dbai

TECHNICAL REPORT

Preprocessing of Complex Non-Ground Rules in Answer Set Programming

DBAI-TR-2011-72

Revised Version

Michael Morak   **Stefan Woltran**

DBAI TECHNICAL REPORT

2012-06-03

Institut für Informationssysteme

Abteilung Datenbanken und

Artificial Intelligence

Technische Universität Wien

Favoritenstr. 9

A-1040 Vienna, Austria

Tel: +43-1-58801-18403

Fax: +43-1-58801-18493

sekret@dbai.tuwien.ac.at

www.dbai.tuwien.ac.at

TECHNISCHE
UNIVERSITÄT
WIEN
Vienna University of Technology

Preprocessing of Complex Non-Ground Rules in Answer Set Programming

Michael Morak[1]   Stefan Woltran[2]

Abstract. In this paper we present a novel method for preprocessing complex non-ground rules in answer set programming (ASP). Using a well-known result from the area of conjunctive query evaluation, we apply hypertree decomposition to ASP rules in order to make the structure of rules more explicit to grounders. In particular, the decomposition of rules reduces the number of variables per rule, while on the other hand, additional predicates are required to link the decomposed rules together. As we show in this paper, this technique can reduce the size of the grounding significantly and thus improves the performance of ASP systems in certain cases. Using a prototype implementation and the benchmark suites of the Answer Set Programming Competition 2011, we perform extensive tests of our decomposition approach that clearly show the improvements in grounding time and size.

---

[1]Institute of Information Systems 184/2, Vienna University of Technology, Favoritenstrasse 9-11, 1040 Vienna, Austria. E-mail: morak@dbai.tuwien.ac.at

[2]Institute of Information Systems 184/2, Vienna University of Technology, Favoritenstrasse 9-11, 1040 Vienna, Austria. E-mail: woltran@dbai.tuwien.ac.at

1 Introduction

Starting from the pioneering work of Gelfond and Lifschitz [16, 17], the declarative problem solving paradigm of answer set programming (short: ASP, see e.g. [2]) has become a central formalism in artificial intelligence and knowledge representation. This is due to its simple, yet expressive modelling language, which is implemented by systems showing a steadily increasing performance. Such systems follow a two-step approach for evaluating a program: The so-called *grounder* instantiates rules by replacing the various variables with applicable constants. This yields a propositional logic program (consisting of propositional or "ground" rules) that is equivalent for the given domain. This program is then finally fed into the actual solver. In systems like *lparse/smodels* [22] or *gringo/clasp* [12] this separation is quite strict whereas *DLV* [20] followed a more integrated approach.

Although today's ASP systems have reached an impressive state of sophistication, we believe that there is still room for improvement, in particular on the level of grounding. In fact, since checking whether a non-ground rule fires is already NP-complete [9] in general (as easily shown by analogy to the conjunctive query evaluation problem, which is also NP-complete, cf. [1]), grounders have to list all possibly applicable instantiations of non-ground rules which are, by the NP-completeness of the aforementioned problem, exponentially many in the worst case. However, often the rules exhibit a particular structure which, in theory, could be used to avoid or at least reduce this blow-up. Several preprocessing and optimization techniques that work well in practice have been developed in the past, see, e.g., [10, 13, 11], but to the best of our knowledge, in the area of ASP, decomposition of rules via hypergraphs has not been implemented or systematically investigated yet.

In this paper we present such a novel preprocessing strategy. It is based on ideas of Gottlob et. al. in [19], who employed a similar mechanism to efficiently solve the boolean conjunctive query evaluation problem. In our approach, each rule is represented as a hypergraph, where each variable in the rule is represented by a vertex and each predicate in the rule is represented by a hyperedge in the hypergraph. Using a hypertree decomposition of this hypergraph representation, the rule can then be split up into an equivalent set of smaller rules, whose grounding is only exponential in the size of the nodes in the hypertree decomposition (i.e., the number of variables in each node). In cases where the size of the nodes is considered to be bound by a fixed constant, the grounding thus remains linear in the size of the non-ground program when using current generation grounders. First experiments with a prototype implementation and the benchmarks from the well-known Third ASP Competition 2011 [7] show a significant decrease both in grounding time and grounding size for certain problems.

2 Preliminaries

In this section we give a brief introduction to Answer Set Programming (ASP) as well as the to the concepts of hypergraphs and hypertree decompositions.

Logic Programs and Answer Set Semantics. We focus here only on the basic definitions; for a comprehensive and recent introduction to answer set programming, see [6].

