ECAI 2014
T. Schaub et al. (Eds.)

doi:10.3233/978-1-61499-419-0-453

Progression in Maximum Satisfiability

A. Ignatiev [1, 2] and **A. Morgado** [1] and **V. Manquinho** [1] and **I. Lynce** [1] and **J. Marques-Silva** [1, 3]

Abstract. Maximum Satisfiability (MaxSAT) is a well-known optimization version of Propositional Satisfiability (SAT), that finds a wide range of relevant practical applications. Despite the significant progress made in MaxSAT solving in recent years, many practically relevant problem instances require prohibitively large run times, and many cannot simply be solved with existing algorithms. One approach for solving MaxSAT is based on iterative SAT solving, which may optionally be guided by unsatisfiable cores. A difficulty with this class of algorithms is the possibly large number of times a SAT solver is called, e.g. for instances with very large clause weights. This paper proposes the use of geometric progressions to tackle this issue, thus allowing, for the vast majority of problem instances, to reduce the number of calls to the SAT solver. The new approach is also shown to be applicable to core-guided MaxSAT algorithms. Experimental results, obtained on a large number of problem instances, show gains when compared to state-of-the-art implementations of MaxSAT algorithms.

1 INTRODUCTION

The problem of (plain) Maximum Satisfiability (MaxSAT) consists in identifying the largest set of simultaneously satisfied clauses. Extensions of MaxSAT consider different types of clauses. Clauses can be hard, in which case they must be satisfied. Clauses can have weights, in which case the goal is to maximize the sum of weights of satisfied clauses. MaxSAT finds an ever growing range of practical aplications that include planning, fault localization in C code and design debugging [27, 23, 11, 25], among others.

The development of MaxSAT algorithms is an active area of research. On the one hand, MaxSAT algorithms have been developed for improving worst-case upper bounds [5, 7]. On the other hand, many algorithms with practical significance have been developed for MaxSAT. These include branch-and-bound search [13] and iterative SAT solving with or without unsatisfiable core-guidance [18, 2], among others. The results of the annual MaxSAT evaluation [3] confirm that different algorithmic approaches are effective at solving different classes of problem instances. Nevertheless, a general trend is that, for problem instances originating from practical domains, approaches based on iterative SAT solving, that may or may not be core-guided, are in general the most effective.

A well-known difficulty with MaxSAT approaches based on iterative SAT solving is that the number of iterations grows with the logarithm of the sum of weights of the clauses. For some applications, clause weights can be large, and this impacts the worst-case number of SAT solver calls. More importantly, the analysis of the results from the last MaxSAT evaluations reveals that, for most instances, the MaxSAT solution (represented as the minimum cost of falsified clauses) is usually much smaller than the trivial upper bound given by the sum of weights of the soft clauses. As a result, for MaxSAT solving approaches where the number of iterations depends on computed upper bounds, the number of iterations may well be significantly larger than necessary, provided the actual optimum cost was to be known.

This paper develops a new approach for solving MaxSAT that provides guarantees on the cost of the upper bound. The proposed approach uses a geometric progression for refining the lower bound by iterative SAT solving. The geometric progression is guaranteed to eventually reach a value above the optimum cost, which represents an upper bound on the optimum value. However, this value cannot be much larger than the optimum value. As a result, one can guarantee that the number of iterations grows with the logarithm of the optimum cost and not with the logarithm of the sum of the weights of the soft clauses.

The paper is organized as follows. Section 2 introduces the notation and definitions used throughout the paper. Section 3 develops a basic progression-based MaxSAT algorithm, and analyzes its worst-case number of calls to a SAT solver. Section 4 uses the basic algorithm to develop core-guided algorithms using geometric progressions. Related work is briefly overviewed in Section 5. Section 6 analyzes experimental results comparing the new algorithms with state-of-the-art MaxSAT solvers. Finally, Section 7 concludes the paper.

2 PRELIMINARIES

This section briefly introduces the definitions used throughout. Standard definitions are assumed (e.g. [6]). Let $X = \{x_1, x_2, \ldots, x_n\}$ be a set of Boolean variables. A *literal* l is either a variable x_i or its negation $\bar{x}_i$. A *clause* c is a disjunction of literals. A clause may also be regarded as a set of literals. An *assignment* $\mathcal{A}$ is a mapping $\mathcal{A} : X \to \{0, 1\}$ which satisfies (unsatisfies) a Boolean variable x if $\mathcal{A}(x) = 1$ ($\mathcal{A}(x) = 0$). Assignments can be extended in a natural way for literals (l) and clauses (c)

$$\mathcal{A}(l) = \begin{cases} \mathcal{A}(x), & \text{if } l = x \\ 1 - \mathcal{A}(x), & \text{if } l = \bar{x} \end{cases} \qquad \mathcal{A}(c) = \max\{\mathcal{A}(l) \mid l \in c\}$$

Assignments can also be regarded as sets of literals, in which case the assignment $\mathcal{A}$ satisfies (unsatisfies) a variable x if $x \in \mathcal{A}$ ($\bar{x} \in \mathcal{A}$). A *complete assignment* contains a literal for each variable, otherwise it is a *partial assignment*.

A formula $\mathcal{F}$ in *conjunctive normal form* (CNF) is a conjunction of clauses. A formula may also be regarded as a set of clauses. A *model* is a complete assignment $\mathcal{A}$ that satisfies all the clauses in a CNF formula $\mathcal{F}$. *Propositional Satisfiability* (*SAT*) is the problem of deciding whether there exists a model for a given formula.

