Local Search with Bootstrapping

Lengning Liu and Mirosław Truszczyński

Department of Computer Science, University of Kentucky,
Lexington, KY 40506-0046, USA

Abstract. We propose and study a technique to improve the performance of those local-search SAT solvers that proceed by executing a prespecified number of tries, each starting with an element of the space of all truth assignments and performing a prespecified number of local-search steps (flips). Based on the input theory T, our method first constructs a collection of its relaxations, that is, theories whose models are easy to compute and "almost" satisfy T. It then uses a local-search algorithm to compute models of the relaxed theories and, finally, uses these models as starting points for tries when executing the local search algorithm on T. To construct relaxations our method takes advantage of high-level representation of search problems, which separate the specification of a search problem from the description of its particular instances. The method is general. We applied it to WSAT, a local-search SAT solver for CNF theories, and to WSAT(cc), a local-search SAT algorithm for theories in the language of propositional logic with cardinality constraints. Experimental results demonstrate its effectiveness for both local-search algorithms we studied.

1 Introduction

We propose and study a general technique to improve the performance of certain classes of local-search SAT solvers in computing solutions to instances of search problems. Our method takes advantage of high-level representations of search problems, which separate the specification of a search problem from the description of its particular instances.

Typically, in order to solve a search problem Π for an instance I, we construct a propositional theory T so that models of T determine solutions. We then use a SAT solver to compute models of T and reconstruct from them the corresponding solutions. However, once a propositional theory representing a search problem and its instance is formed, the connection to the search problem and the structure of its instances is no longer clear. Consequently, SAT solvers view these theories as if they were arbitrary propositional theories. They do not take advantage of the specification of the search problem and properties of the structure of its instances to fine-tune the way they compute models.

Recently, researchers recognized that and proposed formalisms for representing search problems and their instances based on predicate logic [5,6] and the existential fragment of second-order logic [2]. A search problem Π (more precisely, a set of constraints specifying it) is represented by a collection of clauses (*program*) in the logic P_Π, and an instance I (input *data* to the problem Π) is given as a collection D_I of ground atoms, that is, an instance of some prespecified relational schema. To obtain a propositional representation of the search problem Π and its instance I, we *ground* the program P_Π with respect to constants specified in the data set D_I.

This approach to building propositional representations of instances of search problems makes explicit the constraints of a search problem and the data instance schema. Consequently, it opens a possibility to design SAT solvers that can exploit properties of the problem statement and of the structure of the data instances. Two such solvers were proposed recently. [2] described a method that delays certain constraints based on the analysis of the problem specification. [6] introduced an approach to compute models directly from a predicate-logic specification of constraints and a description of a data instance.

In this paper, we present a method that takes advantage of high-level program-data representations of search problems as data-program pairs in the logic PS+ [5] to improve the performance of local-search SAT solvers for propositional theories, possibly extended with boolean combinations of *pseudo-boolean* constraints. Local-search satisfiability methods (we refer to [8] for more details on that topic) that we are interested in proceed by executing a prespecified number, nt, of *tries*. A try starts with a random element of the space of all truth assignments and performs a prespecified number, nf, of *flips*. Each flip modifies the current truth assignment by changing truth values of a small number of atoms so that to optimize some objective function, for instance to minimize the number of clauses that become unsatisfied after the flip. If at any time, the current truth assignment satisfies all clauses, the method reports that assignment and stops. If all tries run their course without finding a satisfying assignment, the method reports failure and then terminates. Local-search methods are *incomplete*. The failure means only that no satisfying assignment was found within nf flips in any of the nt tries, and not that no satisfying assignment exists at all.

Intuitively, the closer the initial truth assignment of a try to a satisfying assignment (model), the better the chance that the try will succeed. In this paper, we introduce a method to select initial truth assignments for tries based on that intuition. Specifically, let A be a local-search algorithm. Based on the input theory T, our method first constructs a collection of *relaxations* of T, that is, theories whose models are easy to compute and "almost" satisfy T. It then uses the algorithm A to compute models of the relaxed theories and, finally, uses these models as starting points for tries when A is executed on T. In a sense, in our method, *the local-search algorithm A relies on itself to improve its own performance* and so, we call the method — *bootstrapping*.

