An *Smodels* System with Limited Lookahead Computation

Gayathri Namasivayam and Mirosław Truszczyński

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

Abstract. We describe an answer-set programming solver *smodels*$^-$, derived from *smodels* by eliminating some lookahead computations. We show that for some classes of programs *smodels*$^-$ outperforms *smodels* and demonstrate the computational potential of our approach.

1 Introduction

In this paper we describe an answer-set programming solver *smodels*$^-$. It is a modification of the *smodels* solver [8]. The main difference is that *smodels*$^-$ attempts to identify and eliminate unnecessary lookaheads.

A common step in many answer-set programming and satisfiability solvers consists of expanding partial truth assignments. That is, given a partial truth assignment P, the solver applies some efficient inference rules to derive additional truth assignments that are forced by P. In the case of satisfiability solvers this process is called *unit-propagation* or *boolean constraint propagation* (cf. [3] for a recent overview).

These rules generalize to logic programs. Together with some other inference rules, specific to logic programming (cf. [5] for examples), they imply an expansion method for logic programs that can be viewed as a computation of the Kripke-Kleene fixpoint [4]. This method can be implemented to run in linear time. A stronger expansion technique, giving in general more inferences, is obtained when we replace the *Kripke-Kleene* fixpoint computation with a computation of the *well-founded* fixpoint [9]. The greater inference power comes at a cost. The well-founded fixpoint computation can be implemented to run in polynomial time, but no linear-time implementation is known.

Any polynomial-time expansion technique can be strengthened to another polynomial-time expansion method by applying the *lookahead*. Given a partial assignment, we assume a truth value for an unassigned atom and apply the expansion method at hand to the resulting partial assignment. If a contradiction is derived, the opposite truth value can be inferred and the expansion procedure is invoked again. The *full lookahead* consists of applying this technique to every unassigned atom and to both ways atoms can be assigned truth values until no more truth values for atoms (no more literals) can be derived. The full lookahead and the well-founded fixpoint computation form the basis for the expansion method used by *smodels*.

When expansion terminates, a typical answer-set programming solver selects an atom for branching. This step has a major effect on the performance of the overall search. *Smodels* uses the results obtained by the lookahead computation to decide which atom to choose.

Thus, in at least two important ways the performance of *smodels* depends on the full lookahead: it strengthens the expansion method, and it provides a good method to select atoms for branching. However, the full lookahead is costly. Our goal in this paper is to propose and implement a method that aims to improve the performance of *smodels* by limiting its use of lookahead so that few essential lookaheads (possibly even none at all) are missed. We call *smodels*$^-$ the resulting modification of *smodels*.

2 Algorithm for Identification of Propagating Literals

An unassigned literal is *propagating* (with respect to a program, a partial assignment and a particular expansion method) if assuming it is true and running the expansion method infers another literal that has been unassigned so far.

We will now present a method to identify propagating literals. The programs we consider consist of rules of the following types (in particular, such programs are output by *lparse*[2]; a and a_i **stands** for atoms, l_i stands for literals, m, w and w_i are non-negative integers):

Basic rule: a :- $l_1, \ldots, l_k$
Choice rule: $\{a_1, \ldots, a_n\}$:- $l_1, \ldots, l_k$
Cardinality rule: a :- $m\{l_1, \ldots, l_k\}$
Weight rule: a :- $w\{l_1 = w_1, \ldots, l_k = w_k\}$.

A rule is *active* with respect to the current partial assignment if its body is neither implied to be **true** nor **false** by the assignment. Our algorithm identifies a literal l as propagating, if there is an active rule r which, if we assume l to be **true**, allows us to make an inference. To this end, we consider all active rules in which l or its dual, $\bar{l}$, appear. Let us assume that r is a rule currently under consideration. Let $hd(r)$ and $bd(r)$ denote the set of literals that appear in the head and body of the rule r, respectively. There are four main cases (the cases not listed below either cannot occur or do not allow additional derivations based just on r).

Case 1. The literal l appears in the $bd(r)$.