Given an unsatisfiable formula $\mathcal{F}$, a subset of clauses $\mathcal{U}$ (i.e. $\mathcal{U} \subseteq \mathcal{F}$) whose conjunction is still unsatisfiable is called an *unsatisfiable*

[1] Instituto Superior Técnico/INESC-ID, Universidade de Lisboa, Portugal
[2] ISDCT SB RAS, Irkutsk, Russia
[3] CSI/CASL, University College Dublin, Ireland

core of the original formula. Modern SAT solvers can be instructed to generate an unsatisfiable core for unsatisfiable formulas [28].

The standard definitions of MaxSAT are assumed (e.g. [13]). Moreover, the following definitions also apply. A *weighted* clause is a pair (c, w), where c is a clause and w is the cost of its falsification, 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 $\top$. Non-mandatory clauses are also called *soft* clauses. A *weighted* CNF formula (WCNF) $\mathcal{F}$ is a set of weighted clauses. For MaxSAT, 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, *Weighted Partial MaxSAT* is the problem of finding a model of minimum cost (denoted by C). A trivial upper bound (UB) is given by $1 + W$, where $W \triangleq \sum_i w_i$.

The pseudo-codes shown in the paper assume the following notation. An initial WCNF formula $\mathcal{F}$ is given as input. The set of hard clauses of $\mathcal{F}$ is the set $\mathcal{F}_H$, and $\mathcal{F}_H$ is assumed in the algorithms to be satisfiable (which can be tested with an initial call to the SAT solver on the set $\mathcal{F}_H$). The set of soft clauses of $\mathcal{F}$ is the set $\mathcal{F}_S$, and contains m soft clauses ($m = |\mathcal{F}_S|$).

The algorithms use *relaxation variables*, which are fresh Boolean variables. The proposed algorithms add at most one relaxation variable per clause, and it is assumed that each relaxation variable r_i is associated to one (and only one) soft clause c_i having weight w_i, $1 \leq i \leq m$. The process of extending a clause with a relaxation variable is called *relaxing* the clause, and in the pseudo-codes it is performed by function $Relax()$. Function $Relax(R_0, \mathcal{F}_0, \mathcal{F}')$ receives a set of relaxation variables R_0, a set of clauses $\mathcal{F}_0$, a set of clauses that need to be relaxed $\mathcal{F}'$ ($\mathcal{F}' \subseteq \mathcal{F}_0$), and returns a pair $(R_1, \mathcal{F}_1)$. $\mathcal{F}_1$ corresponds to the clauses of $\mathcal{F}_0$, but with the clauses in $\mathcal{F}'$ relaxed. R_1 is the set R_0 augmented with the relaxation variables created when relaxing clauses in $\mathcal{F}'$.

The algorithms use cardinality or pseudo-Boolean constraints [21, 24]. These correspond to $\sum_i r_i \leq k$ or $\sum_i w_i r_i \leq k$ (respectively), and are encoded into clauses through the function $\texttt{CNF}()$. Function $\texttt{CNF}(c)$ returns the set of clauses that encodes c into CNF.

The calls to the SAT solver[4] are performed through the function $\texttt{SAT}(\mathcal{F})$, that given a CNF formula $\mathcal{F}$, returns the tuple $(st, \mathcal{U}, \mathcal{A})$. If the formula $\mathcal{F}$ is satisfiable, then st is true and $\mathcal{A}$ is a model of $\mathcal{F}$. Otherwise, st is false and $\mathcal{U}$ is an unsatisfiable core of $\mathcal{F}$.

3 MAXSAT WITH PROGRESSIONS

Existing approaches for solving MaxSAT by iterative SAT solving can either refine lower bounds, upper bounds, or perform some form of binary search [18]. These approaches can either relax all soft clauses at the outset or relax the clauses on demand, given computed unsatisfiable cores. Of existing approaches, algorithms that implement some form of binary search guarantee that in the worst-case (i.e. for weighted formulas) a polynomial number of calls is made to the SAT solver. As motivated earlier, the main drawback of approaches exploiting binary search is that the inital upper bound can be significantly larger than the (usually small) optimum value, and this of course impacts the number of SAT calls.

This section develops an alternative approach that uses a geometric progression to refine the lower bound, and such that, when the search is above the optimum value, the algorithm switches to standard binary search. The approach, although simple to build on top of

[4] It is assumed a complete SAT solver able to prove unsatisfiability and provide unsatisfiable cores.

Input: $\mathcal{F} = \mathcal{F}_S \cup \mathcal{F}_H$
1 $(R, \mathcal{F}_W) \leftarrow \texttt{Relax}(\emptyset, \mathcal{F}, \mathcal{F}_S)$
2 $(\lambda, j) \leftarrow (0, 0)$
3 **while** true **do**
4 $\quad \tau \leftarrow 2^j - 1$
5 $\quad$ **if** $\tau > \sum_{r_i \in R} w_i$ **then return** $\texttt{Bin}(\mathcal{F}_W, R, \lambda, \emptyset)$
6 $\quad (\text{st}, \mathcal{A}) \leftarrow \texttt{SAT}\,(\mathcal{F}_W \cup \texttt{CNF}\,(\sum_{r_i \in R} w_i r_i \leq \tau)\,)$
7 $\quad$ **if** st $=$ true **then return** $\texttt{Bin}(\mathcal{F}_W, R, \lambda, \mathcal{A})$
8 $\quad$ **else**
9 $\quad\quad j \leftarrow j + 1$
10 $\quad\quad \lambda \leftarrow \tau$