The main difficulty in bootstrapping is to construct for a given input theory T its relaxations. To address that issue we exploit representations of search problems as theories in the logic PS+ [5], a version of predicate logic extended to model cardinality constraints. Bootstrapping is a general method and can be used with every local-search algorithm, which follows the template described above and allows the user to specify starting truth assignments for tries. We applied it to $WSAT$ [10], one of the most successful local-search SAT solvers for CNF theories, and to $WSAT(cc)$ [9], a local-search SAT algorithm for computing models of theories in the language of propositional logic extended with cardinality constraints, a class of pseudo-boolean constraints. Our paper provides a description of the bootstrapping method and presents experimental results demonstrating its effectiveness for both local-search algorithms that we studied.

2 Bootstrapping for local-search algorithms

Let A be any local-search algorithm that falls into the category we described above. We write $A(T, nt, s)$ to specify a call to the algorithm A for a propositional theory T, assuming that A executes nt tries and that each try starts with a truth assignment s. If s is not a complete assignment, then the truth values of unassigned atoms are initialized for each try uniformly at random. To improve the performance of A, we propose a method called *bootstrapping*.

By $At(T)$ we denote the set of atoms appearing in theory T. A propositional theory T' is a *relaxation* of a propositional theory T if for every model M of T, $M \cap At(T')$ is a model of T'. In general, not every model of T' extends to a model of T. However, if T' is "close" to T, one might expect that models of T' *almost* satisfy T and, consequently, can serve as good initial assignments for tries when A searches for models of T. Bootstrapping exploits that idea. We will discuss how to construct relaxations in the next section. Here, we describe how bootstrapping uses them to organize search for models.

Given an input theory T, let $S = (T_1, \ldots T_k)$ be a sequence of theories such that $T_k = T$ and, for every i, $1 \leq i \leq k-1$, T_i is a relaxation of T_{i+1}. Bootstrapping starts by executing $A(T_1, nt, \emptyset)$. That call to A searches for models of T_1 with initial assignments for tries generated randomly. If a model, say M_1, is found, bootstrapping proceeds by executing $A(T_2, nt, M_1)$. That is, A now uses M_1 to generate starting assignments for tries by randomly extending it to complete assignments for T_1. If the call $A(T_2, nt, M_1)$ succeeds and finds a model, say M_2, bootstrapping executes $A(T_3, nt, M_2)$, and so on. If at some level i, $A(T_i, nt, M_{i-1})$ fails to find a model, bootstrapping returns to level 1 and starts again by executing $A(T_1, nt, \emptyset)$, up to a prespecified number of *restarts*, rst. If all restarts fail, the method terminates with failure. If bootstrapping descends to the level k and the call $A(T_k, nt, M_{k-1})$ succeeds, the model it returns is a model of the input theory T. Bootstrapping then returns that model and terminates with success. We provide a more formal description of the method in Figure 2.1.

There are alternative ways in which one can organize the bootstrapping search. In particular, one can allow the method to backtrack to intermediate levels before it ultimately backtracks to the top. Full version of this work will contain experimental evaluation of several different strategies.

3 Logic PS+ and relaxations

The key difficulty in bootstrapping is to generate a sequence of relaxations $S = (T_1, \ldots, T_k)$, in which models of theory T_i can be found faster than models of T_{i+1} can, and in which models of T_i indeed satisfy most of the clauses in T_{i+1}. We discuss that issue now. We focus on propositional theories that arise when

```
Input: $S = (T_1, \ldots T_k)$ — a sequence of relaxations of a theory $T$ (with $T_k = T$);
$rst$ — an integer specifying the number of restarts
**repeat** $rst$ times
  $fail$ := **false**;  $i$:= 0;  $M_0 = \emptyset$;
  **while** (**not** $fail$)
    $i$:= $i + 1$;
    $A(T_i, nt, M_{i-1})$;
    **if** (model found)
      denote it with $M_i$;
      **if** ($i = k$) output $M_k$ and terminate
    **else** $fail$:=**true**
```