Disjunctive logic programs are programs that consist of rules of the form

$$H_1 \vee \cdots \vee H_k \leftarrow P_1, \ldots, P_n, \neg N_1, \ldots, \neg N_m$$

where H_i, P_i and N_i are atoms. An *atom* A is a predicate with an arity and accordingly many variables or constant symbols (also called domain elements). If the arity is 0, we simply write A instead of $A()$. Variables are denoted by capital letters, constants by lower-case words. If an atom does not contain variables it is said to be *ground*. For a rule r of above form, we denote by $H(r)$ the set of head atoms of r (i.e. $H(r) = \{H_1, \ldots, H_k\}$); the positive body we denote by $B^+(r) = \{P_1, \ldots, P_n\}$ and the negative body by $B^-(r) = \{N_1, \ldots, N_n\}$. $H_1, \ldots, H_k$ are called the head atoms, and $P_1, \ldots, P_n$ (resp. $N_1, \ldots, N_m$) are called the positive body (resp. negative body) atoms of the rule. Moreover, we use $B(r) = \{P_1, \ldots, P_n, \neg N_1, \ldots, \neg N_n\}$ to denote the set of all *literals* in the body or r. The $\neg$ operator is a unary logical connective, called the *negation as failure* operator or, alternatively, *default negation*. Given a logic program Π, we denote by B_Π its *Herbrand Base*, i.e., the set of all ground atoms which can be constructed from the constants and predicates in Π.

A rule is said to be *safe* if every variable occurring in the head or negative body of the rule also occurs in the positive body of the rule. From this point onward, we only consider logic programs whose rules are safe.

Example 2.1. *An example logic program is given below:*

$$q \leftarrow E(X, Y), \neg E(X, a)$$

It has the intended meaning that the boolean predicate q is true, if there exists an edge from a vertex X to a vertex Y in a graph, but not from the vertex X to a constant vertex a. $\square$

A logic program is said to be *ground*, if it does not contain any rules with variables. A non-ground rule (i.e. one that contains variables) can be seen as an abbreviation for all possible instantiations of the variables with domain elements. In answer set programming, this step is usually explicitly performed by a grounder. Note that such a ground program can be exponential in the size of the non-ground program. In what follows, we denote by $Gr(\Pi)$ the grounding of a program Π. Moreover, we denote by $Gr(r, \Pi)$ the grounding of a single rule r with respect to the domain elements occurring in Π. Clearly, $Gr(\Pi) = \bigcup_{r \in \Pi} Gr(r, \Pi)$.

The *dependency graph* of a ground logic program Π is a graph $G = (V, E)$, where the vertices V are all the ground atoms occurring in Π, and there is an edge from atom B to atom H if both occur in the same rule $r \in \Pi$, and $H \in H(r)$ and $B \in (B^+(r) \cup B^-(r))$.

A set S of ground atoms is a *model* of a disjunctive logic program Π if S satisfies each rule in $Gr(\Pi)$. A ground rule r is satisfied by S if $H(r) \cap S \neq \emptyset$ holds, whenever $B(r)$ is satisfied by S (i.e., whenever $B^+(r) \subseteq S$ and $B^-(r) \cap S = \emptyset$). The *reduct* Π^S of a ground disjunctive logic program Π with respect to a set S of ground atoms is defined as:

$$\Pi^S = \{H(r) \leftarrow B^+(r) \mid r \in \Pi, B^-(r) \cap S = \emptyset\}$$

A set S of ground atoms is an *answer set* of a logic program Π if S is a minimal model of $(Gr(\Pi))^S$, the reduct of the grounding of Π with respect to S.

Hypergraphs and Hypertree Decompositions. Tree decompositions and treewidth, originally defined in [23], are a well known tool to tackle computationally hard problems (see, e.g., [3, 4] for an overview). Treewidth is a measure for the cyclicity of a graph and many NP-complete problems become tractable in cases where the treewidth is bounded. However, many problems are even better represented by hypergraphs. In [18] the concepts of hypertree decompositions and hypertree width were introduced that extend the measurement of cyclicity to hypergraphs.

A *hypergraph* is a pair $H = (V, E)$ with a set V of vertices and a set E of hyperedges. A hyperedge $e \in E$ is itself a set of vertices, with $e \subseteq V$. A *hypergraph of a non-ground logic program rule* r is a pair $HG(r) = (V, E)$ such that V consists of all the variables occurring in r and E is a set of hyperedges, such that for each atom $A \in B(r)$ there exists exactly one hyperedge $e \in E$, which consists of all the variables occurring in A. Furthermore there exists exactly one hyperedge $e \in E$ that contains all the variables occurring in $H(r)$.

The following definition is central for our purposes:

A *(generalized) hypertree decomposition* of a hypergraph $H = (V, E)$ is a triplet $HD = \langle T, \chi, \lambda \rangle$, where $T = (N, F)$ is a (rooted) tree and χ and λ are labelling functions such that for each node $n \in N$, $\chi(n) \subseteq V$ and $\lambda(n) \subseteq E$ and the following conditions hold:

1. for every $e \in E$ there exists a node $n \in N$ such that $e \subseteq \chi(n)$,

2. for every $v \in V$ the set $\{n \in N \mid v \in \chi(n)\}$ induces a connected subtree of T,

3. for every node $n \in N$, $\chi(n) \subseteq \bigcup_{e \in \lambda(n)} e$.

