Minipref: A Tool for Preferences in SAT*

Carmine Dodaro[1] and Alessandro Previti[2]

[1] University of Calabria, Rende, Italy
dodaro@mat.unical.it
[2] Ericsson Research, Stockholm, Sweden
alessandro.previti@ericsson.com

Abstract. SAT solvers are used in a wide variety of applications, including hardware and software verification, planning and combinatorial design. Given a propositional formula, standard SAT solvers are able to decide whether it is satisfiable or not. In the first case, a model is returned as a witness of satisfiability. The returned model is typically selected according to an internal heuristic and no control on the assignment is offered to the final user. This paper serves as a description for our tool MINIPREF, a SAT solver extended to provide an optimal model with respect to a given preference over literals.

1 Introduction

The Boolean satisfiability problem (SAT) [6] is the problem of deciding whether a propositional formula is satisfiable. The importance of this decision problem is underlined by the many practical applications of SAT, which include software/hardware verification and planning among many others. Decision procedures for SAT, especially modern conflict-driven clause learning (CDCL) solvers, are routinely used to solve real-world instances with up to tens of millions of variables and clauses. The problem of finding an optimal solution to a SAT problem with preferences is a natural extension of the original problem which is of central importance in many areas of computer science [2,4,13,19]. For instance, in planning, besides the goals that have to be achieved, it is natural to have other soft goals that it would be desirable to satisfy [14]. Preferences are also essential to treat conflicting constraints [4] or for the computation of a minimal model [13]. Different approaches have been proposed to handle preferences [13,7,9], all of which work on top of modern SAT solvers. These approaches have been all implemented in a tool called SAT&PREF [7]. To the best of our knowledge, this is the only implemented SAT-based tool which deals with preferences directly inside the solver. However, SAT&PREF presents some limitations in its interface that make it not an ideal candidate on tasks that require iterative calls to a solver or enumeration tasks. More specifically, SAT&PREF doesn't provide any API for interacting with the solver. Preferences are directly specified inside the

input problem which is represented as an extension of the popular DIMACS format. In this paper, we present a tool for conveniently expressing preferences over literals which expose a clear and easy to use incremental interface. The tool introduces a level-based specification of the preferences, where a level is used to express a degree of desirability. This has some limits with respect to the more expressible DAG-based specification format, but it is more compact and convenient in many practical real-world applications [2,4,19]. Besides, our approach allows for better performance. When reasoning in terms of levels, variables within the same level are selected according to the original heuristic of the solver. This has a positive effect on the running time, since it is well known that imposing an ordering over the literals may lead to a performance loss [16]. The original MINISAT code has been modified without introducing any invasive change. This has the additional advantage to make it easy to integrate our implementation into other MINISAT-like solvers [3]. Finally, MINIPREF preserves the original MINISAT incremental interface, which is particularly useful in enumeration problems. The paper is organized as follows. Section 2 introduces the definitions used throughout the paper. In Section 3 the tool MINIPREF is presented. In Section 4 we provide some usage examples. An experimental evaluation is presented in Section 5. The paper concludes in Section 7.

2 Preliminaries

Let V be a countable set of propositional variables. A literal is either a variable x, or its negation $\neg x$. For a literal $l = x$, $\bar{l} = \neg x$, while for a literal $l = \neg x$, $\bar{l} = x$. This notation is extended to a set S of literals, i.e. $\overline{S} = \{\bar{l} \mid l \in S\}$. A clause is a set of literals and a formula φ is a set of clauses. The set of variables and literals occurring in φ are denoted as $vars(\varphi)$ and $lits(\varphi)$, respectively.

An assignment $A \subseteq (V \cup \overline{V})$ is a set of literals such that if a literal $l \in A$ then $\neg l \notin A$. Literals in A are said to be true. A clause C is satisfied w.r.t. an assignment A if $C \cap A \neq \emptyset$. A formula φ is satisfied w.r.t. an assignment if all clauses in φ are satisfied. An assignment that satisfies a formula φ is said to be a *model* of φ.

Preferences. Given a set of literals $L \subseteq lits(\varphi)$, a *preference* over literals in L is defined as a function $\omega : L \to \mathbb{N}$. Given a set of literals S and a number $i \geq 0$, $\Omega_i^S = \{l \mid l \in S, \omega(l) = i\}$. With a slight abuse of notation, when clear from the context, Ω_i denotes $\Omega_i^{lits(\varphi)}$, for a formula φ. For two literals l_1 and l_2 in L, l_1 is preferred to l_2, denoted $l_1 \prec l_2$, if $\omega(l_1) > \omega(l_2)$. For two models M_1 and M_2, M_1 is preferred to M_2, denoted $M_1 \prec M_2$, if there exists $j > 0$ such that $\Omega_j^{M_1} \supset \Omega_j^{M_2}$, and for all $i > j$, $\Omega_i^{M_1} = \Omega_i^{M_2}$. In the following, we assume that given a literal l such that $\omega(l) > 0$, then $\omega(\bar{l}) = 0$.