Algorithm 1: Progression-based MaxSAT Algorithm

standard binary search, provides guarantees in terms of the number of calls to a SAT solver. Binary search for MaxSAT requires a number of calls in $\mathcal{O}(\log W)$, where W is the sum of weights of all soft clauses. In contrast, as shown in this section, the use of approaches based on a geometric progression guarantees that the worst number of calls is in $\mathcal{O}(\log C)$, where C is the optimum cost. Algorithm 1 summarizes the progression algorithm for MaxSAT, based on iterative SAT solving (core-guided versions are detailed in Section 4). While the outcome of the SAT solver call is false, the progression is (exponentially) incremented. This is reflected in a tentative lower bound value τ, which is used in the right-hand side of the constraint bounding the cost of the falsified clauses. If the lower bound τ exceeds the sum of the weights of the soft clauses (line 5), then plain binary search (Bin) is executed between the known lower bound (λ) and the default upper bound ($1 + W$). Once the outcome of the SAT solver call is true, the solver switches to a standard model of solving MaxSAT with binary search (Bin) between the known lower bound (λ) and the upper bound given by the computed truth assignment ($\mathcal{A}$).

Proposition 1. *The worst case number of SAT solver calls for Algorithm 1 is in $\mathcal{O}(\log C)$, where C is the cost of the MaxSAT solution.*

Proof. Clearly, as soon as τ is larger than or equal to C, the outcome of the SAT solver will be true. Hence, the number of calls made to the SAT solver until it returns true is $\mathcal{O}(\log C)$. Binary search for MaxSAT requires $\mathcal{O}(\log \text{UB})$ calls to a SAT solver in the worst-case, where UB denotes some upper bound on the MaxSAT solution. Since τ cannot exceed $2\,C$, then binary search also requires $\mathcal{O}(\log C)$ calls to a SAT solver in the worst case. Thus, the worst-case number of calls to a SAT solver for Algorithm 1 is in $\mathcal{O}(\log C)$. □

As indicated earlier, for most known instances, the optimum cost is much smaller than the sum of the weights of the soft clauses. As a result, the use of geometric progressions in MaxSAT is expected to enable reductions in the number of SAT solver calls. In practice, the use of geometric progressions has a few other advantages. Since the right-hand side of the AtMost of Pseudo-Boolean (PB) constraints associated with each SAT solver call are in general smaller, then the resulting CNF encoding will be smaller for most CNF encodings used in practice. Thus, the resulting CNF formulas have fewer variables and fewer clauses.

Additionally, please note that the correctness of the proposed algorithm (as well as of the core-guided algorithms described in Section 4) follows from the fact that the value of the tentative lower bound τ grows only if the working formula is unsatisfiable and then, when it gets satisfiable, relies on the correctness of the Bin algorithm (BC or BCD in the case of the core-guided progression algorithms proposed below).

Input: $\mathcal{F} = \mathcal{F}_S \cup \mathcal{F}_H$
1 $(R, \mathcal{F}_W) \leftarrow (\emptyset, \mathcal{F})$
2 $(\lambda, j) \leftarrow (0, 0)$
3 **while** true **do**
4 $\quad \tau \leftarrow 2^j - 1$
5 $\quad$ **if** $\tau > \sum_{r_i \in R} w_i$ **then return** $\texttt{BC}(\mathcal{F}_W, R, \lambda, \emptyset)$
6 $\quad (\text{st}, \mathcal{U}, \mathcal{A}) \leftarrow \texttt{SAT}\,(\mathcal{F}_W \cup \texttt{CNF}\,(\sum_{r_i \in R} w_i r_i \leq \tau))$
7 $\quad$ **if** st $=$ true **then return** $\texttt{BC}(\mathcal{F}_W, R, \lambda, \mathcal{A})$
8 $\quad$ **else**
9 $\quad\quad$ **if** $\mathcal{U} \cap \mathcal{F}_S = \emptyset$ **then**
10 $\quad\quad\quad j \leftarrow j + 1$
11 $\quad\quad\quad \lambda \leftarrow \tau$
12 $\quad\quad$ **else** $(R, \mathcal{F}_W) \leftarrow \texttt{Relax}(R, \mathcal{F}_W, \mathcal{U} \cap \mathcal{F}_S)$

Algorithm 2: PRG_BC MaxSAT Algorithm

4 CORE-GUIDED ALGORITHMS

This section shows how to apply the lower bound refinement using geometric progression to algorithms that relax clauses on demand. In particular, the geometric progression approach of the previous section is applied to the BC [10] and the BCD [19] algorithms.

First the PRG_BC is presented which uses BC as the supporting algorithm. As in Algorithm 1, the main goal of the PRG_BC algorithm is to refine the lower bound λ with exponential increments. However, PRG_BC must take in consideration the current set of relaxation variables. Unlike Algorithm 1, in PRG_BC the soft clauses are not relaxed beforehand. Instead, and similarly to the BC algorithm, the set of relaxation variables R increases as (yet unrelaxed) soft clauses are identified in unsatisfiable cores. As a result, the lower bound λ and the progression step j are updated only when the current unsatisfiable core does not contain unrelaxed soft clauses.

The pseudo-code of PRG_BC is shown in Algorithm 2. Similar to Algorithm 1, PRG_BC maintains two values for the computation of the geometric increments: the known lower bound λ and the progression step j from which the cost τ is defined to test for unsatisfiability in each iteration ($\tau = 2^j - 1$, line 4). Whenever the cost to test τ becomes greater than the sum of weights of all soft clauses, then the BC algorithm is called with the last known lower bound (line 5).