Fig. 2.1. Bootstrapping of a local-search algorithm A.

solving search problems, and we take advantage of representations of search problems in the language of predicate logic, an approach advanced in [5,6,2]

The logic we use is the logic PS+. For details on that logic, we refer the reader to [5]. Due to space restrictions, we only illustrate its features by discussing it in the context of the graph k-coloring problem, where the goal is to assign colors from a given set to vertices of a given graph so that no two adjacent vertices are colored the same (or to determine that no such coloring exists).

We represent a data instance, a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, and a set of k colors $\{1, \ldots, k\}$, as the set of ground atoms

$$D(G,k) = \{vtx(v) \colon v \in V\} \cup \{edge(v,w) \colon \{v,w\} \in E\} \cup \{color(i) \colon 1 \leq i \leq k\}.$$

It can be viewed as an instance of the relational schema consisting of unary *data predicate* symbols vtx and $color$ and a binary *data predicate* symbol $edge$.

To specify the problem, we encode its constraints as clauses in a version of the language of predicate calculus, determined by predicate symbols vtx, $edge$ and $color$, and an additional binary predicate symbol $clrd$ that models assignments of colors to vertices.

C1: $clrd(X,C) \rightarrow vtx(X)$
C2: $clrd(X,C) \rightarrow color(C)$
C3: $vtx(X) \rightarrow clrd(X,C) : color(C)$
C4: $clrd(X,C) \wedge clrd(X,D) \rightarrow (C = D)$
C5: $edge(X,Y) \wedge clrd(X,C) \wedge clrd(Y,C) \rightarrow \perp$.

The construct $clrd(X,C) : color(C)$ in the clause (C3), not present in the standard language of predicate calculus, represents the disjunction of all atoms of the form $clrd(X,c)$, where c is a color.

Intuitively, the conditions (C1) and (C2) state that the only objects that get colored are vertices and the only objects with which they can be colored are colors. (C3) states that each vertex X gets assigned at least one color. (C4) enforces that each vertex is assigned at most one color. (C5) ensures that two vertices connected by an edge are assigned different colors.

The precise meaning of this representation is determined by grounding. To produce it, we stipulate that ground atoms built of vtx, $edge$ and $color$ are true *if and only if* they appear in $D(G,k)$[1]. For each clause, we generate all its ground instances with respect to constants appearing in $D(G,k)$ and simplify them by taking into account truth assignments determined by $D(G,k)$. The resulting theory consists of the following clauses[2] and coincides with a standard propositional encoding of the graph k-coloring problem:
C3$_{gr}$: $clrd(x,1) \vee clrd(x,2) \vee \ldots \vee clrd(x,k)$, for every $x \in V$
C4$_{gr}$: $clrd(x,c) \wedge clrd(x,c') \rightarrow \perp$, for every $x \in V$ and $1 \leq c \neq c' \leq k$
C5$_{gr}$: $clrd(x,c) \wedge clrd(y,c) \rightarrow \perp$, for every edge $\{x,y\}$ and $1 \leq c \leq k$.

Logic PS+ also supports *cardinality atoms*, that is, explicit constructs to model cardinality constraints. For instance, we could represent clauses (C3) and (C4) with a single clause
CC: $vtx(X) \rightarrow 1\{clrd(X,C)[C] : color(C)\}1$

[1] That is, $D(G,k)$ provides a complete description of the data instance.
[2] Clauses (C1) and (C2) do not contribute to the simplified grounding of the theory.

stating that each vertex X gets exactly one color C. That clause grounds to the following collection of clauses in the language of propositional logic with cardinality atoms[3]:
CC$_{gr}$: $1\{clrd(x,c) :\ c \in \{1,\ldots,k\}\}1$, for every $x \in V$.

