varphi_C$) and $r_i \in R\}$
    $RC \leftarrow \{B' \in SC \mid B' \cap A \neq \emptyset\}$
    $B \leftarrow \bigcup_{B' \in RC} B'$
    $k \leftarrow$ Newbound($AL, B$)
    $SC \leftarrow SC \setminus RC \cup \{B\}$
    $AL \leftarrow AL \cup \{\sum_{i\in B} w_i \cdot r_i \geq k\}$
    $AM \leftarrow AM \setminus \{\sum_{i\in B'} w_i \cdot r_i \leq k' \mid B' \in RC\} \cup \{\sum_{i\in B} w_i \cdot r_i \leq k\}$
end
```

Example 10 Let $\mathcal{C}$ be the set of unsatisfiable cores found so far with $\mathcal{C} = \{\{r_1, r_2\}, \{r_1, r_3\}, \{r_4, r_5, r_6\}\}$, as represented in Fig. 2i. The inner circles represent clauses, the ellipses represent unsatisfiable cores and the rectangles are the covers. Observe that $\mathcal{C}$ contains 3 cores. The first two cores in $\mathcal{C}$ share the relaxation variable r_1. Hence, $CoversOf(\mathcal{C}) = \{\{r_1, r_2, r_3\}, \{r_4, r_5, r_6\}\}$ returns two covers. The first cover is formed by two cores in $\mathcal{C}$ (i.e., $k = 2$), whereas the second cover is formed by one core in $\mathcal{C}$ (i.e., $k = 1$). As a result, PM2.1 would add two AtMostK constraints $AM = \{r_1 + r_2 + r_3 \leq 2, r_4 + r_5 + r_6 \leq 1\}$.

Example 11 Consider that the instance φ of Example 1 is solved with algorithm PM2.1 (Algorithm 13). Table 10 shows the execution of the algorithm. Note that the core found in iteration #4 is equivalent to the cover of the cores found in iterations #2, #3 and #4. Figure 2ii presents a drawing of the cores and the covers at such execution step.

PM2.1 was extended to handle weighted MaxSAT in the WPM2 algorithm [9]. WPM2 (Algorithm 14) starts by adding a relaxation variable r_i to each soft clause c_i (line 8) and initializes the set of *covers* $SC = \{\{1\}, \ldots, \{m\}\}$ (line 9). The set of *AtLeastK* constraints is empty $AL = \emptyset$ (line 10) and the set of *AtMostK* constraints $AM = \{w_1 r_1 \leq 0, ..., w_m r_m \leq 0\}$ (line 11). Initially, each soft clause represents a cover and the associated AtMostK constraint to each cover has a k with value 0.

The main loop (line 12) iterates until a satisfiable solution is found (line 14). At each iteration, the algorithm calls a SAT solver with the current working formula and the constraints in sets AL and AM (line 13). If it returns unsatisfiable, then the information of the unsatisfiable core obtained by the SAT solver is used to update the sets SC, AL, and AM. Observe that there is one AtMostK constraint for each cover at each iteration, whereas one AtLeastK constraint for each cover is added

Algorithm 15 BIN-C: The core-guided binary search Algorithm [63]

```
   Input: $\varphi$
1  $(R, \varphi_W, \varphi_S) \leftarrow (\emptyset, \varphi, \texttt{Soft}(\varphi))$
2  $(\lambda, \mu, last\mathcal{A}) \leftarrow (-1, 1 + \sum_{i=1}^{m} w_i, \emptyset)$
3  while $\lambda + 1 < \mu$ do
4      $\nu \leftarrow \lfloor \frac{\mu+\lambda}{2} \rfloor$
5      $(\text{st}, \varphi_C, \mathcal{A}) \leftarrow \texttt{SAT}(\texttt{Cls}(\varphi_W) \cup \texttt{CNF}(\sum_{r_i \in R} w_i \cdot r_i \leq \nu))$
6      if st = $SAT$ then $(last\mathcal{A}, \mu) \leftarrow (\mathcal{A}, \sum_{i=1}^{m} w_i \cdot (1 - \mathcal{A}\langle c_i \setminus \{r_i\}\rangle))$
7      else
8          if $\varphi_C \cap \varphi_S = \emptyset$ then $\lambda \leftarrow \texttt{RefineBound}(\{w_i \mid r_i \in R\}, \nu) - 1$
9          else $(R, \varphi_W) \leftarrow \texttt{RelaxCls}(R, \varphi_W, \varphi_C \cap \varphi_S)$
10     end
11 end
12 return $\texttt{Init}(last\mathcal{A})$
```

to the working formula at each iteration. One contribution of WPM2 is the way it computes the bound k associated with each AtMostK and AtLeastK constraints, which is stronger for weighted MaxSAT than just using the subset sum approach. First, the algorithm computes the set of all indexes A of the relaxation variables associated with soft clauses in the unsatisfiable core (line 15). Then, the set of covers in SC that share some variable with A are stored in RC (line 16). The covers in RC should be merged in just one cover. The indexes of the relaxation variables contained in RC are stored in B (line 17). At this point, the algorithm proceeds by computing the value k for the new cover defined over the relaxation variables in B. This is done in function Newbound (line 18).

An initial bound k is computed that is essentially the sum of all k of the previous covers contained in the new one (line 2). Then, the function iteratively refines the value of k until a valid value is obtained given the set of weights. The subset sum is computed with the set of weights of the soft clauses referenced by the indexes in B and the current k (line 4). Then, the SAT solver is called with the set of AtLeastK constraints AL and an equality constraint that states that the weights in B should be equal to k (line 5). The algorithm iterates until the SAT solver returns satisfiable.

WPM2 continues by removing the set of covers in RC from SC and replacing them with the new unique cover defined in B (line 19). Then, the set of AtLeastK constraints AL is augmented with an additional AtLeastK constraint bounded by the variables in B and the obtained k (line 20). Finally, the set of AtMostK constraints related to the covers in RC are removed from AM and a new AtMostK constraint is added again bounded by the variables in B and the obtained k (line 21).

Example 12 Consider that the instance φ of Example 1 is solved with algorithm WPM2 (Algorithm 14). Notice in iteration #4 the core detected intersects with the cores found in iterations #2 and #3. This makes the computation of the bound $k = 3$, which is the number of cores found in the cover $\{1, 2, 3, 4, 5\}$. Table 11 shows the execution of the algorithm.

Table 11 Running example for the WPM2 Algorithm

	$\varphi_S = \{(x_1 \vee r_1), (x_2 \vee r_2), (x_3 \vee r_3), (x_4 \vee r_4), (x_5 \vee r_5), (x_6 \vee r_6), (\neg x_6 \vee r_7)\}$ $\varphi_W = \varphi^S \cup \varphi^H$ $SC = \{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}\}$ $AL = \emptyset$ $AM = \{(r_1 \leq 0), (r_2 \leq 0), (r_3 \leq 0), (r_4 \leq 0), (r_5 \leq 0), (r_6 \leq 0), (r_7 \leq 0)\}$
#1	$st = \text{UNSAT}; \quad Soft(\varphi_C) = \{(x_6), (\neg x_6)\};$ $A = \{6, 7\}$ $RC = \{\{6\}, \{7\}\}$ $B = \{6, 7\}$ $k = 1$ $SC = \{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{\mathbf{6}, \mathbf{7}\}\}$ $AL = \{(\mathbf{r_6} + \mathbf{r_7} \geq \mathbf{1})\}$ $AM = \{(r_1 \leq 0), (r_2 \leq 0), (r_3 \leq 0), (r_4 \leq 0), (r_5 \leq 0), (\mathbf{r_6} + \mathbf{r_7} \leq \mathbf{1})\}$
#2	$st = \text{UNSAT}; \quad Soft(\varphi_C) = \{(x_1), (x_2)\};$ $A = \{1, 2\}$ $RC = \{\{1\}, \{2\}\}$ $B = \{1, 2\}$ $k = 1$ $SC = \{\{3\}, \{4\}, \{5\}, \{6, 7\}, \{\mathbf{1}, \mathbf{2}\}\}$ $AL = \{(r_6 + r_7 \geq 1), (\mathbf{r_1} + \mathbf{r_2} \geq \mathbf{1})\}$ $AM = \{(r_3 \leq 0), (r_4 \leq 0), (r_5 \leq 0), (r_6 + r_7 \leq 1), (\mathbf{r_1} + \mathbf{r_2} \leq \mathbf{1})\}$
#3	$st = \text{UNSAT}; \quad Soft(\varphi_C) = \{(x_3), (x_4)\};$ $A = \{3, 4\}$ $RC = \{\{3\}, \{4\}\}$ $B = \{3, 4\}$ $k = 1$ $SC = \{\{1, 2\}, \{5\}, \{6, 7\}, \{\mathbf{3}, \mathbf{4}\}\}$ $AL = \{(r_6 + r_7 \geq 1), (r_1 + r_2 \geq 1), (\mathbf{r_3} + \mathbf{r_4} \geq \mathbf{1})\}$ $AM = \{(r_1 + r_2 \leq 1), (r_5 \leq 0), (r_6 + r_7 \leq 1), (\mathbf{r_3} + \mathbf{r_4} \leq \mathbf{1})\}$
#4	$st = \text{UNSAT}; \quad Soft(\varphi_C) = \{(x_1), (x_2), (x_3), (x_4), (x_5)\};$ $A = \{1, 2, 3, 4, 5\}$ $RC = \{\{1, 2\}, \{3, 4\}, \{5\}\}$ $B = \{1, 2, 3, 4, 5\}$ $k = 3$ $SC = \{\{6, 7\}, \{\mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5}\}\}$ $AL = \{(r_6 + r_7 \geq 1), (r_1 + r_2 \geq 1), (r_3 + r_4 \geq 1), (\mathbf{r_1} + \mathbf{r_2} + \mathbf{r_3} + \mathbf{r_4} + \mathbf{r_5} \geq \mathbf{3})\}$ $AM = \{(r_6 + r_7 \leq 1), (\mathbf{r_1} + \mathbf{r_2} + \mathbf{r_3} + \mathbf{r_4} + \mathbf{r_5} \leq \mathbf{3})\}$
#5	$st = \text{SAT}; \; Init(\mathcal{A}) = \{x_3 = x_5 = 1; x_1 = x_2 = x_4 = x_6 = 0\}$

#*i* represents the *i*−th iteration of the algorithm

5.3 Binary search based algorithms

This section presents two core-guided MaxSAT algorithms that are based on binary search. First is shown the core-guided binary search algorithm followed by the core-guided binary search with disjoint cores. Similar to the algorithms in the previous section, a maximum of one relaxation variable is added per soft clause relaxed. The difference between the two algorithms has to do with the second algorithm

Table 12 Running example for the core-guided binary search Algorithm

	$R = \emptyset$; $\varphi_W = \varphi$; $\varphi_S = \{(x_1), (x_2), (x_3), (x_4), (x_5), (x_6), (\neg x_6)\}$; $\mu = 8$; $\lambda = -1$; $\mathit{last}\mathcal{A} = \emptyset$;
#1	$\nu = 3$; Constraint to include: $\mathit{CNF}(\emptyset) = \emptyset$; $st = \text{UNSAT}$; $\mathit{Soft}(\varphi_C) = \{(x_6), (\neg x_6)\}$; $R = \{\mathbf{r_1}, \mathbf{r_2}\}$; $\varphi_W = \{(\mathbf{x_6} \vee \mathbf{r_1}), (\neg \mathbf{x_6} \vee \mathbf{r_2}), (x_1), (x_2), (x_3), (x_4), (x_5)\} \cup \varphi^H$;
#2	$\nu = 3$; Constraint to include: $\mathit{CNF}(r_1 + r_2 \leq 3)$; $st = \text{UNSAT}$; $\mathit{Soft}(\varphi_C) = \{(x_1), (x_2)\}$; $R = \{r_1, r_2, \mathbf{r_3}, \mathbf{r_4}\}$; $\varphi_W = \{(x_6 \vee r_1), (\neg x_6 \vee r_2), (\mathbf{x_1} \vee \mathbf{r_3}), (\mathbf{x_2} \vee \mathbf{r_4}), (x_3), (x_4), (x_5)\} \cup \varphi^H$;
#3	$\nu = 3$; Constraint to include: $\mathit{CNF}(r_1 + r_2 + r_3 + r_4 \leq 3)$; $st = \text{UNSAT}$; $\mathit{Soft}(\varphi_C) = \{(x_3), (x_4)\}$; $R = \{r_1, r_2, r_3, r_4, \mathbf{r_5}, \mathbf{r_6}\}$; $\varphi_W = \{(x_6 \vee r_1), (\neg x_6 \vee r_2), (x_1 \vee r_3), (x_2 \vee r_4), (\mathbf{x_3} \vee \mathbf{r_5}), (\mathbf{x_4} \vee \mathbf{r_6}), (x_5)\} \cup \varphi^H$;
#4	$\nu = 3$; Constraint to include: $\mathit{CNF}(r_1 + r_2 + r_3 + r_4 + r_5 + r_6 \leq 3)$; $st = \text{UNSAT}$; $\mathit{Soft}(\varphi_C) = \{(x_6 \vee r_1), (\neg x_6 \vee r_2), (x_1 \vee r_3), (x_2 \vee r_4), (x_3 \vee r_5), (x_4 \vee r_6), (x_5)\}$; $R = \{r_1, r_2, r_3, r_4, r_5, r_6, \mathbf{r_7}\}$; $\varphi_W = \{(x_6 \vee r_1), (\neg x_6 \vee r_2), (x_1 \vee r_3), (x_2 \vee r_4), (x_3 \vee r_5), (x_4 \vee r_6), (\mathbf{x_5} \vee \mathbf{r_7})\} \cup \varphi^H$;
#5	$\nu = 3$; Constraint to include: $\mathit{CNF}(r_1 + r_2 + r_3 + r_4 + r_5 + r_6 + r_7 \leq 3)$; $st = \text{UNSAT}$; $\mathit{Soft}(\varphi_C) = \{(x_6 \vee r_1), (\neg x_6 \vee r_2), (x_1 \vee r_3), (x_2 \vee r_4), (x_3 \vee r_5), (x_4 \vee r_6), (x_5 \vee r_7)\}$; $\lambda = 3$;
#6	$\nu = 5$; Constraint to include: $\mathit{CNF}(r_1 + r_2 + r_3 + r_4 + r_5 + r_6 + r_7 \leq 5)$; $st = \text{SAT}$; $\mathit{last}\mathcal{A} = \mathcal{A} = \{r_2 = r_5 = r_7 = 0; r_1 = r_3 = r_4 = r_6 = 1\} \cup \mathit{Init}(\mathcal{A})$; $\mu = 4$;
	$\mathit{Init}(\mathit{last}\mathcal{A}) = \{x_3 = x_5 = 1; x_1 = x_2 = x_4 = x_6 = 0\}$

#*i* represents the *i*−th iteration of the algorithm

maintaining disjoint parts of the formula disjoint, namely unsatisfiable cores that do not intersect previous discovered cores.