3 The Tool MINIPREF

In this section we present our tool MINIPREF, which is built on top of MINISAT [11]. In particular, MINIPREF works by overriding the original MINISAT heuristic. Given a preference $l \prec l'$, MINIPREF branches first on l before selecting l'. Intuitively, this is necessary in order to avoid that the less preferred literal l' can affect the assignment to the most preferred literal l. At each step, MINIPREF selects a new variable following the order imposed by the preferences and assigns to it the preferred value. This is always possible, unless the opposite value is implied by the current partial assignment. In modern SAT solvers, heuristics have a significant impact on performance [5]. In order to prevent performance loss, MINIPREF keeps interference with the heuristic to a minimum. In MINISAT the next variable to branch on is selected from a heap. In our implementation variables are distributed in more than one heap. More specifically, MINIPREF is organized as a stack of heaps, one for each level i such that Ω_i is not empty. Intuitively, levels serve to represent the relative importance of variables. In practice, given two levels n and m, with $n > m$, all the literals in Ω_n are selected before those in Ω_m, while all the literals within the same level are selected according to the original MINISAT policy. Note that, differently from the MINISAT implementation, in our tool random choices are allowed only within the same level. The tool exposes an interface for specifying for each literal its level. Literals without a specified preference are assumed to be at level 0 and their the standard MINISAT heuristic is not modified for them. Indeed, it is important to emphasize here that MINIPREF acts as a standard SAT solver, when no preference is specified. The implementation of the tool guarantees that the first discovered assignment satisfying the input formula is a most preferred model.

Interface. MINIPREF is implemented on top of MINISAT and it extends the original C++ interface with two additional methods:

1. *setPreference(Lit l, int lv)*: This method adds the variable $var(l)$ into the heap at level lv. If no heap at level lv already exists a new one is created. Then, whenever $var(l)$ is selected by the branching heuristic of MINISAT, $var(l)$ is assigned to true if l is positive, and to false otherwise.
2. *removePreference(Lit l)*: This method remove $var(l)$ from the current heap to the heap at level 0 (i.e the default MINISAT level) and restore the default behaviour of the MINISAT heuristic for this variable.

The tool MINIPREF comes with a Python wrapper, PYMINIPREF, which extends the tool PYMINISOLVERS, originally implemented in [17]. The aim of the Python wrapper is to allow for quick prototyping of ideas. The Python interface mostly resembles the C++ interface, with the only difference being an *int* as a first parameter of *setPreference* and *removePreference* instead of a *Lit*. The intended meaning of the *int* matches the one of the DIMACS format, with l and $-l$ used to denote a positive and negative literal respectively. In addition, PYMINIPREF comes with a *Dimacs* class to deal with input formulas. The class

stores the formula before it is pushed into the solver, thus giving the opportunity to apply modifications to it (e.g adding selector variables) before execution. It also keeps information about soft/hard clauses and their weights. At the moment, such a class is not available for the C++ interface, but this will be added in future releases.

MINIPREF and PYMINIPREF both support incremental SAT solving under assumption literals even in combination with preferences. Incremental SAT solving is beneficial when dealing with problems requiring iterative calls to a solver. Both MINIPREF and PYMINIPREF can return a core in case a formula is unsatisfiable or a model otherwise. The interface that returns a model or a core is left unchanged with respect to MINISAT and PYMINISOLVERS, respectively.

4 Usage Examples

The interface presented in the previous section can be used for performing different computational tasks. As usage examples, in this section we report on two algorithms, one for the computation of a Minimal Correction Subset (MCS) and one for the computation of the backbones of a propositional formula. In the following, algorithms are presented in a Python-like language. Given the space limit, we report here only on a simplified version to understand the algorithms; for the full code we refer the reader to [10].

Computation of a MCS. Algorithm 1 reports on a strategy for the computation of a MCS of a given formula, say φ, that is split into two sets of clauses representing the hard and soft clauses, respectively. The algorithm returns a minimal subset C of the soft clauses such that $\varphi \setminus C$ is satisfiable. In the following, we assume that the set of hard clauses is satisfiable. The working principle of the algorithm is quite straightforward: each soft clause c is processed and a fresh literal, called *selector literal*, is added to c. Intuitively, since all the selector literals do not appear elsewhere in the φ, then any set M including all of them is a model of the soft clauses. Therefore, in order to compute a minimal set of true selector literals it is sufficient to associate the level 1 to the preference of the complement of the selector literals (line 9).