In each iteration, PRG_BC calls the SAT solver on the current working formula $\mathcal{F}_W$, together with a constraint restricting the weighted sum of relaxation variables assigned value true to be smaller than or equal to τ (line 6). If the formula is satisfiable, then the BC algorithm is called with the last known lower bound λ and the assignment $\mathcal{A}$ as a witness of satisfiability (used in BC for the computation of an upper bound) (line 7).

Otherwise, an unsatisfiable core $\mathcal{U}$ is obtained from the SAT solver. If all clauses in $\mathcal{U}$ are relaxed (line 9), then the lower bound can be safely updated. As a result, j is incremented by one and λ is updated to the new lower bound τ (lines 10-11). If $\mathcal{U}$ contains unrelaxed soft clauses, then those soft clauses are relaxed (line 12) and values of λ and j remain unchanged.

Observe that in the worst case the PRG_BC relaxes all soft clauses and then proceeds as the iterative Algorithm 1.

Proposition 2. *The worst case number of SAT solver calls for the PRG_BC Algorithm is in $\mathcal{O}(\log C + m)$, where C is the cost of the MaxSAT solution, and m is the number of soft clauses.*

The second algorithm presented in this section is PRG_BCD which uses BCD as the supporting algorithm. As in the previous algorithms, the goal of PRG_BCD is to refine the lower bound. However,

Input: $\mathcal{F} = \mathcal{F}_S \cup \mathcal{F}_H$
1 $(\mathcal{F}_W, \mu) \leftarrow (\mathcal{F}, \sum_{(c_i, w_i) \in \mathcal{F}_S} w_i)$
2 $(\mathcal{D}, k) \leftarrow (\emptyset, 0)$
3 **while** true **do**
4 $\quad$ **if** $\sum_{\mathsf{d}_i \in \mathcal{D}} (2^{j_i} - 1) > \mu$ **then return** $\texttt{BCD}(\mathcal{F}_W, \mathcal{D}, \emptyset)$
5 $\quad \mathcal{F}_\mathcal{D} \leftarrow \bigcup_{\mathsf{d}_i \in \mathcal{D}} \texttt{CNF}\,(\sum_{r_l \in R_i} w_l r_l \leq (2^{j_i} - 1))$
6 $\quad (\text{st}, \mathcal{U}, \mathcal{A}) \leftarrow \texttt{SAT}\,(\mathcal{F}_W \cup \mathcal{F}_\mathcal{D})$
7 $\quad$ **if** st $=$ true **then return** $\texttt{BCD}(\mathcal{F}_W, \mathcal{D}, \mathcal{A})$
8 $\quad$ **else**
9 $\quad\quad \mathcal{D}_\mathcal{U} \leftarrow \texttt{Intersect}(\mathcal{D}, \mathcal{U})$
10 $\quad\quad$ **if** $\mathcal{U} \cap \mathcal{F}_S = \emptyset$ **and** $|\mathcal{D}_\mathcal{U}| = | < R_i, \lambda_i, j_i > | = 1$ **then**
11 $\quad\quad\quad \lambda_i \leftarrow 2^{j_i} - 1$
12 $\quad\quad\quad j_i \leftarrow j_i + 1$
13 $\quad\quad$ **else**
14 $\quad\quad\quad k \leftarrow k + 1$
15 $\quad\quad\quad (R_k, \mathcal{F}_W) \leftarrow \texttt{Relax}(\bigcup_{\mathsf{d}_i \in \mathcal{D}_\mathcal{U}} R_i,\ \mathcal{F}_W,\ \mathcal{U} \cap \mathcal{F}_S)$
16 $\quad\quad\quad j_k \leftarrow \min\ \{j \mid (2^j - 1) > \sum_{\mathsf{d}_i \in \mathcal{D}_\mathcal{U}} \lambda_i\}$
17 $\quad\quad\quad \mathcal{D} \leftarrow \mathcal{D} \setminus \mathcal{D}_\mathcal{U} \cup \{< R_k, \sum_{\mathsf{d}_i \in \mathcal{D}_\mathcal{U}} \lambda_i, j_k >\}$

Algorithm 3: PRG_BCD MaxSAT Algorithm

PRG_BCD also takes advantage of disjoint cores, similar to BCD. If two unsatisfiable cores of a formula do not intersect (in terms of the soft clauses), then two lower bounds can be considered (one for each of the disjoint cores), whose sum defines a lower bound for the overall formula. If an unsatisfiable core is found to intersect one or more of the previously found disjoint cores, then the associated lower bounds can be merged.

As such, PRG_BCD maintains the information of a disjoint core d_i in a structure $< R_i, \lambda_i, j_i >$, where R_i is the set of relaxation variables, λ_i is the known lower bound, and j_i defines the progression step such that a limit cost of $(2^{j_i} - 1)$ is to be tested. The current set of structures d_i's is kept in $\mathcal{D}$. We abuse the notation and refer to such structures as disjoint cores.

The pseudo-code of PRG_BCD is shown in Algorithm 3. In PRG_BCD, the overall cost to test in each iteration is not defined explicitly. It is in fact a sum of the costs for each of the current disjoint cores $\sum_{d_i \in \mathcal{D}} (2^{j_i} - 1)$. As in the previous algorithms, if such cost is greater than the sum of all the weights of the soft clauses, then the supporting algorithm BCD is called with the current set of disjoint cores $\mathcal{D}$ (line 4).