Similar to the (iterative) binary search (Algorithm 5 of Section 4), *core-guided binary search* [63] (Algorithm 15) maintains two bounds, an upper bound μ and a lower bound λ, which are initialized to the sum of weights of soft clauses plus 1 and to −1 (line 2) , respectively. Unlike binary search, core-guided binary search does not add relaxation variables to the soft clauses before starting the main loop (line 3). The algorithm proceeds by iteratively calling a SAT solver with the current formula and with an `AtMostK` constraint considering only the relaxation variables added so far (line 5) and the middle value ν between both bounds (line 4). If the formula is unsatisfiable, it checks whether all soft clauses in the core have been relaxed and λ is updated (line 8). Otherwise, non-relaxed clauses in the core are relaxed (line 9). If the SAT solver returns satisfiable, μ is updated (line 6).

Example 13 Consider that the instance φ of Example 1 is solved with the core-guided binary search algorithm [63] (Algorithm 15). Table 12 shows the execution of the algorithm.

Algorithm 16 BIN-C-D: The core-guided binary search with disjoint cores Algorithm [63]

```
Input: $\varphi$
1  $(\varphi_W, \varphi_S, \mathcal{C}, last\mathcal{A}) \leftarrow (\varphi, \texttt{Soft}(\varphi), \emptyset, \emptyset)$
2  repeat
3      foreach $C_i \in \mathcal{C}$ do $\nu_i \leftarrow (\lambda_i + 1 = \mu_i)?\ \mu_i\ :\ \lfloor \frac{\mu_i + \lambda_i}{2} \rfloor$
4      $(\text{st}, \varphi_C, \mathcal{A}) \leftarrow \texttt{SAT}(\texttt{Cls}(\varphi_W) \cup \bigcup_{C_i \in \mathcal{C}} \texttt{CNF}(\sum_{r_j \in R_i} w_j \cdot r_j \leq \nu_i))$
5      if st = $SAT$ then
6          $last\mathcal{A} \leftarrow \mathcal{A}$
7          foreach $C_i \in \mathcal{C}$ do $\mu_i \leftarrow \sum_{r_j \in R_i} w_j \cdot (1 - \mathcal{A}\langle c_j \setminus \{r_j\}\rangle)$
8      else
9          $sub\mathcal{C} \leftarrow \texttt{IntersectingCores}(\varphi_C, \mathcal{C})$
10         if $\varphi_C \cap \varphi_S = \emptyset$ and $|sub\mathcal{C}| = 1$ then $\lambda \leftarrow \nu$    /* $sub\mathcal{C} = \{< R, \lambda, \nu, \mu >\}$ */
11         else
12             $(R, \varphi_W) \leftarrow \texttt{RelaxCls}(\emptyset, \varphi_W, \varphi_C \cap \varphi_S)$
13             $(\lambda, \mu) \leftarrow (0, 1 + \sum_{r_j \in R} w_j)$
14             foreach $C_i \in sub\mathcal{C}$ do $(R, \lambda, \mu) \leftarrow (R \cup R_i, \lambda + \lambda_i, \mu + \mu_i)$
15             $\mathcal{C} \leftarrow \mathcal{C} \setminus sub\mathcal{C} \cup \{< R, \lambda, 0, \mu >\}$
16         end
17     end
18 until $\forall_{C_i \in \mathcal{C}}\ \lambda_i + 1 \geq \mu_i$
19 return $\texttt{Init}(last\mathcal{A})$
```

Core-guided binary search was extended to maintain *disjoint cores* [63]. The resulting algorithm is referred to as *core-guided binary search with disjoint cores*. The concept of *disjoint core* [63] is essentially *equivalent* to the concept of *cover* [7]. Let $\mathcal{U} = \{U_1, \ldots, U_k\}$ be a set of cores, and each core U_i has a set of relaxation variables R_i. A core $U_i \in \mathcal{U}$ is *disjoint* when $\forall_{U_j \in \mathcal{U}}(R_i \cap R_j = \emptyset \wedge i \neq j)$. The goal of *core-guided binary search with disjoint cores* [63] (Algorithm 16) is to maintain smaller lower and upper bounds for each disjoint core rather than just one global lower bound and one global upper bound. As a result, the algorithm will add several smaller `AtMostK` constraints rather than just one global `AtMostK` constraint.

The algorithm maintains information about the previous cores in a set $\mathcal{C}$ which is initially empty (line 1). Whenever a new core i is found, a new entry in $\mathcal{C}$ is created containing: the set of relaxation variables R_i in the core (after relaxing the required soft clauses), a lower bound λ_i, an upper bound μ_i, and the current middle value ν_i, i.e. $C_i =< R_i, \lambda_i, \nu_i, \mu_i >$. The algorithm iterates while there exists a C_i for which $\lambda_i + 1 < \mu_i$ (line 2). Before calling the SAT solver (line 4), for each disjoint core $C_i \in \mathcal{C}$, its middle value ν_i is computed with the current bounds (line 3) and an `AtMostK` constraint is added to the working formula (line 4). If the SAT solver returns false, then $sub\mathcal{C}$ is computed with *IntersectingCores*() (line 9), and contains every C_i in $\mathcal{C}$ that intersects the current core (i.e., $sub\mathcal{C} \subseteq \mathcal{C}$). If no soft clause needs to be relaxed and $|sub\mathcal{C}| = 1$, then $sub\mathcal{C} = \{< R, \lambda, \nu, \mu >\}$ and λ is updated to ν (line 10). Otherwise, all the required soft clauses are relaxed (line 12), and an entry for the new core is added to $\mathcal{C}$, which aggregates the information of the previous cores in $sub\mathcal{C}$ (lines 13 and 14). Also, each $C_i \in sub\mathcal{C}$ is removed from $\mathcal{C}$ (line 15). If the SAT solver returns true, the algorithm iterates over each disjoint core $C_i \in \mathcal{C}$ and its upper bound μ_i is updated according to the satisfying assignment $\mathcal{A}$ (line 7).

Example 14 Consider that the instance φ of Example 1 is given to *core-guided binary search with disjoint cores* (Algorithm 16). Table 13 shows the execution of the algorithm.

5.4 Characterization of the algorithms

Table 14 presents a characterization of the algorithms presented. The characterization of the iterative algorithms is repeated for a better comparison and a better readability. The first column enumerates several characteristics, which are as follows:

- Progression: indicates if the algorithm refines a lower bound (LB) or an upper bound (UB).
- # Relax. Vars./Clause: The number of relaxation variables per soft clause.
- Total # Relax. Vars.: The total number of relaxation variables added to the formula throughout the algorithm.
- # Const./Iteration: The number of constraints added at each iteration.
- Total # Const.: The total number of constraints added to the formula throughout the algorithm.
- Calls SAT Oracle: The theoretical worst case number of calls to a SAT oracle (solver).
- Weighted: If the algorithm handles weighted MaxSAT.

The second and remaining columns of Table 14 refer to the iterative MaxSAT algorithms: `LIN-US`, `LIN-SU`, `BIN`, `BIN/LIN-SU`, `BIT`, followed by the core-guided MaxSAT algorithms: `WMSU1`, `MSU2`, `WMSU3`, `WMSU4`, `PM2`, `PM2.1`, `WPM2`, core-guided binary search as `BIN-C` and core-guided binary search with disjoint cores as `BIN-C-D`. For an explanation on the iterative MaxSAT algorithms, we refer to Table 6 on page 20.

Recall that m is the total number of soft clauses in $\varphi = \varphi_S \cup \varphi_H$ (i.e., $m = |\varphi_S|$) and W be the sum of weights of soft clauses $W = \sum_{i=1}^{m} w_i$. First, observe that except `MSU2`, `PM2`, and `PM2.1`, the algorithms are presented in their weighted MaxSAT version. `WMSU1`, `MSU2`, `WMSU3`, `PM2`, `PM2.1` and `WPM2` all refine a lower bound. Differently, `WMSU4` alternates the refinement of a lower bound and an upper bound. *Core-guided binary search* and *core-guided binary search with disjoint cores* both compute the middle value between the lower bound and upper bound.

`PM2`, `PM2.1`, and `WPM2` add exactly one relaxation variable per soft clause and exactly a total of m relaxation variables. `WMSU3`, `WMSU4`, *core-guided binary search* and *core-guided binary search with disjoint cores* relax variables on demand, so they may add one relaxation variable or none to each soft clause and a total of m in the worst case (i.e., all soft clauses are relaxed). `MSU2` requires $\log(m)$ relaxation variables per soft clause and a total of $O(m \log(m))$ relaxation variables throughout the algorithm. `WMSU1` is the only algorithm that adds more than one relaxation variable per soft clause. In the worst case, each soft clause can be relaxed W times. The total number of relaxation variables γ for `WMSU1` is detailed in the Appendix.