Given a representation of a search problem in the logic PS+, we can generate not only the corresponding propositional theory but also its relaxations. Here are some possibilities.

Enlarge or shrink extensions of data predicates. For instance, in the graph-coloring problem, removing some elements from the extension of the predicate vtx (and the corresponding elements from the extension of $edge$) and grounding the resulting theory yields a relaxation of the original one. In the case of representations of the hamiltonian cycle problem, enlarging the extension of the predicate $edge$ makes the problem easier and leads to relaxations.

Increase or decrease constants. Representations of search problems often specify bounds on the size of a structure that is sought, for instance, an upper bound on a vertex cover, or a lower bound on the size of a clique. By increasing the constant in the first case (decreasing it, in the second case) we obtain relaxations of the initial theory.

Modify the problem. In the graph coloring problem, we can introduce an additional color and a new clause bounding, by some constant, the number of times this new color can be used. When the constant is set to 0, we obtain the original theory. Moreover, the larger the constant used, the "easier" the theory.

Our ultimate goal is to produce relaxations automatically from problem specifications in the logic PS+. Some of the methods we listed above are clearly amenable to automation and we are presently pursuing this direction. However, in this work, our objective is to demonstrate the effectiveness of bootstrapping and in our experiments, we use relaxation sequences that we constructed ourselves based on PS+ specifications of search problems we studied.

4 Experimental Results

We applied bootstrapping to two local-search solvers: $WSAT$ [10] and $WSAT(cc)$ (in its *virtual-break-count* version) [9]. Both algorithms proceed in tries and provide the user with an option to specify (fully or partially) starting truth assignments for tries. Thus, they are amenable to our approach. We write $WSAT$+$Boot$ and $WSAT(cc)$+$Boot$ for their "bootstrapped" versions.

We will now present results of experiments comparing the performance of $WSAT$ and $WSAT(cc)$ with their bootstrapped versions, and demonstrating that bootstrapping is effective. The benchmark problems we used in experiments are: the graph 4-coloring problem, the graph vertex-cover problem, the Schur-number problem and the van der Waerden-number problem. The graph-coloring and vertex-cover problems are well known. In the instance $S(n,k)$ of the Schur-number problem, given positive integers n and k, the goal is to compute a partition of $\{1,2,\ldots,n\}$ into k parts so that all parts are sum-free [7]. In the instance $W(n,k,l)$ of the van der Waerden problem, given positive integers n, k and l, the goal is to find a partition of $\{1,2,\ldots,n\}$ into k parts, so that no part contains an arithmetic progression of length l [7,4].

We used the same number of flips per try and the same noise ratio when comparing $WSAT$ and $WSAT(cc)$ with their bootstrapped counterparts. We report the number of tries each takes to find the first model (for bootstrapped versions, the count includes the tries from all levels of the bootstrapping search). That measure is closely correlated with the running time, but is machine-independent. All of the following experiments were done on two computers: Intel P4 1.5GHz with 1GB memory and Intel P4 3.2GHz with 1GB memory. Both platforms use Slackware Linux with kernel version 2.4.24 and gcc version 3.2.2.

For the graph 4-coloring problem, we randomly generated three families of 1000-vertex graphs containing 3900, 3910 and 3920 edges, respectively. For each instance graph $G=(V,E)$, we used the theory representing its 4-colorability based on the clauses (C1)-(C5) ((C1), (C2), (C5) and (CC), respectively) to test $WSAT$ ($WSAT(cc)$, respectively). To test $WSAT$+$Boot$ and $WSAT(cc)$+$Boot$, we constructed relaxations of these theories. First, we defined a sequence $V_1 \subseteq \ldots \subseteq V_5$ of subsets of V, such that $|V_1| = 900$, $|V_2| = 925$, ..., $|V_5| = 1000$ (thus, $V_5 = V$). These vertex sets specify subgraphs of G. The theories representing the 4-colorability of these subgraphs, based on clauses (C1)-(C5) ((C1), (C2), (C5) and (CC), respectively) form a sequence of relaxations of the theory representing the 4-colorability of the graph G. We used these sequences with $WSAT$+$Boot$ ($WSAT(cc)$+$Boot$, respectively). In Table 1, we