Computation of backbones. The algorithm described in the following shows the advantage of providing an incremental interface for the specification of preferences. Indeed, it is based on multiple calls to the solver, where preferences are updated after each call.

A model M of a formula φ is said to be *minimal* with respect to a set O of objective literals if there is no model M' of φ such that $(M' \cap O) \subset (M \cap O)$. The algorithm described in the following, and reported as Algorithm 2, takes advantage of this notion. In particular, given a formula φ, it first searches for models of φ that are minimal with respect to the current set of candidates C. The model returned by `S.solve()` either discard some candidate from C, which would lead to a new iteration of the strategy, or are such that all literals in C are

Algorithm 1: Computation of a MCS

```
1  S = pyminipref.MinisatSolver()
2  D = pyminipref.Dimacs()
3  V = []
4  for c in D.clauses : S.addClause(c)                    // Add hard clauses
5  for c in D.soft :
6  |  V.append(S.newVar())                        // Add a selector variable
7  |  c.append(V[-1])          // Add the selector variable to the clause
8  |  S.addClause(c)                          // Add the clause to the solver
9  for v in V : S.setPreference(-v, 1)
10 S.solve()
11 mcs = [c[:-1] for c in D.soft if S.isTrueInModel(c[-1])]
12 print (mcs);
```

Algorithm 2: Computation of Backbones

```
1  S = pyminipref.MinisatSolver()
2  D = pyminipref.Dimacs()
3  C = []
4  for c in D.clauses : S.addClause(c)                        // Add clauses
5  S.solve()                                        // Compute first model
6  for v in range(1, S.nvars()) :
7  |  if S.isTrueInModel(v) : C.append(v)
8  |  else : C.append(-v)
9  |  S.setPreference(-C[-1], 1)
10 while True :
11 |  S.solve()
12 |  newC = [l for l in C if S.isTrueInModel(l)]    // update candidates
13 |  if len(newC) == len(C) : return C          // no candidates removed
14 |  for l in C and l not in newC : S.removePreference(l)
15 |  C = newC
```

true. In the latter case, the set C only contains backbones of φ, and therefore the algorithm terminates returning C.

Note that the minimality of each model with respect to C is enforced by associating the level 1 to the preference of the complements of literals in C (line 9), and then by removing such preferences when literals are proven to be not part of the backbones (line 14).

5 Experiments

The performance of the tool MINIPREF presented in this paper was assessed empirically on three benchmarks

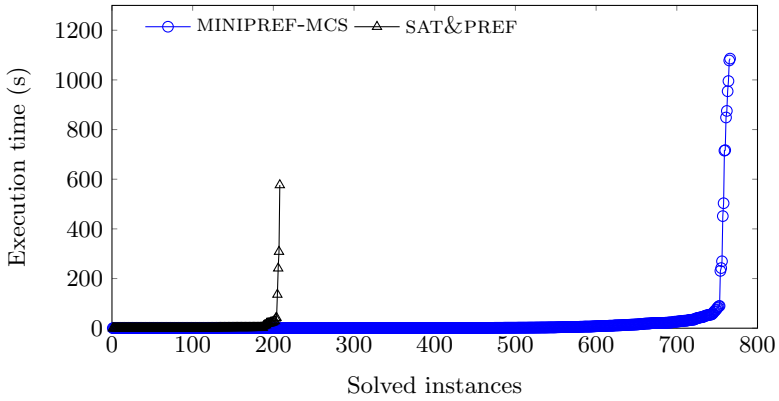

Fig. 1. Comparison of the performance of MINIPREF-MCS and SAT&PREF on instances of benchmark (i).

(i) all instances from [18] for the computation of one MCS. In particular, we considered all the instances used for the computation of a single MCS;
(ii) *all structured* instances from [7] used to test the performance of SAT&PREF and downloaded from [8];
(iii) all the 783 instances from [15] for the computation of backbones of propositional formulas.

For (i) and (ii), we compared an implementation of Algorithm 1 based on the Python interface of MINIPREF (referred to as MINIPREF-MCS) with the tool SAT&PREF [7] using the best performing strategy according to [7], i.e., *sat&pref*. Note that both MINIPREF and SAT&PREF use MINISAT [11] as underlying SAT solver. For (iii), we compared two different implementations of Algorithm 2, based on the C++ (referred to as MINIPREF-BBC) and on the Python library of MINIPREF (referred to as MINIPREF-BBP), respectively. The experiment was run on an Intel CPU 2.4 GHz with 16 GB of RAM. Running time was limited to 1200 seconds.