In each iteration, PRG_BCD calls the SAT solver on the current working formula $\mathcal{F}_\mathcal{W}$ together with a set of clauses $\mathcal{F}_\mathcal{D}$ that defines the tentative cost limit to each disjoint core. Unlike the previous algorithms that considered only one cardinality (or pseudo-Boolean) constraint, PRG_BCD considers one such constraint for each disjoint core in $\mathcal{D}$. All these constraints are encoded into $\mathcal{F}_\mathcal{D}$ (lines 5-6).

If the working formula is satisfiable, then BCD is called with the current set of disjoint cores $\mathcal{D}$, as well as model $\mathcal{A}$ as a witness of satisfiability (used in BCD to obtain an upper bound) (line 7).

If the working formula is not satisfiable, then an unsatisfiable core $\mathcal{U}$ is obtained. Function $\texttt{Intersect}(\mathcal{D}, \mathcal{U})$ is called to obtain the set $\mathcal{D}_\mathcal{U}$ of disjoint cores in $\mathcal{D}$ that have at least one soft clause in common with $\mathcal{U}$ (line 9).

If $\mathcal{U}$ does not contain unrelaxed soft clauses and $\mathcal{D}_\mathcal{U}$ contains only one disjoint core d_i, then PRG_BCD updates the lower bound λ_i and the progression step j_i (lines 11-12). Otherwise, there are disjoint cores that need to be merged and/or $\mathcal{U}$ contains unrelaxed soft clauses. Either way, a new disjoint core is created (with updated in-

dex k). The new set of relaxation variables R_k is the union of the relaxations variables of the disjoint cores in $\mathcal{D}_{\mathcal{U}}$, together with the relaxation variables obtained from relaxing the unrelaxed soft clauses of $\mathcal{U}$ (line 15). The known lower bound of the new disjoint core is the sum of the known lower bounds of the previous disjoint cores ($\sum_{d_i \in \mathcal{D}_{\mathcal{U}}} \lambda_i$). The new j_k is the minimum j whose cost $2^j - 1$ is greater than the new known lower bound (line 16). The new disjoint core thus obtained is added to $\mathcal{D}$, while the previous disjoint cores are removed (line 17).

Observe that in the worst case, PRG_BCD is going to consider m iterations each with an unsatisfiable core containing only one soft clause, thus creating m disjoint cores. Then it performs $m - 1$ iterations that merge the previous disjoint cores into a single disjoint core, and finally proceeds with iterations similar to the iterative Algorithm 1.

Proposition 3. *The worst case number of SAT solver calls for the PRG_BCD Algorithm is in $\mathcal{O}(\log C + m)$, where C is the cost of the MaxSAT solution, and m is the number of soft clauses.*

5 RELATED WORK

The area of MaxSAT algorithms have been active over the last decade, with many different algorithmic solutions proposed. Some of this work is surveyed in a number of recent publications [13, 2, 18]. Moreover, additional classes of MaxSAT algorithms have been proposed in recent years [8, 9]. For the algorithms described in this paper, the algorithms BC [10] and BCD [19] are used.

The use of geometric progressions to improve lower bounds has been studied in the recent past [17, 26], and can be related with earlier work, e.g. [22]. To our best knowledge, the use of geometric progression has not been considered in iterative algorithms for MaxSAT solving, nor have geometric progressions been integrated in core-guided approaches for MaxSAT.

Moreover, the use geometric progressions has recently been proposed in algorithms for the minimal set subject to a monotone predicate (MSMP) problem [15], that finds applications in the computation of minimal unsatisfiable subformulas and minimal correction subformulas. However, the approaches differ substantially, in that the algorithms described in this paper aim to refine a lower bound to compute a tight upper bound of the MaxSAT problem.

6 RESULTS

This section conducts an experimental evaluation of some of the algorithms proposed in this paper, namely PRG_BC and PRG_BCD, with state-of-the-art MaxSAT algorithms. The experiments were performed on an HPC cluster, each node having two processors E5-2620 @2GHz, with each processor having 6 cores, and with a total of 128 GByte of physical memory. Each process was limited to 4GByte of RAM and to a time limit of 1800 seconds. The set of problem instances considered includes all the *industrial* benchmarks from the *2013 MaxSAT Evaluation* [3] and contains 55 MaxSAT industrial instances, 627 partial MaxSAT industrial instances, and 396 weighted partial MaxSAT industrial instances. Thus, the total number of problem instances considered is 1078.

A prototype of a MaxSAT solver implementing all the progression-based algorithms described in Section 3 and Section 4 has been developed. The underlying SAT solver of the prototype implementation is Glucose [4]. The algorithms use the *modulo totalizer* cardinality encoding proposed in [20]. Additionally, the following heuristics were used: disjoint unsatisfiable core enumeration for computing a lower bound (e.g. see [16]) and complete Boolean multilevel optimization (complete BMO) (e.g. described in [14]).