[3] A propositional cardinality atom is an expression of the form $l\{a_1,\ldots,a_n\}u$. Its role is to restrict the number of atoms a_i that are true to be at least l and at most u. A propositional clause in the extended language consists of atoms, cardinality atoms or their negations [1,5,3].

summarize the results by listing the average numbers of tries taken over all instances in each family and the corresponding standard deviations.

Table 1. 4-coloring problem

Family of Graphs	$WSAT(cc)$	$WSAT(cc)$+$Boot$	$WSAT$	$WSAT$+$Boot$
1000 $\times$ 3900 (avg/std dev)	106/244	19/21	91/107	19/24
1000 $\times$ 3910 (avg/std dev)	145/216	15/11	86/87	16/20
1000 $\times$ 3920 (avg/std dev)	300/329	24/38	225/274	23/30

For each algorithm, the bootstrapping significantly improves the performance (on average, by the factor of 9 for $WSAT(cc)$ and 7 for $WSAT$). We also experimented with harder instances of the problem (obtained by generating graphs with more edges). For a family of 50 graphs with 1000 vertices and 4100 edges, methods with bootstrapping found solutions in 49 cases using on average 2611 tries. Methods without bootstrapping are unable to find solutions in any of these 50 cases, even with as many as 13000 tries per instance.

For the vertex-cover problem, we randomly generated 50 graphs having 2000 vertices and 4000 edges. Then, for each graph G we set the upper bound k_G on the size of the vertex cover so that $WSAT(cc)$+$Boot$ can find a solution but requires several tries. In each case, the bound is in the range 1030-1049. For graphs of that size and the bound on the vertex-cover set at about half of the size of the vertex set the use of cardinality atoms when modeling the problem constraints is critical. CNF encodings are too large for $WSAT$ to be effective. Therefore, we do not report results on the performance of $WSAT$(+*Boot*) on these instances. For each graph G, the relaxation sequence we used with $WSAT(cc)$+$Boot$ is the sequence of theories encoding instances of the vertex-cover problem for G with the upper bound on the vertex cover decreasing (with step 1) from 1050 down to k_G. Clearly, each such theory is a relaxation of its successor.

Table 2. Vertex-cover problem

	$WSAT(cc)$	$WSAT(cc)$+$Boot$
# of tries (avg/std dev)	98/140	46/101
Success rate	82%	100%

The results for the vertex-cover problem (Table 2) also demonstrate the effectiveness of bootstrapping. Within the prespecified number of tries per instance set at 600, $WSAT(cc)$+$Boot$ finds solutions for *all* the instances while $WSAT(cc)$ only for 82% of them. In addition, the average number of tries to the first solution (over all instances for which both methods found solutions) is about 2 times smaller for the bootstrapped version.

For the Schur-number problem, we considered its instance $S(138,5)$. This instance has solutions. In fact, every instance $S(n,5)$, where $n \leq 160$, does (the larger n, the harder the problem gets, it is not known whether the instance $S(161,5)$ has solutions). We chose the instance $S(138,5)$ for our experiments as even that instance is already very hard for typical local-search solvers. We used theories representing instances $S(n,5)$, where n is even and $118 \leq n \leq 138$, as the sequence of relaxations as input to $WSAT$+$Boot$ and $WSAT(cc)$+$Boot$. Similarly, for the van der Waerden-number problem, we considered the instance $W(120,5,3)$ (which is satisfiable). The encodings of the instances $W(n,5,3)$, $110 \leq n \leq 120$, form the input to algorithms $WSAT$+$Boot$ and $WSAT(cc)$+$Boot$.