`WMSU1` adds one `AtMost1` constraint at each iteration and a total of W in the worse case. `MSU2` has no constraints; it just adds relaxation variables. `WMSU3`, `WMSU4`,

Table 13 Running example for *core-guided binary search with disjoint cores*

	$\varphi_W = \varphi$; $\varphi_S = \{(x_1), (x_2), (x_3), (x_4), (x_5), (x_6), (\neg x_6)\}$; $\mathcal{C} = \emptyset$; $last\mathcal{A} = \emptyset$;
#1	Constraints to include: $\emptyset$ $st =$ UNSAT; $Soft(\varphi_C) = \{(x_6), (\neg x_6)\}$ $sub\mathcal{C} = \emptyset$ $R = \{r_1, r_2\}$ $\varphi_W = \{(\mathbf{x_6} \vee \mathbf{r_1}), (\neg \mathbf{x_6} \vee \mathbf{r_2}), (x_1), (x_2), (x_3), (x_4), (x_5)\} \cup \varphi^H$ $\lambda = 0$ $\mu = 3$ $\mathcal{C} = \{< \{\mathbf{r_1}, \mathbf{r_2}\}, \mathbf{0}, \mathbf{0}, \mathbf{3} >\}$
#2	Constraints to include: $\{(r_1 + r_2 \leq 1)\}$ $st =$ UNSAT; $Soft(\varphi_C) = \{(x_1), (x_2)\}$ $sub\mathcal{C} = \emptyset$ $R = \{r_3, r_4\}$ $\varphi_W = \{(x_6 \vee r_1), (\neg x_6 \vee r_2), (\mathbf{x_1} \vee \mathbf{r_3}), (\mathbf{x_2} \vee \mathbf{r_4}), (x_3), (x_4), (x_5)\} \cup \varphi^H$ $\lambda = 0$ $\mu = 3$ $\mathcal{C} = \{< \{r_1, r_2\}, 0, 0, 3 >, < \{\mathbf{r_3}, \mathbf{r_4}\}, \mathbf{0}, \mathbf{0}, \mathbf{3} >\}$
#3	Constraints to include: $\{(r_1 + r_2 \leq 1), (r_3 + r_4 \leq 1)\}$ $st =$ UNSAT; $Soft(\varphi_C) = \{(x_3), (x_4)\}$ $sub\mathcal{C} = \emptyset$ $R = \{r_5, r_6\}$ $\varphi_W = \{(x_6 \vee r_1), (\neg x_6 \vee r_2), (x_1 \vee r_3), (x_2 \vee r_4), (\mathbf{x_3} \vee \mathbf{r_5}), (\mathbf{x_4} \vee \mathbf{r_6}), (x_5)\} \cup \varphi^H$ $\lambda = 0$ $\mu = 3$ $\mathcal{C} = \{< \{r_1, r_2\}, 0, 0, 3 >, < \{r_3, r_4\}, 0, 0, 3 >, < \{\mathbf{r_5}, \mathbf{r_6}\}, \mathbf{0}, \mathbf{0}, \mathbf{3} >\}$
#4	Constraints to include: $\{(r_1 + r_2 \leq 1), (r_3 + r_4 \leq 1), (r_5 + r_6 \leq 1)\}$ $st =$ UNSAT; $Soft(\varphi_C) = \{(x_1 \vee r_3), (x_2 \vee r_4), (x_3 \vee r_5), (x_4 \vee r_6), (x_5)\}$ $sub\mathcal{C} = \{< \{r_3, r_4\}, 0, 0, 3 >, < \{r_5, r_6\}, 0, 0, 3 >\}$ $R = \{r_3, r_4, r_5, r_6, \mathbf{r_7}\}$ $\varphi_W = \{(x_6 \vee r_1), (\neg x_6 \vee r_2), (x_1 \vee r_3), (x_2 \vee r_4), (x_3 \vee r_5), (x_4 \vee r_6), (\mathbf{x_5} \vee \mathbf{r_7})\} \cup \varphi^H$ $\lambda = 0$ $\mu = 8$ $\mathcal{C} = \{< \{r_1, r_2\}, 0, 0, 3 >, < \{\mathbf{r_3}, \mathbf{r_4}, \mathbf{r_5}, \mathbf{r_6}, \mathbf{r_7}\}, \mathbf{0}, \mathbf{0}, \mathbf{8} >\}$
#5	Constraints to include: $\{(r_1 + r_2 \leq 1), (r_3 + r_4 + r_5 + r_6 + r_7 \leq 4)\}$ $st =$ SAT; $\mathcal{A} \setminus Init(\mathcal{A}) = \{r_2 = r_3 = r_5 = r_7 = 0; r_1 = r_4 = r_6 = 1\}$ $\mathcal{C} = \{< \{\mathbf{r_1}, \mathbf{r_2}\}, \mathbf{0}, \mathbf{0}, \mathbf{1} >, < \{\mathbf{r_3}, \mathbf{r_4}, \mathbf{r_5}, \mathbf{r_6}, \mathbf{r_7}\}, \mathbf{0}, \mathbf{0}, \mathbf{2} >\}$
#6	Constraints to include: $\{(r_1 + r_2 \leq 0), (r_3 + r_4 + r_5 + r_6 + r_7 \leq 1)\}$ $st =$ UNSAT; $Soft(\varphi_C) = \{(x_1 \vee r_3), (x_2 \vee r_4), (x_3 \vee r_5), (x_4 \vee r_6), (x_5 \vee r_7)\}$ $sub\mathcal{C} = < \{r_3, r_4, r_5, r_6, r_7\}, 0, 0, 2 >$ $\mathcal{C} = \{< \{r_1, r_2\}, 0, 0, 1 >, < \{\mathbf{r_3}, \mathbf{r_4}, \mathbf{r_5}, \mathbf{r_6}, \mathbf{r_7}\}, \mathbf{1}, \mathbf{0}, \mathbf{2} >\}$
#7	$st =$ SAT; $Init(\mathcal{A}) = \{x_3 = x_5 = 1; x_1 = x_2 = x_4 = x_6 = 0\}$

#*i* represents the *i*−th iteration of the algorithm

PM2, and BIN-C add (and remove) one AtMostK constraint at each iteration. PM2.1, WPM2, and BIN-C-D add (and remove) at most *m* AtMostK constraints at each iteration, exactly one for each cover/disjoint core. Additionally, PM2, PM2.1, and WPM2 add one AtLeastK constraint at each iteration. All of the algorithms

Table 14 Characteristics of iterative and core-guided MaxSAT Algorithms

Charact.	LIN-US	LIN-SU	BIN	BIN/LIN-SU	BIT	WMSU1	MSU2	WMSU3	WMSU4	PM2	PM2.1	WPM2	BIN-C	BIN-C-D
Progression	LB	UB	LB+UB	LB+UB	LB+UB	LB	LB	LB	LB+UB	LB	LB	LB	LB+UB	LB+UB
# Relax. Vars./ Clause	1	1	1	1	1	$O(W)$	$O(\log(m))$	1 or 0	1 or 0	1	1	1	1 or 0	1 or 0
Total # Relax. Vars.	m	m	m	m	m	γ	$O(m\log(m))$	$O(m)$	$O(m)$	m	m	m	$O(m)$	$O(m)$
# Const./ Iteration	1	1	1	1	1	1	0	1	1	$1+1$	$1+O(m)$	$1+O(m)$	1	$O(m)$
Total # Const.	1	1	1	1	1	$O(W)$	0	1	1	$1+O(m)$	$O(m)+O(m)$	$O(W)+O(m)$	1	$O(m)$
Calls SAT Oracle	$O(W)$	$O(W)$	$O(\log(W))$	$O(\log(W))$	$O(\log(W))$	$O(W)$	$O(m)$	$O(W)$	$O(W)$	$O(m)$	$O(m)$	$O(W)$	$m+O(\log(W))$	$m+O(\log(W))$
Weighted	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	No	No	Yes	Yes	Yes

m is the number of *soft* clauses of φ and W is the sum of weights of soft clauses $W=\sum_{i=1}^{m} w_i$

require an exponential number of calls to the SAT solver, except `BIN-C` and `BIN-C-D` which require a linear number of calls.

6 MaxSMT: a MaxSAT generalization for SMT

The *Satisfiability Modulo Theories* problem, or SMT, is an extension of Boolean Satisfiability where the goal is to check the satisfiability of an SMT formula with respect to a background theory $\mathcal{T}$ [22, 97, 113]. As such, an SMT formula is allowed to have *atomic formulas* or predicates in higher-level *theories*, such as linear arithmetic or a higher-order logic, in addition to Boolean variables and their negations. As an example, the following SMT formula mixes Boolean literals and linear inequalities: $(a \vee (5x - y \leq 3)) \wedge (\neg a \vee (x + y \leq 12))$.

Algorithms for solving SMT problems often have a core engine that is similar to a modern SAT solver. The core engine then employs additional theory solvers that can process conjunctions of literals over the given theories (every SMT solver handles some subset of possible theories). Further details on SMT, theories, and SMT solving can be obtained in [97, 113].

The *MaxSMT* problem is a generalization of MaxSAT to the SMT domain. Given an SMT formula φ, MaxSMT is the problem of finding a model $\mathcal{A}$, consistent with the theories in φ, that maximizes the number of satisfied clauses in φ. As in SAT, weights can be given to every clause in a SMT formula, and *Weighted Partial* MaxSMT follows directly from its SAT equivalent.

Given the similarities between SAT and SMT, many algorithms for MaxSAT can be adapted to the MaxSMT problem, and existing MaxSMT algorithms closely resemble the MaxSAT algorithms presented in this survey.

The first work to address MaxSMT can be found in [96]. In this work, SMT is extended to allow theories to be *strengthened*, essentially letting the context in which a formula is evaluated change without altering the formula itself. Then, an approach is proposed using their framework to solve weighted MaxSMT by placing information about an objective function, and a bound on it, in the theory solver itself. In this approach, initially every weighted clause (c_i, w_i) receives a new Boolean variable p_i, and the constraints $(p_i \rightarrow (k_i = w_i))$, and $(\neg p_i \rightarrow (k_i = 0))$ are added to the theory. Further, the constraint $(k_1 + \ldots + k_m \leq B)$ is added to the theory together with the relation $(B < B_0)$ (where B_0 is an estimation of the initial cost). Each time a new cost B_i is found, the theory is strengthened by adding the relation $(B < B_i)$ to the theory. In this way, progressively better solutions are found, minimizing the total cost of unsatisfied clauses by making the theory solver reject models worse than the best known thus far. The final model found will be a MaxSMT solution. Of the MaxSAT algorithms presented earlier, this algorithm is similar to the Linear Search Sat-Unsat algorithm; both explore a solution space of models that satisfy some subset of clauses, requiring ever lower costs for those clauses not satisfied by the models, until no further models can be found. The related Linear Search Unsat-Sat algorithm could not be applied in this extended SMT framework directly, as it requires *relaxing* the constraint that bounds the cost of unsatisfied clauses in each iteration, and the framework only allows theories to be strengthened.

The work in [35] proposed a "theory of *Costs*" $\mathcal{C}$, along with a decision procedure for it, that allows modeling multiple cost functions within the SMT framework. In [35], the theory of costs $\mathcal{C}$ is used to address the problem of minimizing the value of one cost function subject to the satisfaction of an SMT formula named as the *Boolean Optimization Modulo Theory* (BOMT) problem. The optimization itself can be performed by asserting atoms of $\mathcal{C}$ that bound one cost function and using an incremental SMT solver.

Essentially, [35] proposed to encode the weighted partial MaxSMT problem into BOMT by adding a new Boolean variable A^i_j to each soft clause. Then, the cost function is the sum of the weights of the soft clauses whose variable A^i_j is assigned true. Again, this is similar to existing MaxSAT approaches that use relaxation variables, and so an algorithm similar to Linear Search Sat-Unsat is described in such paper as well as a Binary Search. While this is similar to the early work in [96], it differs in that it allows for multiple cost functions and in that it does not require strengthening the theory to update cost bounds.

While both of the above techniques require a specialized SMT solver, many MaxSAT algorithms that use a SAT solver as a black box can be made MaxSMT algorithms by substituting a black box SMT solver instead. Furthermore, if the SMT solver can produce unsatisfiable cores, the core-guided MaxSAT algorithms can be so adapted as well. In this way, the full range of MaxSAT algorithms overviewed in this survey can become MaxSMT algorithms. The Z3 SMT solver [43] has been recently extended to handle (unweighted) MaxSMT in this way, and it currently implements MSU1 and Linear Search Sat-Unsat algorithms for MaxSMT.

Note that while the MaxSAT algorithms make use of cardinality constraints encoded to Boolean CNF, SMT allows for cardinality constraints in other forms. A constraint of the form $\sum_{i=1}^{m} w_i \cdot r_i \leq k$ is an atomic formula in the theory of linear integer arithmetic, and so it can be included *natively*, with no translation, given an SMT solver with that theory. This may provide better performance than a CNF encoding of the same constraint, though this will depend on the SMT solver used.

7 Experimental analysis

This section presents an empirical evaluation of the algorithms described in the paper. The objective of the experimental analysis is to evaluate the performance of all the algorithms under the same implementation framework, with the same internal data structures, the same encodings for cardinality and pseudo-Boolean constraints, and the same underlying SAT solver. The purpose of the experiment is to understand which are the most effective algorithms without the influence of implementation details, which can have a an impact on the performance of the resulting algorithm.

Based on the results of the experimental evaluation, an analysis is presented in Section 7.1, to evaluate the performance of the algorithms, but taking in consideration the optimum value of the instances with respect to the sum of weights of the soft clauses.

For the experimental evaluation, all the algorithms described have been implemented in the MSUnCore system [91]. Additionally, the weighted version of several algorithms are evaluated for the first time. All non-random instances from the

MaxSAT Evaluations[3] between 2009 and 2011, inclusive, have been considered. The instances are classified in 8 sets depending on whether they are unweighted/weighted, partial/non-partial, crafted/industrial or an aggregation of all the instances. The classes of instances are referred to as follows:

- MS Crafted: plain MaxSAT crafted instances
- MS Industrial: plain MaxSAT industrial instances
- PMS Crafted: (unweighted) partial MaxSAT crafted instances
- PMS Industrial: (unweighted) partial MaxSAT industrial instances
- WMS Crafted: weighted (non-partial) MaxSAT crafted instances
- WPMS Crafted: weighted partial MaxSAT crafted instances
- WPMS Industrial: weighted partial MaxSAT industrial instances
- All: aggregation of all the non-random MaxSAT Evaluation instances 2009–2011

Experiments were conducted on a HPC cluster with 50 nodes; each node contains a Xeon E5450 3GHz CPU and 32GB RAM and runs Linux. For each run, the time limit was set to 1800 seconds, and the memory limit was set to 4GB.