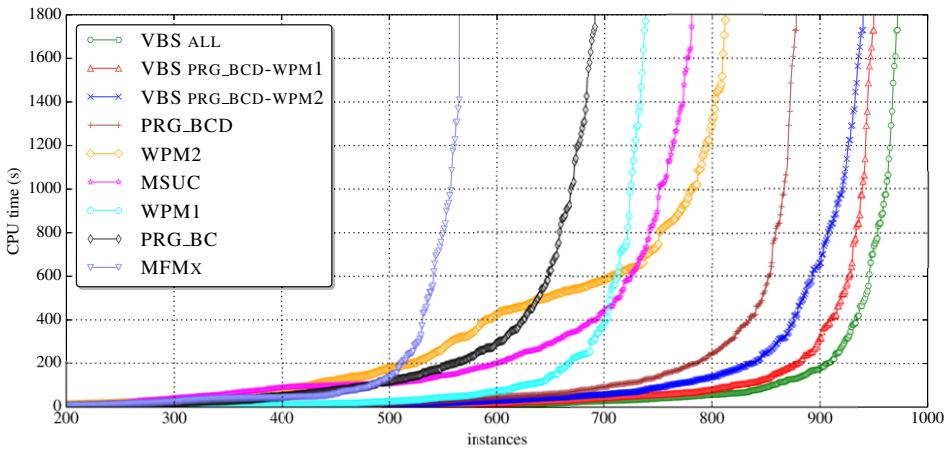
Besides, in the experiments we also used the following MaxSAT solvers, which took best places (among non-portfolio complete solvers) in different industrial categories of the 2013 MaxSAT Evaluation: QMaxSAT[5] 0.21 [12], WPM1 [2], WPM2 [1, 2], MSUnCore [19], and MiFuMaX[6]. QMaxSAT 0.21 implements the iterative linear search SAT-UNSAT and is also based on Glucose [4] as the backend SAT solver. The versions of WPM1 and WPM2 are from 2013. In contrast to other considered solvers, WPM1 and WPM2 are based on the Yices SMT solver[7], and so do not directly use a SAT solver as the backend. Also note that the configuration of MSUnCore used in the experiments is the enhanced version of the BCD algorithm [19]. In the experimental evaluations QMaxSAT, WPM1, WPM2, MSUnCore, and MiFuMaX are denoted by QMxS, WPM1, WPM2, MSUC, and MFMx, respectively.

Figure 1a shows a cactus plot illustrating the performance of the considered solvers on the total set of all instances in MaxSAT industrial, Partial MaxSAT industrial, and Weighted Partial MaxSAT industrial. PRG_BCD exhibits the best overall performance, being able to solve 878 instances out of 1078. WPM2 comes second with 812 instances solved. Thus PRG_BCD solves 8.13% more instances than WPM2. Also note (and it is also suggested by the scatter plot in Figure 3b) that there is a significant gap in the cactus plot, in terms of the running times, between PRG_BCD and WPM2. For example, whereas PRG_BCD can solve 575 instances within 40 seconds each, WPM2 can solve 329. To solve the same 575 instances as PRG_BCD, the run time limit of WPM2 must be increased until 335 seconds. Moreover, MSUC comes third with 781 instances solved within the 1800 seconds timeout. Detailed information about the number of solved instances is presented in Figure 1b.

The *virtual best solver* (VBS) among all the considered solvers is able to solve 972 instances, while the VBS incorporating PRG_BCD and WPM1 copes with 950 instances, and the VBS among PRG_BCD and WPM2 — 940. In order to get a deeper understanding of which solver contributes to the VBSes the most, we carried out a pairwise comparison between the best performing algorithms in the experiments. The comparison of PRG_BCD and MSUC is shown in Figure 3a while the difference between PRG_BCD and WPM2 is detailed in Figure 3b. The summaries of the comparison can be seen in Table 1a. Here we declare a winner if the difference between the run times exceeds 10%; otherwise, a tie is declared. Let us analyze the case of PRG_BCD and WPM2. Assuming the time spent by PRG_BCD is denoted by T_1 and the time of WPM2 is denoted by T_2, it is important to note that for 731 instances PRG_BCD is more than 10% faster than WPM2 (i.e. $100 \times \frac{T_2 - T_1}{T_2} > 10$), whereas WPM2 is more than 10% faster (i.e. $100 \times \frac{T_1 - T_2}{T_1} > 10$) for 184 instances. Finally, for the remaining 163 instances the run times are within 10% of each other.

Note that the QMxS version used in the experiments supports only unweighted partial formulas. Thus, we considered the set of Partial MaxSAT industrial instances separately, and analyzed the performance of all the solvers. The corresponding cactus plot is shown in Figure 2a. The cactus plot indicates that PRG_BCD performs better than other solvers and is able to solve 548 instances (out of 627). QMxS, which took the first place in the 2013 MaxSAT Evaluation for this set of benchmarks (for non-portfolio MaxSAT

[5] https://sites.google.com/site/qmaxsat/.
[6] http://sat.inesc-id.pt/~mikolas/sw/mifumax/.
[7] http://yices.csl.sri.com/.



(a) Cactus plot for all benchmarks

Solver	# solved (out of 1078)
PRG_BCD	878
WPM2	812
MSUC	781
WPM1	738
PRG_BC	691
MFMx	565

(b) Number of solved instances

Figure 1: Cactus plot and statistics for the all problem instances

Table 1: Pairwise comparison (win: >10% diff)

(a) MS+PMS+WPMS industrial

Row wins	WPM2	MSUC	PRG_BCD
WPM2	–	426	184
MSUC	426	–	91
PRG_BCD	731	788	–

(b) PMS industrial

Row wins	QMxS	WPM2	MSUC	PRG_BCD
QMxS	–	432	424	247
WPM2	116	–	258	114
MSUC	120	256	–	73
PRG_BCD	283	425	471	–

solvers), comes second with 534 instances solved. The performance of PRG_BCD and QMxS is compared by the scatter plot in Figure 3c. The pairwise comparison of the numbers of wins (for more than 10%) for QMxS and PRG_BCD is shown Table 1b. The number of solved instances per each solver is detailed in Figure 2b.

