A Dynamic-Programming Based ASP-Solver*

Michael Morak, Reinhard Pichler, Stefan Rümmele, and Stefan Woltran

Institut für Informationssysteme E184/2, Technische Universität Wien
Favoritenstr. 9–11, 1040 Wien, Austria

Abstract. We present a novel system for propositional Answer-Set Programming (ASP). This system, called dynASP, is based on dynamic programming and thus significantly differs from standard ASP-solvers which implement techniques stemming from SAT or CSP.

1 Introduction

Answer-Set Programming (ASP, for short) [6,7] is nowadays a well acknowledged paradigm for declarative problem solving as witnessed by many successful applications in the areas of AI and KR. Evaluating ASP-programs relies on two steps, the grounding (which instantiates the variables in the program's rules) and the solving process itself which works on ground (i.e. propositional) programs. For the latter task, many different solvers (see [1] for an overview) exist nowadays, and also the system presented here falls into this category.

Solving ground programs still is an intractable problem. More precisely, decision problems for disjunctive programs (DLPs) are located on the second level of the polynomial hierarchy, but also decision problems defined over disjunction-free programs — usually called normal programs (NLPs) — are NP- or coNP-complete. The same complexity as for NLPs holds if a certain restriction on the usage of disjunction (head-cycle free programs, HCFPs) is assumed. Due to these intractability results, standard-ASP solvers use techniques stemming from SAT or CSP, where intractability has been successfully tackled in practice.

Our solver, which we present here, relies on a more theoretical approach to deal with intractable problems, namely *parameterized complexity theory* where the idea is to bound a certain parameter by a constant in order to obtain tractable fragments for the problems under consideration. One important such parameter is *treewidth*, which measures the "tree-likeness" of a graph. For instance, the problem of deciding ASP consistency (i.e. whether a disjunctive logic program has at least one answer set) has been shown tractable [3] for programs having an incidence graph of bounded treewidth. Treewidth is defined over so-called tree decompositions which in turn can be used by dynamic programming (DP) methods to solve the considered problem. One such algorithm for disjunctive ASP has been presented recently [4]. We refer to [4] also for details how concepts as incidence graphs, treewidth, tree decomposition, etc. are defined in terms of ASP programs. It is however important to note that such DP algorithms can

* Supported by the Austrian Science Fund (FWF), project P20704-N18.

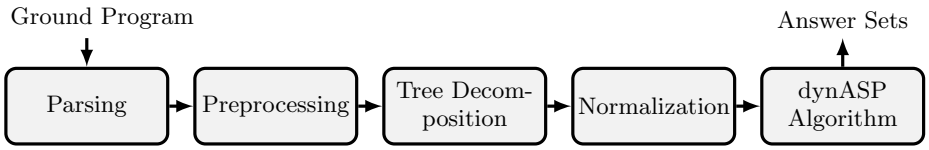

Fig. 1. Architecture of dynASP.

be applied to an arbitrary program as soon as a tree decomposition is found for that program. The running time of the DP algorithms however heavily depends on the width of the supplied tree decomposition.

To evaluate this new DP-based approach for ASP, we have implemented two such methods: one for DLPs, cf. [4]; and another one applicable to HCFPs which relies on a rather different idea[1]. Thus, we call our solver dynASP, which first finds a tree decomposition for the given program, and then uses the aforementioned DP methods to evaluate the program. This entire process underlying our approach is hidden from the user. In fact, dynASP presents itself like a standard-ASP solver.

The main aim of this paper is to announce the release of the first prototype of our ASP-solver and to present some implementation details as well as some preliminary experiments. Ongoing and future work will carefully evaluate the potential of this novel approach for solving ASP programs.

2 Architecture

Figure 1 gives an overview of the overall architecture of the system. Generally the system works in five discrete steps:

- **Parsing** DLV-style programs is done via a lex/yacc parser.
- **Preprocessing** is done after successfully reading the input. This task is twofold: Firstly, it performs equivalence-preserving transformations for the provided program (for the moment, this just takes care of some special cases like tautological and empty rules in order to circumvent problems which might arise if such rules are present in the later steps; for future versions this task can be extended to find simplifications in terms of reducing the tree-width of the program without changing its semantics), second it constructs the incidence graph of the program which is then passed to the Tree Decomposition module.
- For computing the tree decomposition, we use an algorithm based on heuristics [2]. This algorithm does not guarantee to find a tree decomposition of minimal width. But it usually finds a tree decomposition of width close to the minimum at comparatively low computational cost. An implementation of this algorithm is freely available from `http://www.dbai.tuwien.ac.at/proj/hypertree/downloads.html`.

[1] Basically, the HCFP algorithm follows the ideas underlying a DP algorithm for weight-constraints programs introduced in [8].