Concerning benchmark (i), results are reported in the cactus plot of Figure 1. We recall that in a cactus plot a line is reported for each tested solver; where instances are ordered by solving time and a point (i, j) in the graph represents that the i-th instance is solved in j seconds. It is possible to observe that MINIPREF-MCS 558 instances more than SAT&PREF. However, from the analysis of the results, we noticed that SAT&PREF is buggy in the majority of the tested instance. In particular, the solver terminates without printing neither the correct solution nor an error message. Therefore, in order to perform a fair comparison, we restricted our analysis to the 231 instances where SAT&PREF was either correct or it exceeded the time limit. In this setting, SAT&PREF solves 208 instances, whereas MINIPREF-MCS obtains a better performance solving 215 instances.

A slight improvement of the performance can be also observed on instances of the benchmark (ii) as shown in the scatter plot of Figure 2. We recall that

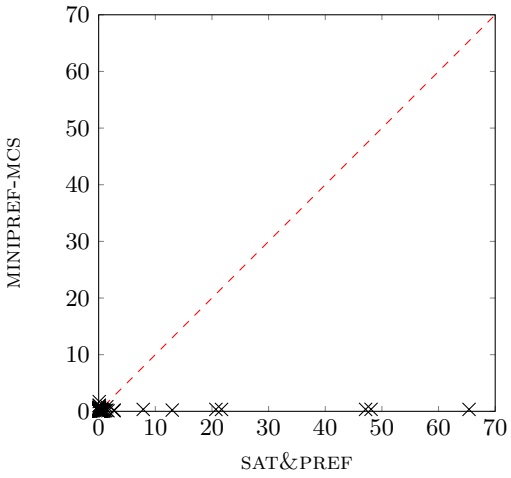

Fig. 2. Instance-wise comparison between MINIPREF-MCS and SAT&PREF on benchmark (ii).

in a scatter plot a point (x, y) is reported for each instance, where x and y are the solving times of the compared systems. Points standing below the diagonals represent instances where the system reported on the x-axis was slower than the system reported on the y-axis. In this setting, it is possible to observe that MINIPREF-MCS outperforms SAT&PREF since the majority of the points are below the diagonal. Moreover, we report that the latter is able to solve all the 157 instances with a total running time of approximately 260 seconds, whereas MINIPREF-MCS solves all the instances with a total running time of 32 seconds.

Finally, concerning benchmark (iii), we report that the two tested solvers, MINIPREF-BBC and MINIPREF-BBP, are almost on par, solving 670 and 674 instances, respectively. Interestingly, the Python version obtains slightly better performance than the C++ version. This can be explained by a heuristic factor, due to the different implementation of the two solvers. Indeed, candidates in MINIPREF-BBC are handled using a data structure called *unordered set* which in general does not preserve the order of literals added to it, while in MINIPREF-BBP they are handled using a Python list. Therefore, the order of processed literals is different in the two versions, thus the solvers perform different heuristic choices.

6 Related Work

The SAT problem with preferences finds many practical applications, but to date the only available solver (SAT&PREF) is outdated and mostly buggy. SAT&PREF implements three different procedures to enforce satisfaction of the preferences over literals, namely *optsat-hs* [13], *optsat-bf* [9] and *sat&pref* [7]. Our tool is similar in spirit to *optsat-hs*, since they both work by modifying the heuristic.

However, differently from *optsat-hs*, an ordering over the literals is enforced without modifying the scoring mechanism of MINISAT. We compare our tool against the technique referred to as *sat&pref*, which according to [7] is the most performing one. As shown in the experimental section, our tool outperforms SAT&PREF. Moreover, in SAT&PREF preferences over literals have to be specified into the input formula, which makes using the tool unpractical for tasks requiring multiple calls to the solver. MINIPREF offers an easy interface to use the solver that can be adopted as a good starting point for dealing with preferences. The underline MINISAT solver provides good guarantees in terms of performance out-of-the-box, as shown empirically in the experimental section. Moreover, to the best of our knowledge, this is the first tool for SAT preferences exposing a programmatic interface.