The empirical results are presented in *cactus* plots that show the total number of solved instances and the run time of the algorithms implemented in MSUnCore system. The plots also contain results for several publicly available MaxSAT solvers that participated in recent MaxSAT Evaluations, referred to as *other tools*. The plots are presented in five figures depending on whether the plots refer to plain MaxSAT, partial MaxSAT, weighted MaxSAT, weighted partial MaxSAT or an aggregation of all instances. Each figure shows on its left hand side the cactus plots of the algorithms implemented in MSUnCore, while the right hand side shows cactus plots of the three best algorithms implemented in MSUnCore together with the *other tools*. On the top-left corner of each figure, is presented a legend of all the solvers considered for the figure. The solvers are ordered in decreasing order of number of solved instances.

The algorithms implemented in the MSUnCore system are:

- Linear Search Unsat-Sat (Algorithm 2): `LIN-US`
- Linear Search Sat-Unsat (Algorithm 4): `LIN-SU`
- Binary Search (Algorithm 5): `BIN`
- Alternating Binary Search with Linear Search Sat-Unsat (Algorithm 6): `BIN/LIN-SU`
- Bit-Based Search (Algorithm 7): `BIT`
- `WMSU1` (Algorithm 8)
- `WMSU3` (Algorithm 10)
- `WMSU4` (Algorithm 11)
- `PM2` (Algorithm 12)
- `PM2.1` (Algorithm 13)
- `WPM2` (Algorithm 14)
- Core-Guided Binary Search (Algorithm 15): `BIN-C`
- Core-Guided Binary Search with Disjoint Cores (Algorithm 16): `BIN-C-D`

All algorithms have been implemented in C++ inside the MSUnCore system [91] which uses PICOSAT [26] as its underlying *non-incremental* SAT solver, with the

[3] http://www.maxsat.udl.cat

exception of MSU2. Note that MSU2 is based on a specific encoding of the AtMostK constraints, it is unclear if can be extended to handle weighted MaxSAT and was not competitive in practice [85]. For those reasons, it has not been considered to be implemented. Note also that picosat was built to be an efficient SAT solver for "application/industrial" benchmarks, which are the type of target instances in this survey. All algorithms use cardinality constraints and pseudo-Boolean constraints to represent AtLeastK and AtMostK constraints, except WMSU1. For all these algorithms *cardinality networks* [19] are used to encode cardinality constraints, and *BDDs* [45] are used to encode pseudo-Boolean constraints. For WMSU1, only AtMost1 constraints are needed, which are translated into clauses using the *bitwise* encoding [105]. Those encodings were selected based on the results of a recent evaluation of encodings available at [91]. Additionally, all the algorithms compute an initial lower bound and upper bound as suggested in [63]. The initial bounds allow them to save a remarkable number of iterations, specially for algorithms that may require an exponential number of calls to a SAT oracle in the worst case.

The *other tools* are the *original* implementation of some of the algorithms described in this paper. In order to distinguish the implementations in MSUnCore from the original tools, the algorithms in MSUnCore are all tagged with (MSUC). The remaining tools implement state-of-the-art algorithms based on a *branch and bound scheme* or on translating the MaxSAT instance to a different optimization framework and apply a native solver for the specific framework (i.e., *PBO* and *ASP*). In particular, the tools considered are:

- The original WPM1 solver that uses the *regular encoding* [11] for the AtMost1 constraints (MaxSAT 2011 Evaluation version).
- The original PM2.1 solver. It is restricted to unweighted plain and partial MaxSAT. Given that the original PM2.1 solver was implemented on top of the original PM2 solver, the latter is no longer available. It uses the *sequential counters* for the AtMostK and AtLeastK constraints.
- The original WPM2 solver that uses different encodings for cardinality and pseudo-Boolean constraints inherited from MINISAT+ [45].
- The original QMAXSAT 0.11 solver, which applies a Linear Search Sat-Unsat, and QMAXSAT 0.4, which alternates Binary Search and Linear Search Sat-Unsat. The QMAXSAT solver is restricted to unweighted plain and partial MaxSAT. The encoding for the AtMostK constraints corresponds to [20].
- SAT4JMAXSAT and PWBO translate the MaxSAT problem to a *Pseudo-Boolean Optimization Problem* (*PBO*). They make use of PB constraints and handle them natively (they are not encoded into clauses). SAT4JMAXSAT [25] follows a Linear Search Sat-Unsat scheme. PWBO [83] allows the execution of multiple threads in parallel (in the experiments the default of two threads was used). In particular, one computes an upper bound using a Linear Search Sat-Unsat approach and the other refines a lower bound using a WMSU1-like approach.
- CLASPMAXSAT [4, 48] translates the MaxSAT problem into the MaxASP problem and solves it with a MaxASP solver.
- 3 branch and bound (BB) MaxSAT algorithms that apply equivalence preserving transformations and compute lower bounds and underestimations using unit propagation. The considered BB algorithms are AKMAXSAT [70], MINIMAXSAT [61], which additionally applies clause learning on hard clauses, and INCWMAXSATZ [79], which *incrementally* computes the underestimations.

Recall that both original versions of QMAXSAT (0.11 and 0.4), PM2, and PM2.1 do not handle weighted MaxSAT. Hence, they will not appear on comparisons including weighted MaxSAT benchmarks.

Figure 3 presents 4 cactus plots for unweighted plain crafted and industrial MaxSAT instances. The performance of iterative and core-guided MaxSAT algorithms is quite poor on plain crafted MaxSAT instances as shown in Fig. 3a. No algorithm solves more than 9 instances. The same pattern is observed in Fig. 3b with the original tools implementing iterative and core-guided MaxSAT algorithms, whereas branch and bound algorithms (AKMAXSAT, INCWMAXSATZ and MINIMAXSAT) are the best option for those benchmarks, being able to solve 140 or more instances.

The results for plain MaxSAT industrial instances can be found in Fig. 3c. In general, iterative algorithms solve significantly fewer instances than core-guided MaxSAT algorithms. In particular, bit-based search (BIT) is the best performing iterative algorithm, solving only 30 instances. WPM2, WMSU4, BIN-C, BIN-C-D, WMSU3 and PM2 (ordered from worst to best performing algorithm) solve from 64 (WPM2) to 92 (PM2.1) instances. Finally, WMSU1 is the best algorithm solving 105 instances. The results including the other tools are presented in Fig. 3d. Core-guided MaxSAT algorithms are the best option in this category. WMSU1 (MSUC) and WPM1 which are different implementations of the same algorithm, are the best performing solvers (105 and 102 solved instances, respectively). Then, the original tool PM2.1 and its implementation in MSUnCore solve 95 and 92 instances, respectively. The remaining algorithms solve fewer instances, but most of them solve at least 70 instances. The only exceptions are branch and bound algorithms, none of which solve more than 4 instances, and SAT4JMAXSAT that solves 28 instances.

Figure 4 shows 4 cactus plots for unweighted partial crafted and industrial MaxSAT instances. The cactus plot on Fig. 4a shows the performance of iterative and core-guided MaxSAT algorithms on partial crafted instances. BIN-C is the best performing algorithm (266 instances solved) followed by BIT (261), BIN (261), BIN/LIN-SU (260), BIN-C-D and WMSU3 (both 255), PM2.1 and LIN-US (both 250) and WMSU4 (247). Interestingly, iterative algorithms such as bit-based (BIT) and binary search (BIN) perform better than several core-guided MaxSAT algorithms which can be explained by their linear number of calls to a SAT oracle in the worst case. LIN-SU (185) is slightly better than WPM2 (165) and PM2 (124), whereas WMSU1 is the worst algorithm solving only 58 instances. The cactus plot on Fig. 4b includes the original tools. Branch and bound algorithms seem more appropriate for those benchmarks with MINIMAXSAT, AKMAXSAT, and INCWMAXSATZ solving 305, 285, and 283 instances, respectively. However, the two different versions of QMAXSAT 0.11 and 0.4 are quite competitive, ranking second (295) and third (294). The 3 best performing algorithms in MSUnCore solve more instances than the remaining algorithms, with an WMSU1-like algorithm (WPM1) again being the worst option.

The cactus plot on Fig. 4c shows the results for iterative and core-guided MaxSAT algorithms on partial industrial MaxSAT instances. WMSU1 is again the worse algorithm and only solves 564 instances. PM2 and WPM2 perform slightly better than WMSU1 but are still far from the best performing algorithms, BIN-C-D (882) and BIN-C (854). The remaining algorithms solve fewer instances than BIN-C but are quite close. The cactus plot on Fig. 4d shows the results including the original tools. BIN-C-D (MSUC) is still the best algorithm, and BIN-C (MSUC) is the third best option, only behind the original QMAXSAT 0.11 solver. Branch and bound algorithms

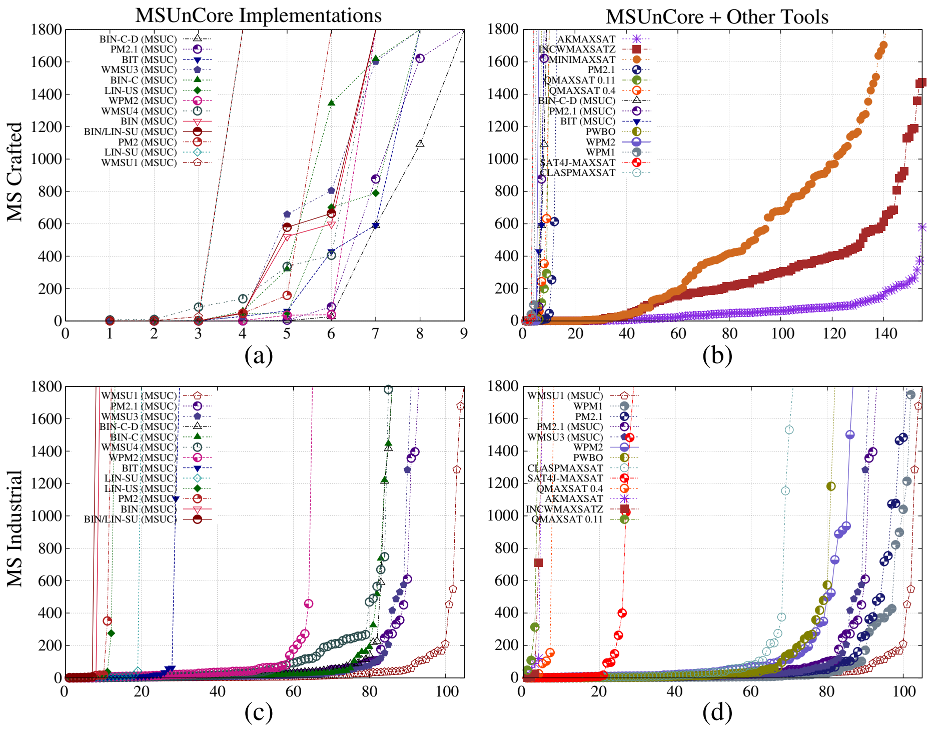

Fig. 3 Run time distributions for MaxSAT instances

perform poorly, and WPM1 (583) is the only core-guided MaxSAT algorithm with similar poor performance.

Figure 5 introduces 2 cactus plots for weighted crafted MaxSAT instances. The results for iterative and core-guided MaxSAT algorithms can be found in Fig. 5a. The performance of all the algorithms is quite similar. Most of the algorithms solve between 60 and 37 instances, with BIT solving 60 instances and WMSU1 solving 37. Figure 5b analyzes the results including the other tools. Similarly to plain crafted MaxSAT, the best performing algorithms are based on branch and bound (AKMAXSAT,INCWMAXSATZ and MINIMAXSAT) and solve over 110 instances. The remaining solvers solve between 40 (WPM1) and 60 (BIT (MSUC)) instances.