Case 1a: The rule r is basic. If l is the only unassigned literal in the $bd(r)$ and the $hd(r)$ is unassigned then, we identify l as propagating (assuming l is **true** allows us to derive the $hd(r)$).

If l and exactly one other literal, say l', in the $bd(r)$ are unassigned and, in addition, the $hd(r)$ is assigned **false**, then assuming l allows us to infer that l' must be **false**. Thus, we identify l as propagating.

Case 1b: The rule r is a weight rule (the case of the cardinality rule is a special case). If the $hd(r)$ is unassigned and the sum of the weights of literals in the weight atom of r that are assigned **true** in the current partial assignment plus the weight of the literal l exceeds the lower bound w, then we identify l as propagating (assuming l is **true** allows us to derive the $hd(r)$).

If the $hd(r)$ is assigned **false** and the sum of the weights of literals in the weight atom of r that are assigned **true** in the current partial assignment plus the weight of l plus the largest weight of an unassigned literal other than l (say this literal is l') exceeds the lower bound, then we identify l as propagating (assuming l is **true** would allow us to infer that l' is **false**).

Case 2. The literal $\bar{l}$ appears in the $bd(r)$. To handle this case, for each atom a we maintain a counter, $ctr(a)$, for the number of active rules with this atom in the head.

Case 2a: The rule r is a basic or choice rule. If the head of r is an unassigned atom, say h, with $ctr(h) = 1$, or contains an unassigned atom h with $ctr(h) = 1$, we identify l as propagating (assuming l to be **true** blocks r and allows us to establish that h is **false**).

If the $hd(r)$ is an atom h such that h is assigned **true** and $ctr(h) = 2$, or if the $hd(r)$ contains an atom h assigned **true** and such that the $ctr(h) = 2$, we identify l as propagating. Assuming l is **true** blocks r and leaves only one active rule, say r', to justify h. This allows us to infer that the body of r' is **true** and may allow new inferences of literals. We do not check whether knowing that the body of r' is **true** allows new inferences. Thus, we may identify l as propagating even though it actually is not. This may lead to some unnecessary lookaheads that *smodels*$^-$ will occasionally perform. We are currently developing an implementation which eliminates this possibility here (and in some other similar cases below).

Case 2b: The rule r is a weight rule (the cardinality rule is a special case). If the $hd(r)$, say h is unassigned, if $ctr(h) = 1$ and if setting l to **true** makes the weight atom in the $bd(r)$ **false**, then we identify l as propagating (setting it to **true** blocks r and allows us to infer that h is **false**).

If h is assigned **true** and the $ctr(h) = 1$, then we identify l as propagating (assuming l to be **true** may force other literals in the weight atom to be **true**; in this case we again do not actually guarantee that l will lead to new inferences).

If h is assigned **true**, $ctr(h) = 2$, and assuming l to be true forces the weight atom in the body of r to be false (blocks r), then there is only one other rule, say r' that could be used to justify h. The body r' must be **true** and it may lead to new inferences of literals. Again, we identify l as propagating, even though there is actually no guarantee that it is.

Case 3. The literal l appears in the $hd(r)$. It follows that l is an atom. If the $ctr(l) = 1$, we identify l as propagating (assuming l **true** forces the $bd(r)$ to be true and will lead to new inferences in the case of basic and choice rules, and may lead to new inferences in the case of cardinality and weight rules).

Case 4. The literal $\bar{l}$ is the $hd(r)$ (that is, $l = not\ h$ for some atom h).

Case 4a: The rule r is a basic rule. If the $bd(r)$ contains a single unassigned literal, say t, we identify l as propagating (assuming l **true** forces t to be **false**).

Case 4b: The rule r is a weight rule (the cardinality rule is a special case). If the sum of the weights of all literals assigned true in the weight atom of r plus the largest weight of an unassigned literal (say t) exceeds the lower bound, we identify l as propagating (assuming l **true** allows us to infer t to be **false**).

3 Implementation and Usage

We implemented *smodels*$^-$ by modifying the source code of *smodels*. In *smodels*$^-$, we take each atom from the queue of atoms that *smodels* performs lookaheads on and check, using the approach described above, if any of the two literals of this atom is propagating. If so, then *smodels*$^-$ performs lookahead on this literal. Otherwise, *smodels*$^-$ skips this lookahead.