Preferences can be also specified in other languages, like Answer Set Programming (ASP), and several ASP solvers, like CLINGO [12] and WASP [1] are able to deal with preferences. Concerning CLINGO, preferences can be expressed by using the class of `HeuristicType`. However, the interface of CLINGO is more complex since it is mainly dedicated to ASP programs, and therefore requires an additional effort to convert the SAT instance into an ASP program. Concerning WASP, it offers a MINISAT-like interface to specify the SAT formula and it allows to add preferences among literals in a way that is similar to the one of MINIPREF. However, WASP supports only one level of preferences, therefore it is not possible to specify a total order among literals. Moreover, the interface of WASP is available in C++ only, whereas MINIPREF also offers a Python interface.

Finally, we mention that the current implementation of MINIPREF does not allow to compute cardinality-minimal models; therefore it is not suitable for the computation of (partial) MaxSAT solutions.

7 Conclusion

In this paper we presented a new tool, MINIPREF, built on top of a CDCL SAT solver, which allows for expressing preferences over literals. An experimental analysis conducted on publicly-available instances shows that MINIPREF outperforms the tool SAT&PREF. Possible future directions include a new efficient data structure to handle a DAG-based preference specification, the possibility to provide different heuristics for different levels and other approaches as presented in [7]. Finally, we mention that MINIPREF is open-source and available at [10].

References

1. Alviano, M., Amendola, G., Dodaro, C., Leone, N., Maratea, M., Ricca, F.: Evaluation of disjunctive programs in WASP. In: Proc. of LPNMR. Lecture Notes in Computer Science, vol. 11481, pp. 241–255. Springer (2019)
2. Alviano, M., Dodaro, C., Järvisalo, M., Maratea, M., Previti, A.: Cautious reasoning in ASP via minimal models and unsatisfiable cores. Theory and Practice of Logic Programming **18**(3-4), 319–336 (2018)

3. Audemard, G., Simon, L.: On the glucose SAT solver. International Journal on Artificial Intelligence Tools **27**(1), 1–25 (2018)
4. Bacchus, F., Davies, J., Tsimpoukelli, M., Katsirelos, G.: Relaxation search: A simple way of managing optional clauses. In: Proc. of AAAI. pp. 835–841. AAAI Press (2014)
5. Biere, A., Fröhlich, A.: Evaluating CDCL variable scoring schemes. In: Proc. of SAT. Lecture Notes in Computer Science, vol. 9340, pp. 405–422. Springer (2015)
6. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)
7. Di Rosa, E., Giunchiglia, E.: Combining approaches for solving satisfiability problems with qualitative preferences. AI Communications **26**(4), 395–408 (2013)
8. Di Rosa, E., Giunchiglia, E., Maratea, M.: sat&pref, `http://www.star.dist.unige.it/index.php?option=com_uhp2&Itemid=&task=viewpage&user_id=86&pageid=77`
9. Di Rosa, E., Giunchiglia, E., Maratea, M.: A new approach for solving satisfiability problems with qualitative preferences. In: Proc. of ECAI. Frontiers in Artificial Intelligence and Applications, vol. 178, pp. 510–514. IOS Press (2008)
10. Dodaro, C., Previti, A.: minipref, `https://github.com/apreviti/minipref`
11. Eén, N., Sörensson, N.: An extensible sat-solver. In: Proc. of SAT. Lecture Notes in Computer Science, vol. 2919, pp. 502–518. Springer (2003)
12. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Wanko, P.: Theory solving made easy with clingo 5. In: Proc. of ICLP (Technical Communications). OASICS, vol. 52, pp. 2:1–2:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2016)
13. Giunchiglia, E., Maratea, M.: Solving optimization problems with DLL. In: Proc. of ECAI. Frontiers in Artificial Intelligence and Applications, vol. 141, pp. 377–381. IOS Press (2006)
14. Giunchiglia, E., Maratea, M.: Planning as satisfiability with preferences. In: Proc. of AAAI. pp. 987–992. AAAI Press (2007)
15. Janota, M., Lynce, I., Marques-Silva, J.: Algorithms for computing backbones of propositional formulae. AI Communications **28**(2), 161–177 (2015)
16. Järvisalo, M., Junttila, T.A., Niemelä, I.: Unrestricted vs restricted cut in a tableau method for boolean circuits. Annals of Mathematics and Artificial Intelligence **44**(4), 373–399 (2005)
17. Liffiton, M.: PyMiniSolvers, `https://github.com/liffiton/PyMiniSolvers`
18. Marques-Silva, J., Heras, F., Janota, M., Previti, A., Belov, A.: On computing minimal correction subsets. In: Proc. of IJCAI. pp. 615–622. IJCAI/AAAI (2013)
19. Previti, A., Järvisalo, M.: A preference-based approach to backbone computation with application to argumentation. In: Proc. of SAC. pp. 896–902. ACM (2018)