Figure 6 presents the results for weighted partial crafted and industrial MaxSAT instances. The cactus plot in Fig. 6a presents the results for iterative and core-guided MaxSAT algorithms for weighted partial crafted instances. Most of the algorithms are able to solve around 350 instances (BIN-C-D, BIT, BIN/LIN-SU, BIN-C, BIN and WMSU4). In particular, BIN-C-D is the best algorithm (364). WMSU1 is again the worst performing algorithm (225), followed by LIN-US (262), WPM2 (260), WMSU3 (263), and LIN-SU (337). The experiments including the other tools are presented in Fig. 6b. The results indicate that branch and bound solvers are the best option for those benchmarks with MINIMAXSAT (448), INCWMAXSATZ (448) and AKMAXSAT (433). CLASPMAXSAT and SAT4JMAXSAT are the next best performing

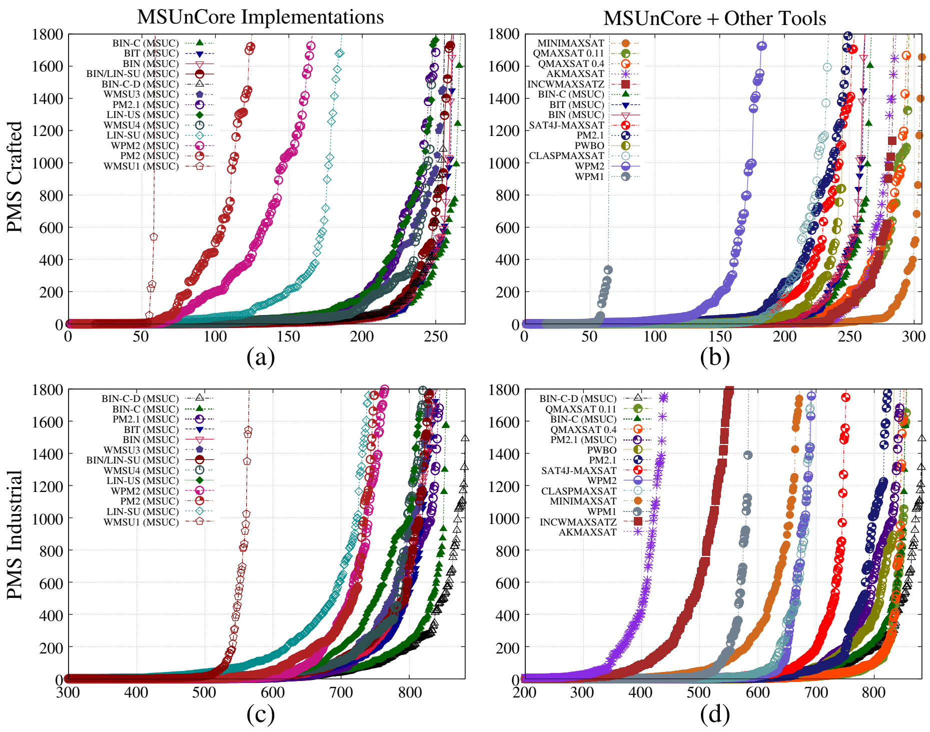

Fig. 4 Run time distributions for partial MaxSAT instances

solvers, and surprisingly WPM1 solves 396 instances. This could be explained by the *stratified* approach included in the considered version [6]. BIN-C-D (MSUC) and BIT (MSUC) are far from the best performing solvers but are still much better than the original WPM2 (276).

The cactus plot in Fig. 6c presents the results for iterative and core-guided MaxSAT algorithms for weighted partial industrial instances. WMSU1 is the best

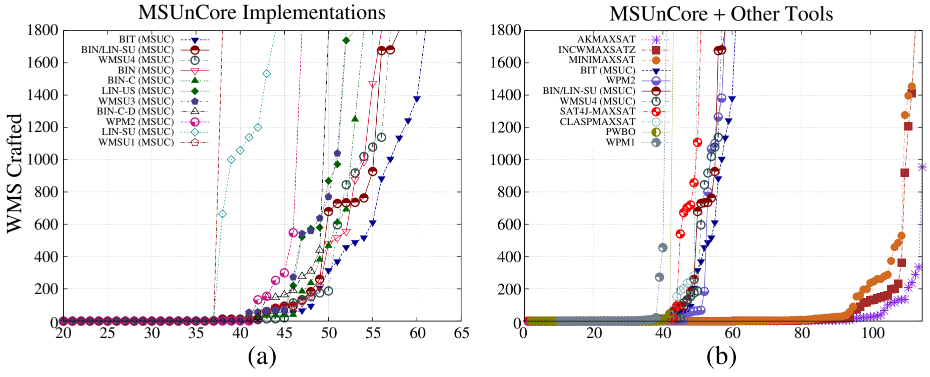

Fig. 5 Run time distributions for weighted MaxSAT crafted instances

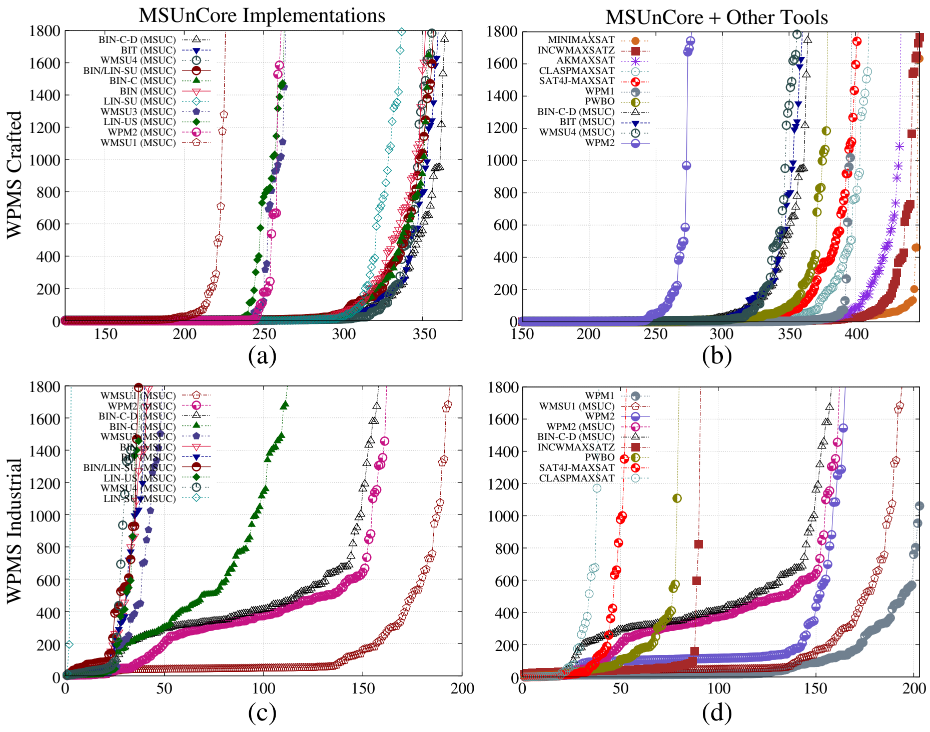

Fig. 6 Run time distributions for weighted partial MaxSAT instances

approach for these benchmarks and solves 193 instances. The next best algorithms are core-guided MaxSAT algorithms including WPM2 (161), BIN-C-D (157) and BIN-C (111). The remaining core-guided and iterative MaxSAT algorithms perform poorly. The results in Fig. 6d present the performance of the remaining tools. Again, WMSU1 (MSUC) and WPM1 are the best approaches, followed by WPM2/WPM2 (MSUC) and BIN-C-D (MSUC) as in the previous plot. The remaining approaches including branch and bound algorithms, PBO, and ASP tools are quite far from the best performing algorithms.

Figure 7 introduces 2 cactus plots aggregating all of the instances considered. The cactus plot in Fig. 7a shows the results for iterative and core-guided MaxSAT algorithms. The best overall performing algorithm is BIN-C-D followed by BIN-C. The third best solver is WMSU4 and it is followed by other algorithms which compute a middle value between a lower bound and upper bound. Algorithms based on exclusively refining a lower bound or an upper bound perform worse. Whereas WMSU1 is the overall worst performing solver, recall that it is in fact the best approach in two industrial categories.

The cactus plot in Fig. 7b shows the results for the other tools. Again, BIN-C-D (MSUC) and BIN-C (MSUC) are the overall best algorithms. From these results, several conclusions can be extracted. First, core-guided MaxSAT algorithms perform better than classic iterative algorithms. Branch and bound algorithms are more appropriate for crafted instances, specially on non-partial benchmarks. However, iterative and core-guided MaxSAT algorithms also achieve very good performance on

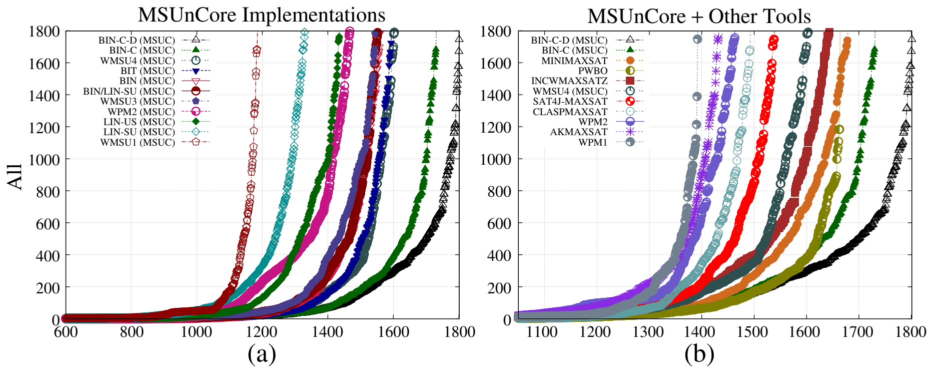

Fig. 7 All: run time distributions for all non-random MaxSAT instances

partial crafted instances. For industrial instances, core-guided MaxSAT algorithms are the most effective; WMSU1-like algorithms are the best option for unweighted plain and weighted partial MaxSAT instances, and core-guided binary search with disjoint cores (BIN-C-D) is the best approach for unweighted partial MaxSAT. Finally, the most robust approach overall is BIN-C-D.

7.1 Analysis of optimum vs sum of weights of soft clauses

The previous evaluation does not take into consideration the optimum value with regards to the sum of weights of the soft clauses. Taking such consideration in account, it can be expected that if the optimum value is close to 0, then algorithms that are based on refining a lower bound potentially have fewer iterations, and algorithms based on refining an upper bound may have to search through a higher number of satisfying assignments before reporting the optimum. On the other side of the spectrum, if the optimum value is closer to the sum of weights of soft clauses, then algorithms based on refining a lower bound may have to perform a higher number of iterations until the optimum is found, and algorithms based on refining an upper bound may have a better performance.

Based on the previous observation and on the results of the previous section, all the instances solved by any of the implementations in the MSUnCore system were collected. The reason for collecting data only for the algorithms implemented in MSUnCore is for no influence of any factor, other than the type of algorithm.

For each instance X, both the optimum value ($Optimum(X)$) and the sum of weights of soft clauses ($SumWeightSoft(X)$) were computed. Then a percentage was calculated as the division between both values as $Optimum(X)/SumWeightSoft(X)$. For unweighted instances, each clause was considered to have weight 1. In partial instances, hard clauses were disregarded. In the remainder of the analysis the percentage $Optimum(X)/SumWeightSoft(X)$ is named as $POvsS$.

The instances were then grouped according to the POvsS percentage and to their category. The results are presented in Table 15. The first row of Table 15 shows the intervals of percentage considered, from 0 to 100 % (intervals of size 10 %). The first column shows the category to which the instances belong to, that is, MS for MaxSAT instances, PMS for partial MaxSAT instances, WMS for weighted MaxSAT instances

Table 15 Number of instances according to the POvsS percentage

		[0; 0.1[	[0.1; 0.2[	[0.2; 0.3[	[0.3; 0.4[	[0.4; 0.5[	[0.5; 0.6[	[0.6; 0.7[	[0.7; 0.8[	[0.8; 0.9[	[0.9; 1]
MS	Crafted	2	5	1	0	0	0	0	0	0	0
	Industrial	106	0	0	0	0	0	0	0	0	0
PMS	Crafted	20	27	30	21	45	23	8	3	6	88
	Industrial	324	125	155	100	73	104	5	1	0	1
WMS	Crafted	61	2	0	0	0	0	0	0	0	0
WPMS	Crafted	125	53	33	19	24	18	34	13	10	84
	Industrial	199	0	0	0	0	0	0	0	0	0
ALL		837	212	219	140	142	145	47	17	16	173

and WPMS for weighted partial MaxSAT instances. The ALL considers all instances independent of the category. The second column shows whether the instances are crafted or industrial. The remaining cells show the number of instances that fall in a certain category (given by the row) and have a POvsS percentage in a certain interval (given by the column). As it can be seen, the majority of instances have an optimum value that is below 10 % of the sum of weights of the soft clauses. In fact it represents 43 % (837/1948) of the total of instances considered. This is specially true in industrial instances, the only category that has instances with an optimum greater than 10 % of the sum of weights of clauses is the industrial partial MaxSAT category.

Given the data in Table 15, the instances were further grouped in three intervals of POvsS percentages, namely [0; 0.1[, [0.1; 0.6[, and [0.6; 1]. A cactus plot was created for each of the intervals. Figure 8a, b, and c present the cactus plots for each of the intervals for all of the instances (independent of category they belong to). Figure 8d presents the cactus plot for the instances with a POvsS in [0; 0.1[that belong to the WPMS Industrial category.

From Fig. 8a, it can be seen that for a low POvsS (within [0; 0.1[), two of the top three algorithms to have a better performance are based on refining a lower bound, namely `WMSU1` and `WPM2`. The best performing algorithm is `BIN-C-D` that refines both a lower and an upper bound, but restricting these instances to WPMS Industrial instances (Fig. 8d) it can be seen that `WMSU1` performs better than `BIN-C-D` or `WPM2` on these instances. The main reason for this is due to the low POvsS of these instances as well as the fact that both `BIN-C-D` and `WPM2` have to encode pseudo-Boolean constraints on these instances whereas `WMSU1` always encodes cardinality constraints.