Our results for the Schur and van der Waerden problems, presented in Table 3, provide additional confirmation of the effectiveness of bootstrapping (improvement roughly by the factors between 4 and 10). They represent averages over 10 independent executions of the algorithms on the instances $S(5,138)$ and $W(120,5,3)$, and the corresponding standard deviations. Another fact to mention is that, the methods with bootstrapping have a 100% success rate in the ten runs for the $W(120,5,3)$, while the ones without bootstrapping only have 80% and 40% success rates, respectively.

Table 3. Schur and van der Waerden Problem

Problem	$WSAT(cc)$	$WSAT(cc)$+$Boot$	$WSAT$	$WSAT$+$Boot$
$S(5,138)$ (avg/std dev)	811/835	78/95	763/982	91/55
$W(120,5,3)$ (avg/std dev)	438/322	117/130	605/347	125/81

We also mention that $WSAT$+$Boot$ and $WSAT(cc)$+$Boot$ found solutions for the instances $S(160,5)$ (in 5086 and 247 tries, respectively), and $W(125,5,3)$ (in 19155 and 2359 tries, respectively), matching the best bounds known so far. Even with the 5-fold increase in the numbers of tries, $WSAT$ and $WSAT(cc)$ were unable to find solutions for these two instances.

5 Conclusions

We introduced bootstrapping, a general method to improve the performance of local-search satisfiability solvers. We demonstrated that bootstrapping is highly effective for $WSAT$ and $WSAT(cc)$, the two solvers that we tested.

In the paper, we presented results concerning just one strategy for the bootstrapping search when, after failure, the search is resumed from the beginning. In our future work, we will consider alternative strategies of backtracking to intermediate levels.

Bootstrapping requires a sequence of relaxations of input theories. In our work so far, we constructed these relaxations "by hand", based on high-level representations of input theories in the language of the logic PS+. Our experience indicates that the process can be automated and we are currently pursuing this direction.

References

1. B. Benhamou, L. Sais, , and P. Siegel. Two proof procedures for a cardinality based language in propositional calculus. In *Proceedings of STACS-94*, pages 71–82. 1994.
2. M. Cadoli and T. Mancini. Automated reformulation of specifications by safe delay of constraints. In Alan M. Frisch, editor, *Proceedings of the Second International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, pages 33–47, 2003. (In conjunction with CP-2003).
3. H.E. Dixon and M.L. Ginsberg. Inference methods for a pseudo-boolean satisfiability solver. In *Proceedings of AAAI-2002*, pages 635–640, 2002.
4. M.R. Dransfield, V.M. Marek, and M. Truszczyński. Satisfiability and computing van der waerden numbers. In *Proceedings of SAT-2003*. LNCS 2919, Springer Verlag, 2003.
5. D. East and M. Truszczyński. Propositional satisfiability in answer-set programming. In *Proceedings of Joint German/Austrian Conference on Artificial Intelligence, KI'2001*, pages 138–153. LNAI 2174, Springer Verlag, 2001. (Full version at `http://xxx.lanl.gov/abs/cs.LO/0211033`).
6. M.L. Ginsberg and A.J. Parkes. Satisfiability algorithms and finite quantification. In *Proceedings of KR-2000*, pages 690–701. Morgan Kaufmann Publishers, 2000.
7. R.L. Graham, B.L. Rothschild, and J.H. Spencer. *Ramsey Theory*. Wiley, 1980.
8. H.H. Hoos and T. Stützle. Local search algorithms for sat: An empirical evaluation. In I. Gent, H. van Maaren, and T. Walsh, editors, *SAT2000: Highlights of Satisfiability Research in the Year 2000*, volume 62 of *Frontiers in Artificial Intelligence and Applications*, pages 43–88. IOS Press, Amsterdam, 2000.
9. L. Liu and M. Truszczyński. Local-search techniques in propositional logic extended with cardinality atoms. In *Proceedings of CP-2003*. LNCS 2833, Springer Verlag, 2003.
10. B. Selman, H.A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of AAAI-94*, Seattle, USA, 1994.