In summary, the experimental results indicate that the proposed progression-based algorithms represent one of the most robust approaches for Maximum Satisfiability, which can be successfully applied to a wide range of practical problem instances. Considering the total set of benchmarks, the PRG_BCD algorithm solves more instances than all the other algorithms considered in our experimental evaluation including WPM1, WPM2, and BCD. Moreover, the use of VBSes indicates that in most of the cases PRG_BCD is also the fastest algorithm in comparison to others. As for the Partial MaxSAT industrial set of benchmarks cosidered separately, PRG_BCD solves more instances than QMaxSAT even though QMaxSAT solely targets partial MaxSAT formulas, being optimized for those instances.

7 CONCLUSIONS

This paper describes the use of geometric progressions in MaxSAT algorithms based on iterative SAT solving, which may optionally use unsatisfiable core finding. The objective of using a geometric progression is to refine lower bounds on the MaxSAT solution, such that a guaranteedly tight upper bound is eventually identified. As a result, the number of calls to the SAT solver is guaranteed to be bounded by the logarithm of the optimum solution and not by the logarithm of the sum of soft clause weights. The paper shows how the use of geometric progressions can be used with plain iterative algorithms, but also with core-guided MaxSAT algorithms, namely the recently proposed BC and BCD algorithms [10, 19]. Experimental results, obtained on the (industrial) instances from the most recent MaxSAT evaluation, show consistent performance gains over state-of-the-art MaxSAT solvers, both in terms of the number of solved instances and in terms of pairwise performance comparison. This is also observed separately for the case of Partial MaxSAT industrial instances, where the new algorithms outperform QMaxSAT [12], a solver that is specific for partial MaxSAT. The experimental results also indicate that the practical deployment of MaxSAT solvers should consider the use of portfolios of MaxSAT solvers. This observation is also independently supported by the results of the 2013 MaxSAT evaluation.

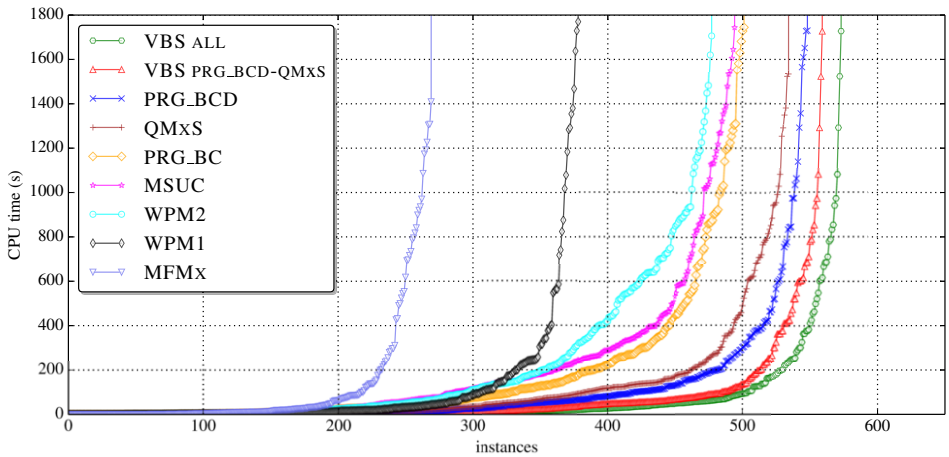
A number of research directions can be envisioned. Given the results for the VBS in Section 6, one line of research are algorithm portfolios for MaxSAT. Nevertheless, even considering all the algorithms compared in this paper, there are still more than 100 instances that cannot be solved to optimality. As a result, another line of research is the integration of recently proposed techniques for core-guided MaxSAT algorithms [2, 1].

ACKNOWLEDGEMENTS

This work is partially supported by SFI PI grant BEACON (09/IN.1/I2618), FCT grants POLARIS (PTDC/EIA-CCO/123051/2010) and ASPEN (PTDC/EIA-CCO/110921/2009), and INESC-ID multiannual PIDDAC funding PEst-OE/EEI/LA0021/2013.

REFERENCES

[1] C. Ansótegui, M. L. Bonet, J. Gabàs, and J. Levy. Improving WPM2 for (weighted) partial MaxSAT. In *CP*, pages 117–132, 2013.
[2] C. Ansótegui, M. L. Bonet, and J. Levy. SAT-based MaxSAT algorithms. *Artif. Intell.*, 196:77–105, 2013.
[3] J. Argelich, C. M. Li, F. Manyà, and J. Planes. The first and second Max-SAT evaluations. *JSAT*, 4(2-4):251–278, 2008. http://maxsat.ia.udl.cat/.
[4] G. Audemard and L. Simon. Predicting learnt clauses quality in modern SAT solvers. In *IJCAI*, pages 399–404, 2009.
[5] N. Bansal and V. Raman. Upper bounds for MaxSat: Further improved. In *ISAAC*, pages 247–258, 1999.