– **Normalization** guarantees that the difference between a node and its predecessor in the tree decomposition is at most one variable or rule. Our algorithms require tree decompositions of this particular form.
– The actual **algorithm** is run. Depending on the supplied program options, this is either the ASP algorithm for general DLPs or the algorithm for HCFPs, as described earlier.

3 System Specifics

An executable version of dynASP is available under

`http://www.dbai.tuwien.ac.at/research/project/tractability/dynasp/`

dynASP is invoked via command line and provides the following options:

```
dynasp [-b] [-t] [-s <seed>] [-f <file>] -a <alg> -o <output>
```

`-b` print benchmark information
`-t` perform only tree decomposition step
`-s seed` initialize random number generator with "seed"
`-f file` the file to read from
`-a alg` the algorithm to use, one of {`sat`, `minsat`, `asp`, `hcfasp`}
`-o output` the output-type, one of {`enum`, `count`, `yesno`}

The benchmark information consists of timing information being printed to the screen containing information about CPU time used for the tree decomposition, normalization, algorithm and evaluation steps and of course the overall time.

Depending on whether a file option is given, input is either read from a file or from standard input. Depending on the algorithm, the input has to be in a certain format. For the DLP algorithm and its HCF pendant the file has to come in the core language of dlv [5], i.e. restricted to ground disjunctive programs. The algorithm and output options specify which algorithm to use and what the output should be (i.e. enumeration, counting or consistency checking).

dynASP is written in C++ using lex/yacc for parsing the input. In its current version dynASP has nearly 6700 lines of code written in C++ (including the tree decomposition functionality). The core algorithm for DLPs has around 600 lines of C++ code, the algorithm for head-cycle-free programs has about 200 more. Certain auxiliary functionality used by both algorithms is implemented in another 200 lines.

dynASP uses an extensible class structure, allowing for code re-use and easy implementation of various algorithms based on tree decompositions. Both the DLP algorithm and the one for head-cycle-free programs are implemented using this class structure, with the two implementations sharing much of the code used for consistency checking and answer-set enumeration. To illustrate the ability to implement other types of algorithms based on tree decompositions, DP algorithms for SAT and MINSAT have also been implemented using the same framework, however reading files in DIMACS CNF format.

4 Discussion

First experiments with our system on logic programs of low treewidth have resulted in competitive performance compared with state-of-the-art ASP solvers. The performance of our system is particularly favorable in situations where only one pass of the tree decomposition is required (i.e., to check consistency or for counting the answer sets). The evaluation of HCF programs with the dedicated HCFP algorithm tends to display a better performance than the algorithm for general programs. However, in its current implementation, our HCFP algorithm is quite sensitive to the particular form of the tree decomposition. In contrast, the algorithm for general DLPs is rather robust in this respect. Its performance is mainly determined by the treewidth. Up to treewidth 5 – 7, this performance is comparable to that of the DLV system.

Work on our system is ongoing. Major goals for the near future are further performance improvements, e.g., by introducing better "data structures" or by simplifying the programs to reduce the treewidth (without changing the semantics of the programs). Eliminating the sensitivity of the HCFP algorithm to the particular form of the tree decomposition also falls into this category. Moreover, we plan to extend this framework by implementing further algorithms like e.g., for programs with weight/cardinality constraints, as well as support for input in the SMODELS solver syntax.

References

1. M. Denecker, J. Vennekens, S. Bond, M. Gebser, and M. Truszczynski. The second answer set programming competition. In *Proc. LPNMR'09*, volume 5753 of *LNCS*, pages 637–654. Springer, 2009.
2. A. Dermaku, T. Ganzow, G. Gottlob, B. J. McMahan, N. Musliu, and M. Samer. Heuristic methods for hypertree decomposition. In *Proc. MICAI'08*, volume 5317 of *LNCS*, pages 1–11. Springer, 2008.
3. G. Gottlob, R. Pichler, and F. Wei. Bounded treewidth as a key to tractability of knowledge representation and reasoning. In *Proc. AAAI'06*, pages 250–256. AAAI Press, 2006.
4. M. Jakl, R. Pichler, and S. Woltran. Answer-set programming with bounded treewidth. In *Proc. IJCAI'09*, pages 816–822. AAAI Press, 2009.
5. N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The dlv system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, 2006.
6. V. W. Marek and M. Truszczyński. Stable Models and an Alternative Logic Programming Paradigm. In *The Logic Programming Paradigm – A 25-Year Perspective*, pages 375–398. Springer Verlag, 1999.
7. I. Niemelä. Logic programming with stable model semantics as a constraint programming paradigm. *Ann. Math. Artif. Intell.*, 25(3–4):241–273, 1999.
8. R. Pichler, S. Rümmele, S. Szeider, and S. Woltran. Tractable answer-set programming with weight constraints: Bounded treewidth is not enough. In *Proc. KR'10*, pages 508–517. AAAI Press, 2010.