Our program[1] is used in exactly the same way as *smodels*. It requires that input programs consist of rules described above. *Lparse* can be used to produce programs in the appropriate input.

4 Experimental Results and Discussion

For tight logic programs our method to limit lookaheads does not miss any essential lookaheads. Therefore, on tight programs *smodels*$^-$ and *smodels* traverse the same search space. As concerns the time performance, *smodels*$^-$ performs (in general) fewer lookaheads. However, it incurs an overhead related to identifying atoms that do not propagate. For programs with many fewer lookaheads, the savings outweigh the costs and we expect *smodels*$^-$ to perform better than *smodels*. On programs where few lookaheads are saved, one might expect that *smodels*$^-$ would perform worse but not drastically worse, as our algorithm to eliminate lookaheads works in polynomial time (in the worst case).

Our experiments confirm this expected behavior. We considered three classes of programs: (1) programs obtained by encoding as logic programs instances used in the SAT 2006 competition of pseudo-boolean solvers [7] (154 instances); (2) randomly generated tight logic programs (39 instances); and (3) logic programs encoding instances of the weighted spanning-tree problem (27 instances) [1]. The results are presented in Figure 1. The graphs show how many times *smodels*$^-$ is slower or faster (whichever is the case) than *smodels*, based on the running times. The instances are arranged according to ascending running time of *smodels*$^-$. The dotted lines separate the instances into those for which *smodels*$^-$ is slower than, runs in the same time as, and is faster than *smodels*.

For the first category of programs, *smodels*$^-$ clearly outperforms *smodels*. It is due to two factors: *smodels*$^-$ performs on average 71% fewer lookahead computations; and the time needed by lookahead in *smodels* to discover that no inferences can be made is non-negligible.

For the next two classes of programs, the benefits of limiting lookahead are less obvious. Overall there is no significant difference in time in favor of any of the methods. For programs in the second group, calling lookahead for an atom and discovering no new inferences can be made does not take much computation due

[1] *Smodels*$^-$ can be obtained from http://www.cs.engr.uky.edu/ai/.

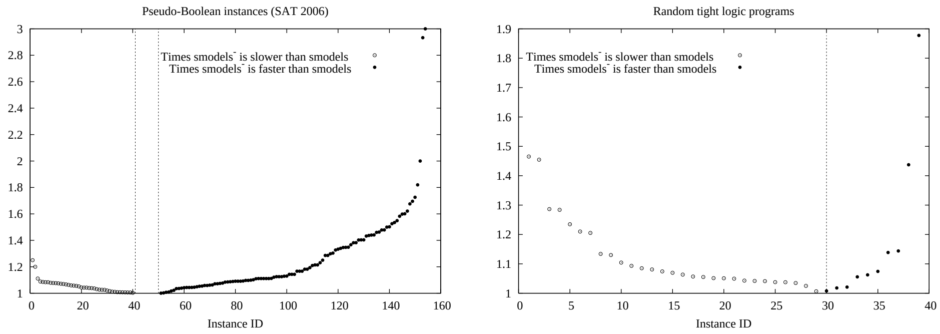

Fig. 1. PB Instances and random tight logic programs

to a simple form of rules in programs (no weight or cardinality atoms, each rule consisting of three literals). Thus, even though fewer lookaheads were made by *smodels*$^-$ (16% fewer on average on programs in the second group), the savings often did not always compensate the overhead. For the programs in the third group, most atoms appear in 2-literal clauses and no such atom will be excluded from lookahead by our technique. Thus, both *smodels*$^-$ and *smodels* perform an identical number of lookahead computations and their time performance is within a small percentage from each other (in all but one case, within 1.5% from each other; 4% in the remaining case). As the times are essentially identical, we do not provide the graph.

For non-tight programs, our method may eliminate lookaheads yielding new inferences through the well-founded fixpoint computation (whether the computation yields new inferences cannot be determined by inspecting rules individually). Therefore, the expansion method of *smodels*$^-$ is in general weaker than that of *smodels*. Moreover, due to missing essential lookaheads, the search heuristics of *smodels*$^-$ may miss atoms that will be used for branching by *smodels*.