From Fig. 8c, it can be seen that for a high POvsS (within [0.6; 1]), the best performing algorithm is `WMSU4`. Despite `WMSU4` refining both a lower and an upper bound, `WMSU4` is much more oriented by satisfying assignments than by the unsatisfiable cores. On the other hand, both `WMSU1` and `WPM2` (characterized by refining lower bounds) are the worst performing algorithms for these instances, since the number of iterations required by these algorithms is high.

For the instances that have a POvsS within [0.1; 0.6[, Fig. 8b shows that the best performing algorithms (all very close) are based on binary search or bit-based search (similar to a binary search on the bit value of the optimum).

The experiments is this section indicates that depending on the location of the optimum value with regards to the sum of weights of soft clauses, then one type of algorithm may perform better than others, namely on instances with a very low POvsS is expected that algorithms that refine lower bounds, such as `WMSU1` and

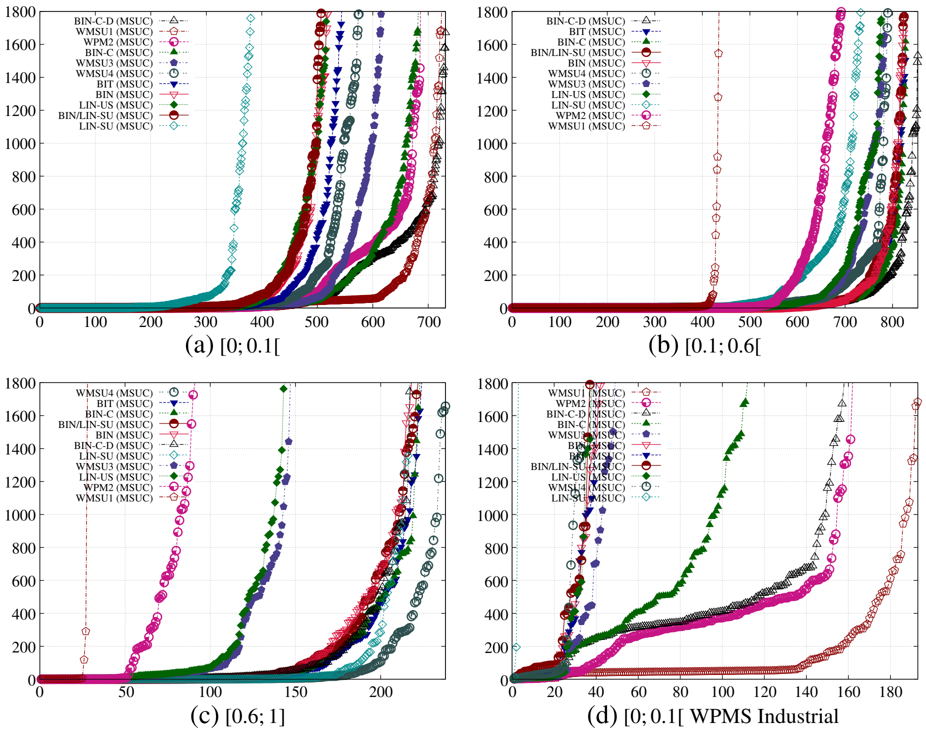

Fig. 8 Run time distributions for non-random MaxSAT instances with given POvsS percentage

WPM2 have a better performance. On the other hand if the instance is known to have a high POvsS, then algorithms like WMSU4 which are heavily oriented by satisfying assignments, are expected to perform better.

Finally, if the POvsS of the instance is somewhere in the middle or it is unknown, then an algorithm based on binary search such as BIN-C-D may be a better choice.

8 Conclusions

MaxSAT is an optimization variant of the Satisfiability problem that finds a wide range of applications. In particular, MaxSAT has recently been applied in industrial contexts including Design Debugging [33, 111], Fault Localization of ANSI-C Programs [66, 67], Post-Silicon Validation [130], Planning [38, 68, 108, 129], and Model-Based-Diagnosis [46], to name a few.

In the last decade, a large number of exact algorithms have been proposed for MaxSAT, some of which are based on iteratively calling a SAT solver and are called *iterative* MaxSAT solvers. Exploiting the ability of SAT solvers to compute reasons for unsatisfiable formulas (unsatisfiable cores), new approaches for MaxSAT solving have emerged that take advantage of the presence of cores to further enhance their performance. These algorithms are called *core-guided* MaxSAT algorithms.

This paper overviews existing *iterative* and *core-guided* MaxSAT algorithms and characterizes them in terms of number of relaxation variables, number of constraints and number of calls to a SAT oracle. Additionally, several algorithms originally available only for unweighted MaxSAT have been extended to handle weighted formulas.

Iterative algorithms initially relax all soft clauses. Then, they iteratively add a cardinality constraint on the number of relaxation variables allowed to be assigned *true* and call a SAT solver. In the case of weighted versions of the algorithms, instead of the cardinality constraint, a pseudo-Boolean constraint is added that constrains the allowed weight of the associated cost function. Core-guided MaxSAT algorithms compute unsatisfiable cores when the outcome of the SAT solver is *unsatisfiable*, and use the information of unsatisfiable cores to relax soft clauses on demand and to compute more precise and smaller constraints.

An extensive empirical study has been conducted that considers all non-random instances from recent MaxSAT Evaluations, the surveyed algorithms and other state-of-the-art approaches such as branch and bound MaxSAT solvers, pseudo-Boolean solvers and ASP solvers. The iterative and core-guided MaxSAT algorithms were developed in the same implementation framework, which allows for a fair comparison of all algorithmic schemes independently of implementation details. Both original (if any) and our own implementation of each iterative and core-guided MaxSAT algorithm were considered. Branch and bound algorithms [61, 70, 79] are known to be the best complete approach to handle random benchmarks, and the results in this paper indicate that they are also the best approach to handle non-partial crafted benchmarks. Differently, for crafted partial benchmarks iterative and core-guided MaxSAT algorithms are quite competitive. Regarding industrial (partial and non-partial) benchmarks, core-guided MaxSAT algorithms followed by iterative MaxSAT algorithms are among the best approaches, whereas branch and bound algorithms perform poorly. In fact, the results indicate that WMSU1/WPM1 is the best approach to handle industrial unweighted MaxSAT and weighted partial industrial MaxSAT, whereas core-guided binary search with disjoint cores is the best approach for partial MaxSAT industrial benchmarks and it is quite robust in general.

Additionally, a study of the optimum value of the instances with regards to the sum of weights of all the soft clauses was conducted, where the POvsS percentage was computed as the division between those two values (for each instance). The results suggest that for instances with a high value of POvsS, algorithms that are oriented by satisfying assignments may perform better, such as the WMSU4 algorithm. On the other hand, algorithms that are based on refining a lower bound have a better behavior on instances with a low POvsS; for example WMSU1 and WPM2. For values of POvsS that fall in the middle, then algorithms that are based on binary search like BIN-C-D may be a better choice.

Overall, this survey attempts to highlight some of the algorithms for MaxSAT that are based on iteratively calling a SAT solver, whether they take advantage of unsatisfiable cores or not. Such algorithms have been consecutively demonstrated to be the best performing MaxSAT algorithms for industrial benchmarks, by MaxSAT evaluations since 2008 [14]. The area itself is an active research field where new algorithms and new techniques emerge every year ([6, 10, 89, 90, 92] to name a few).

An extensive experimental evaluation is carried on the survey, and in general the results indicate that core-guided MaxSAT algorithms (which take advantage of unsatisfiable core) are better than iterative algorithms and that core-guided MaxSAT algorithms are fairly competitive compared to the remaining approaches and the current best approach for industrial benchmarks.

Acknowledgements The authors thank the reviewers for the many suggestions made. which helped improving the manuscript significantly. This work is partially supported by SFI PI grant BEACON (09/IN.1/I2618), by FCT grants ATTEST (CMU-PT/ELE/0009/2009), POLARIS (PTDC/EIA-CCO/123051/2010), and INESC-ID's multiannual PIDDAC funding PEst-OE/EEI/LA0021/2013, and by Spanish MICINN Project ARINF (TIN2009-14704-C03-01).

Appendix

`WMSU1`—worst-case number of relaxation variables

This section presents the general idea to obtain the worst case total number of relaxation variables added by `WMSU1`.

The number of relaxation variables added is related to the quality of the unsatisfiable subformulas computed by the SAT solver. In the following, it is assumed the worst case scenario where the SAT solver always returns the complete CNF formula as the unsatisfiable core. In practice SAT solvers return reasonably accurate unsatisfiable cores. A detailed characterization of the worst-case number of relaxation variables/iterations in this case is an open research topic.

Consider an input WCNF formula φ given to `WMSU1`. It is assumed the worst case scenario where φ contains a clause with weight 1. Since the unsatisfiable core is always the complete formula, then in particular contains a clause with weight 1. Thus in the following, the minimum weight of the unsatisfiable cores returned by the SAT solver is always 1.

Suppose that φ has m soft clauses with different weights that range from 1 to α, that is $\alpha = \max\{w_i | (c_i, w_i) \in \varphi, 1 \leq i \leq m\}$. Also, let W represent the sum of weights of all soft clauses ($W = \sum_{i=1}^{m} w_i$). In the worst case scenario the optimum cost of the unsatisfiable clauses is W.

As explained in Section 5, `WMSU1` works by making calls to the SAT solver until a satisfiable outcome is obtained. In each unsatisfiable iteration, each soft clause belonging to the core with a weight of 1 is augmented with a new relaxation variable. On the other hand, soft clauses belonging to the core with a weight greater than 1 are replicated, which means that the clause is duplicated and the duplicated clause is augmented with a new relaxation variable. Under the assumptions explained above, the new relaxed clause due to replication is associated with a weight of 1, and the weight of the original clause is reduced by 1.

`WMSU1` obtains a satisfiable iteration when the sum of the contributions of each core reaches the optimum solution. In this case each unsatisfiable core contributes with a cost of 1 and the optimum cost is assumed to be W, then there are total of W iterations.

Consider that variables N_i represent the number of soft clauses in φ with a weight i, where $1 \leq i \leq \alpha$. Table 16 shows the total number of relaxation variables, and total number of new clauses (due to replication) added to the formula after each iteration lower or equal to α. In the table, the brackets [and] represent that none of the clauses associated to the expression inside were replicated, that is, all the clauses associated to the expression inside have weight 1.

For the first iteration, each clause with weight 1 is augmented with a new relaxation variable. As there are N_1 such clauses, then `WMSU1` adds N_1 new relaxation variables. The clauses with a weight greater than 1 are replicated, meaning new clauses are created each containing a new relaxation variable. Since there are

Table 16 Total number of relaxation variables and number of clauses added after each iteration lower or equal to α

Iteration	Number of relaxation variables	Number of added clauses
1	$[N_1] + N_2 + N_3 + \ldots + N_\alpha$	$N_2 + \ldots + N_\alpha$
2	$[2N_1] + [2N_2 + N_2] + 2N_3 + N_3 + \ldots + 2N_\alpha + N_\alpha$	$[N_2] + [N_3] + N_3 + \ldots + [N_\alpha] + N_\alpha$
3	$[3N_1] + [3N_2 + 2N_2] + [3N_3 + 2N_3 + N3] + \ldots$ $\ldots + 3N_\alpha + 2N_\alpha + N_\alpha$	$[N_2] + [2N_3] + [2N_4] + N_4 + \ldots$ $\ldots + [2N_\alpha] + N_\alpha$
$\vdots$	$\vdots$	$\vdots$
α	$[\alpha N_1]+$	$[N_2]+$
	$[\alpha N_2 + (\alpha - 1)N_2]+$	$[2N_3]+$
	$[\alpha N_3 + (\alpha - 1)N_3 + (\alpha - 2)N3]+$	$[3N_4]+$
	$\vdots$	$\vdots$
	$[\alpha N_\alpha + (\alpha - 1)N_\alpha + \ldots + N_\alpha]$	$[(\alpha - 1)N_\alpha]$

$N_2 + \ldots + N_\alpha$ clauses with weight greater than 1, then $N_2 + \ldots + N_\alpha$ new relaxation variables are added, and $N_2 + \ldots + N_\alpha$ new clauses are created.

The second and following iterations are similar, but taking into account the clauses with weight 1 created from the replicated clauses in the previous iteration. Also, as the weight of the original clauses are decreased by 1, then after iteration i, the weight of the clauses originally with weight i is decreased to 1 and they are no longer replicated but are still augmented with new relaxation variables.