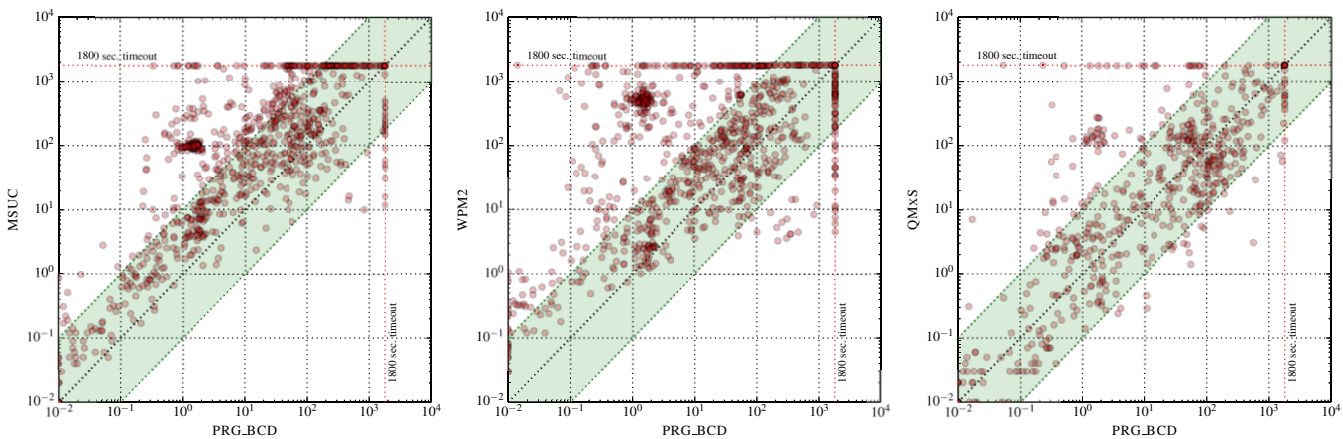


(a) Cactus plot for partial MaxSAT benchmarks

Solver	# solved (out of 627)
PRG_BCD	548
QMXS	534
PRG_BC	501
MSUC	494
WPM2	477
WPM1	378
MFMX	269

(b) Number of solved instances

Figure 2: Cactus plots and statistics for partial MaxSAT instances



(a) MSUC vs. PRG_BCD (b) WPM2 vs. PRG_BCD (c) QMXS vs. PRG_BCD (PMS industrial)

Figure 3: Scatter plots for pairwise comparisons

[6] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[7] E. Dantsin and A. Wolpert. MAX-SAT for formulas with constant clause density can be solved faster than in $O(2^n)$ time. In *SAT*, pages 266–276, 2006.

[8] J. Davies and F. Bacchus. Solving MAXSAT by solving a sequence of simpler SAT instances. In *CP*, pages 225–239, 2011.

[9] J. Davies and F. Bacchus. Exploiting the power of MIP solvers in MAXSAT. In *SAT*, pages 166–181, 2013.

[10] F. Heras, A. Morgado, and J. Marques-Silva. Core-guided binary search algorithms for maximum satisfiability. In *AAAI*, 2011.

[11] M. Jose and R. Majumdar. Cause clue clauses: error localization using maximum satisfiability. In *PLDI*, pages 437–446, 2011.

[12] M. Koshimura, T. Zhang, H. Fujita, and R. Hasegawa. QMaxSAT: A partial Max-SAT solver. *JSAT*, 8(1/2):95–100, 2012.

[13] C. M. Li and F. Manyà. MaxSAT, hard and soft constraints. In Biere et al. [6], pages 613–631.

[14] J. Marques-Silva, J. Argelich, A. Graça, and I. Lynce. Boolean lexicographic optimization: algorithms & applications. *Ann. Math. Artif. Intell.*, 62(3-4):317–343, 2011.

[15] J. Marques-Silva, M. Janota, and A. Belov. Minimal sets over monotone predicates in Boolean formulae. In *CAV*, pages 592–607, 2013.

[16] J. Marques-Silva and J. Planes. On using unsatisfiability for solving maximum satisfiability. *CoRR*, abs/0712.1097, 2007.

[17] C. Mencia, M. R. Sierra, and R. Varela. Intensified iterative deepening A* with application to job shop scheduling. *J. Intell. Manuf.*, 2013.

[18] A. Morgado, F. Heras, M. H. Liffiton, J. Planes, and J. Marques-Silva. Iterative and core-guided MaxSAT solving: A survey and assessment. *Constraints*, 18(4):478–534, 2013.

[19] A. Morgado, F. Heras, and J. Marques-Silva. Improvements to core-guided binary search for MaxSAT. In *SAT*, pages 284–297, 2012.

[20] T. Ogawa, Y. Liu, R. Hasegawa, M. Koshimura, and H. Fujita. Modulo based cnf encoding of cardinality constraints and its application to maxsat solvers. In *ICTAI*, pages 9–17. IEEE, 2013.

[21] S. D. Prestwich. CNF encodings. In Biere et al. [6], pages 75–97.

[22] J. Rintanen. Evaluation strategies for planning as satisfiability. In *ECAI*, pages 682–687, 2004.

[23] N. Robinson, C. Gretton, D. N. Pham, and A. Sattar. Partial weighted MaxSAT for optimal planning. In *PRICAI*, pages 231–243, 2010.

[24] O. Roussel and V. M. Manquinho. Pseudo-boolean and cardinality constraints. In Biere et al. [6], pages 695–733.

[25] S. Safarpour, H. Mangassarian, A. G. Veneris, M. H. Liffiton, and K. A. Sakallah. Improved design debugging using maximum satisfiability. In *FMCAD*, pages 13–19. IEEE Computer Society, 2007.

[26] M. J. Streeter and S. F. Smith. Using decision procedures efficiently for optimization. In *ICAPS*, pages 312–319, 2007.

[27] L. Zhang and F. Bacchus. MAXSAT heuristics for cost optimal planning. In *AAAI*, 2012.

[28] L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *DATE*, pages 10880–10885, 2003.