Thus, for non-tight programs the benefits of using *smodels*$^-$ may diminish or disappear entirely. However, in our experiments it was not so. We tested two classes of programs: the non-tight programs encoding the traveling salesperson problem (TSP) [6] (47 instances); and random logic programs [1] (281 instances, all turned out to be non-tight, even though it is not *a priori* guaranteed). The results are presented in Figure 2. For TSP problem, *smodels*$^-$ still shows an overall better performance than *smodels* (although the improvement does not exceed 7%). In the case of random programs, no program seems to have any discernible edge.

5 Conclusion

We presented an answer-set programming solver *smodels*$^-$, obtained by limiting lookaheads in *smodels*. The experiments show that on tight programs *smodels*$^-$ often outperforms *smodels*, especially on programs using many weight atoms. It

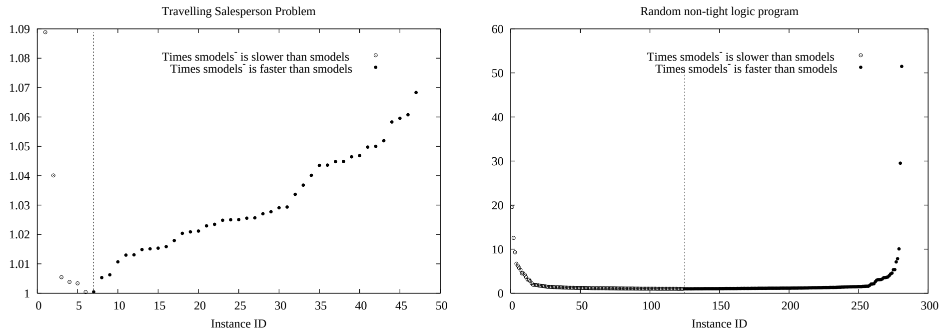

Fig. 2. Traveling Salesperson Problem, Random Logic Program (non-tight)

is never much worse than *smodels* as it always searches through the same search space and extra computation it incurs runs in polynomial time. For non-tight programs, our experiments showed that *smodels*$^-$ performs comparably to *smodels* despite the fact it may miss essential lookaheads. However, we expect that there are classes of non-tight programs on which *smodels* would prove superior.

In our ongoing work we study additional techniques to limit lookahead and more efficient ways to implement them to decrease the overhead.

Acknowledgments

The authors thank Ilkka Niemelä for suggesting this research direction and Lengning Liu for help in preparing the graphs. The authors acknowledge the support of NSF grant IIS-0325063 and KSEF grant 1036-RDE-008.

References

1. Asparagus, http://www.asparagus.cs.uni-potsdam.de/.
2. Lparse, http://www.tcs.hut.fi/Software/smodels/.
3. H.E. Dixon, M.L. Ginsberg, and A.J. Parkes, *Generalizing Boolean Satisfiability I: Background and Survey of Existing Work*, Journal of Artificial Intelligence Research **21** (2004), 193–243.
4. M. C. Fitting, *A Kripke-Kleene semantics for logic programs*, Journal of Logic Programming **2** (1985), no. 4, 295–312.
5. M. Gebser and T. Schaub, *Tableau calculi for answer set programming*, Proceedings of ICLP 2006 (S. Etalle and M. Truszczyński, eds.), LNCS, vol. 4079, Springer, 2006, pp. 11–25.
6. L. Liu and M. Truszczyński, http://www..cs.uky.edu/ai/pbmodels.
7. V. Manquinho and O. Roussel, *Pseudo boolean evaluation 2005*, 2006, http://www.cril.univ-artois.fr/PB06/.
8. P. Simons, I. Niemelä, and T. Soininen, *Extending and implementing the stable model semantics*, Artificial Intelligence **138** (2002), 181–234.
9. A. Van Gelder, K.A. Ross, and J.S. Schlipf, *The well-founded semantics for general logic programs.*, Journal of the ACM **38** (1991), no. 3, 620–650.