After iteration α all clauses have weight 1. Any clause belonging to an unsatisfiable core is then relaxed with a new relaxation variable (and no replication of clauses is performed). As such the total number of clauses added by WMSU1 in these conditions is given in the last line of Table 16 in the last column, and is bounded by the following expression where is considered that $N_i \leq m$:

$$\begin{aligned}\#newcls &= \sum\nolimits_{i=2}^{\alpha} \left((i-1)N_i\right) \\ &\leq \sum\nolimits_{i=2}^{\alpha} \left((i-1)m\right) \\ &= \frac{m\alpha(\alpha-1)}{2}\end{aligned}$$

For iterations $(\alpha + 1)$ to W, the total number of relaxation variables added is shown in Table 17. The main difference is the addition of relaxation variables without replication of clauses, thus the increase of the constants. The total number of relaxation variables added by WMSU1 is given in the last line of Table 17 and is bounded by the following expression where again is considered that $N_i \leq m$:

$$\begin{aligned}\#rvars &= \sum\nolimits_{i=1}^{\alpha} \left(\sum\nolimits_{j=W-i+1}^{W} jN_i\right) \\ &= \sum\nolimits_{i=1}^{\alpha} \left(N_i \sum\nolimits_{j=W-i+1}^{W} j\right) \\ &\leq m \sum\nolimits_{i=1}^{\alpha} \frac{(2W-i+1)}{2} i \\ &= \frac{m\alpha(\alpha+1)(3W-\alpha+1)}{6}\end{aligned}$$

Table 17 Number of relaxation variables added after each iteration greater than α

Iteration	Number of relaxation variables
$\alpha + 1$	$[(\alpha+1)N_1]+$
	$[(\alpha+1)N_2+\alpha N_2]+$
	$[(\alpha+1)N_3+\alpha N_3+(\alpha-1)N3]+$
	$\vdots$
	$[(\alpha+1)N_\alpha+\alpha N_\alpha+\ldots+2N_\alpha]$
$\vdots$	$\vdots$
$\alpha + k$	$[(\alpha+k)N_1]+$
	$[(\alpha+k)N_2+(\alpha+k-1)N_2]+$
	$[(\alpha+k)N_3+(\alpha+k-1)N_3+(\alpha+k-2)N3]+$
	$\vdots$
	$[(\alpha+k)N_\alpha+(\alpha+k-1)N_\alpha+\ldots+(k+1)N_\alpha]$
$\vdots$	$\vdots$
W	$[WN_1]+$
	$[WN_2+(W-1)N_2]+$
	$[WN_3+(W-1)N_3+(W-2)N3]+$
	$\vdots$
	$[WN_\alpha+(W-1)N_\alpha+\ldots+(W-\alpha+1)N_\alpha]$

References

1. Aksoy, L., daCosta, E.A.C., Flores, P.F., Monteiro, J. (2008). Exact and approximate algorithms for the optimization of area and delay in multiple constant multiplications. *IEEE Transactions on CAD of Integrated Circuits and Systems on CAD, 27*(6), 1013–1026.
2. Aloul, F., Ramani, A., Markov, I., Sakallah, K. (2002). PBS: A backtrack search pseudo-Boolean solver. In *Symposium on theory and applications of satisfiability testing* (pp. 346–353).
3. Aloul, F.A., Ramani, A., Markov, I.L., Sakallah, K.A. (2002). Generic ILP versus specialized 0-1 ILP: An update. In *International conference on computer-aided design* (pp. 450–457).
4. Andres, B., Kaufmann, B., Matheis, O., Schaub, T. (2012). Unsatisfiability-based optimization in clasp. In *International conference on logic programming (Technical communications)* (pp. 211–221).
5. Anjos, M.F. (2006). Semidefinite optimization approaches for satisfiability and maximum-satisfiability problems. *Journal on Satisfiability, Boolean Modeling and Computation, 1*(1), 1–47.
6. Ansótegui, C., Bonet, M.L., Gabàs, J., Levy, J. (2012). Improving SAT-based weighted MaxSAT solvers. In *International conference on principles and practice of constraint programming* (pp. 86–101).
7. Ansótegui, C., Bonet, M.L., Levy, J. (2009). On solving MaxSAT through SAT. In *International conference of the Catalan Association for artificial intelligence* (pp. 284–292).
8. Ansótegui, C., Bonet, M.L., Levy, J. (2009). Solving (weighted) partial MaxSAT through satisfiability testing. In *International conference on theory and applications of satisfiability testing* (pp. 427–440).
9. Ansótegui, C., Bonet, M.L., Levy, J. (2010). A new algorithm for weighted partial MaxSAT. In *AAAI conference on artificial intelligence* (pp. 3–8).
10. Ansótegui, C., Bonet, M.L., Levy, J. (2013). SAT-based MaxSAT algorithms. In *Artificial inteligence journal* (Vol. 196, pp. 77–105).
11. Ansótegui, C., & Manyà, F. (2004). Mapping problems with finite-domain variables into problems with Boolean variables. In *International conference on theory and applications of satisfiability testing* (pp. 111–119).

12. Ardagna, C.A., diVimercati, S.D.C., Foresti, S., Paraboschi, S., Samarati, P. (2010). Minimizing disclosure of private information in credential-based interactions: A graph-based approach. In *International conference on social computing/international conference on privacy, security, risk and trust* (pp. 743–750).
13. Argelich, J., Berre, D.L., Lynce, I., Marques-Silva, J., Rapicault, P. (2010). Solving linux upgradeability problems using Boolean optimization. In *International workshop on logics for component configuration* (pp. 11–22).
14. Argelich, J., Li, C.M., Manya, F., Planes, J. (2011). Experimenting with the instances of the MaxSAT Evaluation. In *International conference of the catalan association for artificial intelligence* (pp. 360–361).
15. Argelich, J., & Lynce, I. (2008). CNF instances from the software package installation problem. In *RCRA international workshop on "Experimental Evaluation of Algorithms for solving problems with combinatorial explosion"*.
16. Argelich, J., Lynce, I., Marques-Silva, J. (2009). On solving Boolean multilevel optimization problems. In *International joint conference on artificial intelligence* (pp. 393–398).
17. Asín, R., & Nieuwenhuis, R. (2010). Curriculum-based course timetabling with SAT and MaxSAT. In *International conference on the practice and theory of automated timetabling* (pp. 42–56).
18. Asín, R., & Nieuwenhuis, R. (2012). Curriculum-based course timetabling with SAT and MaxSAT. *Annals of Operations Research*, 1–21.
19. Asín, R., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E. (2011). Cardinality networks: a theoretical and empirical study. *Constraints, 16*(2), 195–221.
20. Bailleux, O., & Boufkhad, Y. (2003). Efficient CNF encoding of Boolean cardinality constraints. In *International conference on principles and practice of constraint programming* (pp. 108–122).
21. Bansal, N., & Raman, V. (1999). Upper bounds for MaxSat: Further improved. In *International symposium on algorithms and computation* (pp. 247–258).
22. Barrett, C., Sebastiani, R., Seshia, S.A., Tinelli, C. (2009). Satisfiability modulo theories. In *Handbook of satisfiability* (pp. 825–884). IOS Press.
23. Barth, P. (1995). *A Davis-Putnam enumeration algorithm for linear pseudo-Boolean optimization*. Technical Report MPI-I-95-2-003, Max Plank Institute for Computer Science.
24. Batcher, K.E. (1968). Sorting networks and their applications. In *AFIPS spring joint computing conference* (pp. 307–314).
25. Berre, D.L., & Parrain, A. (2010). The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation, 7*, 59–64.
26. Biere, A. (2008). PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation, 2*, 75–97.
27. Birnbaum, E., & Lozinskii, E.L. (2003). Consistent subsets of inconsistent systems: structure and behaviour. *Journal of Experimental and Theoretical Artificial Intelligence, 15*(1), 25–46.
28. Bonet, M.L., Levy, J., Manyà, F. (2007). Resolution for Max-SAT. *Artificial Intelligence Journal, 171*(8–9), 606–618.
29. Borchers, B., & Furman, J. (1999). A two-phase exact algorithm for MAX-SAT and weighted MAX-SAT problems. *Journal of Combinatorial Optimization, 2*, 299–306.
30. Brihaye, T., Bruyère, V., Doyen, L., Ducobu, M., Raskin, J.-F. (2011). Antichain-based QBF solving. In *International symposium on automated technology for verification and analysis* (pp. 183–197).
31. Cha, B., Iwama, K., Kambayashi, Y., Miyazaki, S. (1997). Local search algorithms for partial MaxSAT. In *AAAI conference on artificial intelligence/IAAI innovative applications of artificial intelligence conference* (pp. 263–268).
32. Chai, D., & Kuehlmann, A. (2003). A fast pseudo-Boolean constraint solver. In *Design automation conference* (pp. 830–835).
33. Chen, Y., Safarpour, S., Marques-Silva, J., Veneris, A.G. (2010). Automated design debugging with maximum satisfiability. *IEEE Transactions on CAD of Integrated Circuits and Systems, 29*(11), 1804–1817.
34. Chen, Y., Safarpour, S., Veneris, A., Marques-Silva, J. (2009). Spatial and temporal design debug using partial MaxSAT. In *IEEE great lakes symposium on VLSI*.
35. Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R., Stenico, C. (2010). Satisfiability modulo the theory of costs: Foundations and applications. In *International conference tools and algorithms for the construction and analysis of systems* (pp. 99–113).
36. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R. (2013). A modular approach to MaxSAT modulo theories. In *International conference on theory and applications of satisfiability testing* (pp. 150–165).

37. Codish, M., Lagoon, V., Stuckey, P.J. (2008). Logic programming with satisfiability. *Journal of Theory and Practice of Logic Programming, 8*(1), 121–128.
38. Cooper, M.C., Cussat-Blanc, S., deRoquemaurel, M., Régnier, P. (2006). Soft arc consistency applied to optimal planning. In *International conference on principles and practice of constraint programming* (pp. 680–684).
39. Cooper, M.C., deGivry, S., Sanchez, M., Schiex, T., Zytnicki, M., Werner, T. (2010). Soft arc consistency revisited. *Artificial Intelligence Journal, 174*(7–8), 449–478.
40. Davies, J., & Bacchus, F. (2011). Solving MaxSAT by solving a sequence of simpler SAT instances. In *International conference on principles and practice of constraint programming* (pp. 225–239).
41. Davies, J., Cho, J., Bacchus, F. (2010). Using learnt clauses in MaxSAT. In *International conference on principles and practice of constraint programming* (pp. 176–190).
42. deGivry, S., Larrosa, J., Meseguer, P., Schiex, T. (2003). Solving Max-SAT as weighted CSP. In *International conference on principles and practice of constraint programming* (pp. 363–376).
43. deMoura, L.M., & Bjørner, N. (2008). Z3: An efficient SMT solver. In *International conference tools and algorithms for the construction and analysis of systems* (pp. 337–340).
44. Eén, N., & Sörensson, N. (2003). An extensible SAT-solver. In *International conference on theory and applications of satisfiability testing* (pp. 502–518).
45. Een, N., & Sörensson, N. (2006). Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation, 2*, 1–26.
46. Feldman, A., Provan, G., deKleer, J., Robert, S., van Gemund, A. (2010). Solving model-based diagnosis problems with MaxSAT solvers and vice versa. In *International workshop on the principles of diagnosis*.
47. Fu, Z., & Malik, S. (2006). On solving the partial MAX-SAT problem. In *International conference on theory and applications of satisfiability testing* (pp. 252–265).
48. Gebser, M., Kaufmann, B., Schaub, T. (2009). The conflict-driven answer set solver clasp: Progress report. In *International conference on logic programming and nonmonotonic reasoning* (pp. 509–514).
49. Gent, I.P., & Nightingale, P. (2004). A new encoding of alldifferent into SAT. In *International workshop on modelling and reformulating constraint satisfaction problems* (pp. 95–110).
50. Giunchiglia, E., Lierler, Y., Maratea, M. (2006). Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning, 36*(4), 345–377.
51. Giunchiglia, E., & Maratea, M. (2006). Optsat: A tool for solving SAT related optimization problems. In *European conference on logics in artificial intelligence (JELIA)* (pp. 485–489).
52. Giunchiglia, E., & Maratea, M. (2006). Solving optimization problems with DLL. In *European conference on artificial intelligence* (pp. 377–381).
53. Giunchiglia, E., & Maratea, M. (2007). Planning as satisfiability with preferences. In *AAAI conference on artificial intelligence* (pp. 987–992).
54. Gomes, C.P., van Hoeve, W.J., Leahu, L. (2006). The power of semidefinite programming relaxations for Max-SAT. In *International conference integration of AI and OR techniques in constraint programming for combinatorial optimization problems* (pp. 104–118).
55. Gottlob, G. (1995). NP trees and Carnap's modal logic. *Journal of ACM, 42*(2), 421–457.
56. Graca, A., Lynce, I., Marques-Silva, J., Oliveira, A. (2010). Efficient and accurate haplotype inference by combining parsimony and pedigree information. In *International conference algebraic and numeric biology* (pp. 38–56).
57. Graca, A., Marques-Silva, J., Lynce, I., Oliveira, A. (2011). Haplotype inference with pseudo-Boolean optimization. *Annals of Operations Research, 184*(1), 137–162.
58. Guerra, J., & Lynce, I. (2012). Reasoning over biological networks using maximum satisfiability. In *International conference on principles and practice of constraint programming* (pp. 941–956).
59. Hachtel, G.D., & Somenzi, F. (1996). *Logic synthesis and verification algorithms*. Kluwer.
60. Heras, F., Larrosa, J., deGivry, S., Schiex, T. (2008). 2006 and 2007 Max-SAT evaluations: contributed instances. *Journal on Satisfiability, Boolean Modeling and Computation, 4*(2–4), 239–250.
61. Heras, F., Larrosa, J., Oliveras, A. (2008). MiniMaxSat: an efficient weighted Max-SAT solver. *Journal of Artificial Intelligence Research, 31*, 1–32.
62. Heras, F., & Marques-Silva, J. (2011). Read-once resolution for unsatisfiability-based Max-SAT algorithms. In *International joint conference on artificial intelligence* (pp. 572–577).
63. Heras, F., Morgado, A., Marques-Silva, J. (2011). Core-guided binary search for maximum satisfiability. In *AAAI conference on artificial intelligence*.

64. Hoos, H., & Stützle, T. (2005). *Stochastic local search: Foundations and applications*. Morgan Kaufmann.
65. Hoos, H.H. (2002). An adaptive noise mechanism for WalkSAT. In *AAAI conference on artificial intelligence/IAAI innovative applications of artificial intelligence conference* (pp. 655–660).
66. Jose, M., & Majumdar, R. (2011). Bug-assist: Assisting fault localization in ANSI-C programs. In *International conference on computer aided verification* (pp. 504–509).
67. Jose, M., & Majumdar, R. (2011). Cause clue clauses: Error localization using maximum satisfiability. In *ACM SIGPLAN conference on programming language design and implementation* (pp. 437–446).
68. Juma, F., Hsu, E.I., McIlraith, S.A. (2012). Preference-based planning via MaxSAT. In *Canadian conference on AI* (pp. 109–120).
69. Koshimura, M., Zhang, T., Fujita, H., Hasegawa, R. (2012). QMaxSAT: a partial Max-SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation, 8*, 95–100.
70. Kuegel, A. (2010). Improved exact solver for the weighted MAX-SAT problem. In *Pragmatics of SAT*.
71. Larrosa, J., & Heras, F. (2005). Resolution in Max-SAT and its relation to local consistency in weighted CSPs. In *International joint conference on artificial intelligence* (pp. 193–198).
72. Larrosa, J., Heras, F., deGivry, S. (2008). A logical approach to efficient Max-SAT solving. *Artificial Inteligence Journal, 172*(2–3), 204–233.
73. Larrosa, J., & Schiex, T. (2004). Solving weighted CSP by maintaining arc consistency. *Artificial Inteligence Journal, 159*(1–2), 1–26.
74. Li, C.M., & Manyà, F. (2009). MaxSAT, hard and soft constraints. In *Handbook of satisfiability* (pp. 613–632). IOS Press.
75. Li, C.M., Manyà, F., Planes, J. (2005). Exploiting unit propagation to compute lower bounds in branch and bound Max-SAT solvers. In *International conference on principles and practice of constraint programming* (pp. 403–414).
76. Li, C.M., Manyà, F., Planes, J. (2007). New inference rules for Max-SAT. *Journal of Artificial Intelligence Research, 30*, 321–359.
77. Liffiton, M.H., Sakallah, K.A. (2005). On finding all minimally unsatisfiable subformulas. In *International conference on theory and applications of satisfiability testing* (pp. 173–186).
78. Liffiton, M.H., & Sakallah, K.A. (2008). Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal Automated Reasoning, 40*(1), 1–33.
79. Lin, H., Su, K., Li, C.M. (2008). Within-problem learning for efficient lower bound computation in Max-SAT solving. In *AAAI conference on artificial intelligence* (pp. 351–356).
80. Mancinelli, F., Boender, J., diCosmo, R., Vouillon, J., Durak, B., Leroy, X., Treinen, R. (2006). Managing the complexity of large free and open source package-based software distributions. In *International conference on automated software engineering* (pp. 199–208).
81. Mangassarian, H., Veneris, A.G., Safarpour, S., Najm, F.N., Abadir, M.S. (2007). Maximum circuit activity estimation using pseudo-Boolean satisfiability. In *Conference on design, automation and test in Europe* (pp. 1538–1543).
82. Manquinho, V., Marques-Silva, J., Planes, J. (2009). Algorithms for weighted Boolean optimization. In *International conference on theory and applications of satisfiability testing* (pp. 495–508).
83. Manquinho, V., Martins, R., Lynce, I. (2010). Improving unsatisfiability-based algorithms for Boolean optimization. In *International conference on theory and applications of satisfiability testing* (pp. 181–193).
84. Marques-Silva, J., Argelich, J., Graça, A., Lynce, I. (2011). Boolean lexicographic optimization: Algorithms & applications. *Annals of Mathematics and Artificial Intelligence, 62*(3–4), 317–343.
85. Marques-Silva, J., & Manquinho, V. (2008). Towards more effective unsatisfiability-based maximum satisfiability algorithms. In *International conference on theory and applications of satisfiability testing* (pp. 225–230).
86. Marques-Silva, J., & Planes, J. (2007). On using unsatisfiability for solving maximum satisfiability. *Computing Research Repository*. arXiv: abs/0712.0097.
87. Marques-Silva, J., Planes, J. (2008). Algorithms for maximum satisfiability using unsatisfiable cores. In *Conference on design, automation and testing in Europe* (pp. 408–413).
88. Marques-Silva, J., & Sakallah, K.A. (1996). GRASP—a new search algorithm for satisfiability. In *International conference on computer-aided design* (pp. 220–227).

89. Martins, R., Manquinho, V., Lynce, I. (2012). On partitioning for maximum satisfiability. In *European conference on artificial intelligence* (pp. 913–914).
90. Martins, R., Manquinho, V.M., Lynce, I. (2013). Community-based partitioning for MaxSAT solving. In *International conference on theory and applications of satisfiability testing* (pp. 182–191).
91. Morgado, A., Heras, F., Marques-Silva, J. (2011). The MSUnCore MaxSAT solver. In *Pragmatics of SAT*.
92. Morgado, A., Heras, F., Marques-Silva, J. (2012). Improvements to core-guided binary search for MaxSAT. In *Theory and applications of satisfiability testing* (pp. 284–297).
93. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S. (2001). Chaff: Engineering an efficient SAT solver. In *Design automation conference* (pp. 530–535).
94. Neveu, B., Trombettoni, G., Glover, F. (2004). ID Walk: A candidate list strategy with a simple diversification device. In *International conference on principles and practice of constraint programming* (pp. 423–437).
95. Niedermeier, R., Rossmanith, P. (2000). New upper bounds for maximum satisfiability. *Journal of Algorithms, 36*(1), 63–88.
96. Nieuwenhuis, R., & Oliveras, A. (2006). On SAT modulo theories and optimization problems. In *International conference on theory and applications of satisfiability testing* (pp. 156–169).
97. Nieuwenhuis, R., Oliveras, A., Tinelli, C. (2006). Solving SAT and SAT modulo theories: from an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of ACM, 53*(6), 937–977.
98. Oikarinen, E., Järvisalo, M. (2009). Max-ASP: Maximum satisfiability of answer set programs. In *International conference on logic programming and nonmonotonic reasoning* (pp. 236–249).
99. Palubeckis, G. (2009). A new bounding procedure and an improved exact algorithm for the MAX-2-SAT problem. *Applied Mathematics and Computation, 215*(3), 1106–1117.
100. Papadimitriou, C. (1994). *Computational complexity*. USA: Addison-Wesley.
101. Papadimitriou, C., & Zachos, S. (1983). Two remarks on the power of counting. *Theoretical Computer Science*, 269–276.
102. Park, J.D. (2002). Using weighted MAX-SAT engines to solve MPE. In *AAAI conference on artificial intelligence* (pp. 682–687).
103. Pipatsrisawat, K., Palyan, A., Chavira, M., Choi, A., Darwiche, A. (2008). Solving weighted Max-SAT problems in a reduced search space: a performance analysis. *Journal on Satisfiability Boolean Modeling and Computation, 4*, 191–217.
104. Prestwich, S. (2009). CNF encodings. In *Handbook of satisfiability* (pp. 75–98). IOS Press.
105. Prestwich, S.D. (2007). Variable dependency in local search: Prevention is better than cure. In *International conference on theory and applications of satisfiability testing* (pp. 107–120).
106. Ramírez, M., & Geffner, H. (2007). Structural relaxations by variable renaming and their compilation for solving MinCostSAT. In *International conference on principles and practice of constraint programming* (pp. 605–619).
107. Reiter, R. (1987). A theory of diagnosis from first principles. *Artificial Inteligence Journal, 32*(1), 57–95.
108. Robinson, N., Gretton, C., Pham, D.N., Sattar, A. (2010). Partial weighted MaxSAT for optimal planning. In *Pacific rim international conference on artificial intelligence* (pp. 231–243).
109. Rosa, E.D., Giunchiglia, E., Maratea, M. (2010). Solving satisfiability problems with preferences. *Constraints, 15*(4), 485–515.
110. Roussel, O., & Manquinho, V. (2009). Pseudo-Boolean and cardinality constraints. In *Handbook of satisfiability* (pp. 695–734). IOS Press.
111. Safarpour, S., Mangassarian, H., Veneris, A., Liffiton, M.H., Sakallah, K.A. (2007). Improved design debugging using maximum satisfiability. In *Formal methods in computer-aided design*.
112. Sandholm, T. (1999). An algorithm for optimal winner determination in combinatorial auctions. In *International joint conference on artificial intelligence* (pp. 542–547).
113. Sebastiani, R. (2007). Lazy satisfiability modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation, 3*, 141–224.
114. Sebastiani, R., & Tomasi, S. (2012). Optimization in SMT with LA(Q) cost functions. In *International joint conference in automated reasoning* (pp. 484–498).
115. Selman, B., Kautz, H.A., Cohen, B. (1994). Noise strategies for improving local search. In *AAAI conference on artificial intelligence* (pp. 337–343).
116. Selman, B., Levesque, H.J., Mitchell, D.G. (1992). A new method for solving hard satisfiability problems. In *AAAI conference on artificial intelligence* (pp. 440–446).

117. Sheini, H., & Sakallah, K. (2006). Pueblo: a hybrid pseudo-Boolean SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation, 2*(3–4), 165–189.
118. Shen, H., & Zhang, H. (2005). Improving exact algorithms for MAX-2-SAT. *Annals of Mathematics and Artificial Intelligence, 44*(4), 419–436.
119. Sinz, C. (2005). Towards an optimal CNF encoding of Boolean cardinality constraints. In *International conference on principles and practice of constraint programming* (pp. 827–831).
120. Strickland, D., Barnes, E., Sokol, J. (2005). Optimal protein structure alignment using maximum cliques. *Operations Research, 53*(3), 389–402.
121. Teresa Alsinet, J.P., Manyà, F. (2004). A Max-SAT solver with lazy data structures. In *Ibero-American conference on AI (IBERAMIA)* (pp. 334–342).
122. Tompkins, D.A.D., & Hoos, H.H. (2004). UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT & Max-SAT. In *International conference on theory and applications of satisfiability testing* (pp. 37–46).
123. Tucker, C., Shuffelton, D., Jhala, R., Lerner, S. (2007). OPIUM: Optimal package install/uninstall manager. In *International conference on software engineering* (pp. 178–188).
124. Vasquez, M., & Hao, J. (2001). A logic-constrained knapsack formulation and a tabu algorithm for the daily photograph scheduling of an earth observation satellite. *Journal of Computational Optimization and Applications, 20*(2), 137–157.
125. Warners, J.P. (1998). A linear-time transformation of linear inequalities into conjunctive normal form. *Information Processing Letters, 68*(2), 63–69.
126. Xing, Z., & Zhang, W. (2005). MaxSolver: an efficient exact algorithm for (weighted) maximum satisfiability. *Artificial Inteligence Journal, 164*(1–2), 47–80.
127. Xu, H., Rutenbar, R., Sakallah, K. (2002). sub-SAT: A formulation for relaxed boolean satisfiability with applications in routing. In *International symposium on physical design* (pp. 182–187).
128. Zhang, L., & Malik, S. (2003). Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Conference on design, automation and testing in Europe* (pp. 10880–10885).
129. Zhang, L., & Bacchus, F. (2012). MaxSAT heuristics for cost optimal planning. In *AAAI conference on artificial intelligence* (pp. 1846–1852).
130. Zhu, C.S., Weissenbacher, G., Malik, S. (2011). Post-silicon fault localisation using maximum satisfiability and backbones. In *Formal methods in computer-aided design* (pp. 63–66).