A *hypertree decomposition of a logic program rule* r is therefore a hypertree decomposition of the hypergraph of r. The *width* of a hypertree decomposition is the maximum λ-set size over all its nodes. The minimum width over all possible hypertree decompositions is called the *(generalized) hypertree width*. Similarly, the *treewidth* of a hypertree decomposition is defined by the maximum χ-set size, minus one, of a hypretree decomposition of minimal width.

Unfortunately, for a given hypergraph, it is NP-hard to compute a hypertree decomposition of minimum width. However, efficient heuristics have been developed that offer good approximations (cf. [8, 5]). In practice it turns out that these approximations are often sufficient to obtain good results with decomposition-based algorithms (i.e., algorithms that take the problem and its hypertree decomposition as input).

3 Preprocessing of Non-ground Rules

In this section we describe our main contribution, a novel method for preprocessing complex logic program rules in order to decrease the size of the grounding.

Current grounders for answer set programming do not consider the structure of a rule and thus, when grounding, the number of ground rules produced can in the worst case be exponential

in the number of variables occurring in the rule. However, given a hypertree decomposition of such a rule, the exponentiality of the grounding can be restricted to the maximum χ-set size of the decomposition.

In order to describe our algorithm, we introduce the following notational aids: For a node n in a hypertree decomposition, we represent by $\text{parent}(n)$ and $\text{desc}(n)$ the parent node of n and the set of descendants (or child nodes) of n respectively. For a set (or sequence) B of literals and a set X of variables, we denote with $B \cap \mathbf{X}$ (with some abuse of notation) the literals in B that have at least one of the variables in $\mathbf{X}$ occurring in them. E.g., if $B(r) = E(X_1, X_2), E(X_2, X_3), \neg E(X_3, X_4, c)$, then the intersection $B(r) \cap \{X_1, X_4\} = E(X_1, X_2), \neg E(X_3, X_4, c)$.

Given these shorthands, the rewriting of logic program rules according to our method works by running the following algorithm Preprocess:

1. We compute a (generalized) hypertree decomposition $HD(r) = HD(HG(r)) = \langle T = (N, F), \chi, \lambda \rangle$ of a given logic program rule r, trying to minimize the maximal χ-set size. W.l.o.g. we assume that the edge representing $H(r)$ occurs only in the root node of T.

2. We do a bottom-up traversal of the hypertree decomposition of r. For each node $n \in N$ (except the root) in the decomposition, let $\mathbf{Y_n} = \chi(n) \cap \chi(\text{parent}(n))$ and T_n be a fresh predicate to store the current result. At each node $n \in N$ we generate a rule r_n of the form:

$$
\begin{aligned}
T_n(\mathbf{Y_n}) \quad \leftarrow \quad & (B(r) \cap \chi(n)) \\
& \cup \{\Sigma_X(X) \mid X \in B^-(r) \cap \chi(n)\} \\
& \cup \{T_m(\mathbf{Y_m}) \mid m \in \text{desc}(n)\}
\end{aligned}
$$

The additional temporary predicates $\Sigma_X(X)$ are necessary to guarantee safety of the generated rule. To this end, for each variable X occurring in $B^-(r) \cap \chi(n)$, we generate a rule

$$
\Sigma_X(X) \leftarrow b
$$

where $b \in B^+(r)$ with X as one if its arguments[1].

For the root node n, we generate a rule similar to r_n but replace $T_n(\mathbf{Y_n})$ by $H(r)$ and we furthermore add all ground atoms of $B(r)$ to this generated rule (since those atoms are not represented in the tree decomposition). We refer to this generated rule as the *head rule*. Generated rules stemming from a leaf node $n \in N$ are referred to as *leaf rules*. Atoms of the from $T_n(\mathbf{Y})$ and $\Sigma_X(X)$ are subsequently called temporary atoms.

Definition 3.1. *Given a rule r we denote by r^* the set of rules obtained by running* Preprocess *on r. Moreover, for a logic program Π and $r \in \Pi$, we define $\Pi_{r^*} = (\Pi \setminus \{r\}) \cup r^*$.*

The intuition underlying the Preprocess algorithm is the following: Grounders have to compute all the groundings for every rule in a given logic program. When these rules involve multiple joins, this can be inefficient, because the grounder has to compute all possible tuples satisfying the

---

[1] We select here such a b from $B^+(r)$ with minimal arity. Note that such a predicate exists since r is safe.

first join, and then, for *each* of those, compute all possible tuples satisfying the next join, and so forth.

However, the grounder actually only needs to store the values that are involved in the next join, and perform the join operation on them, instead of the complete set of tuples. The Preprocess algorithm makes this explicit: The hypertree decomposition takes care of splitting the rules into multiple parts (i.e., the nodes in the decomposition). By construction of the decomposition, the join operations performed inside a node cannot be split up any further, thus, for each of the nodes we generate a rule performing these joins. However, in the temporary head predicate we then only store the variables that are actually involved in a join in the next node, thereby allowing the grounder to ignore the other variables for any subsequent joins.

The following brief example shows this behaviour:

Example 3.2. *Given the rule*

$$r = H(A, D) \leftarrow E(A, B), E(B, C), \neg E(C, D), E(D, A)$$

we compute a (simple) decomposition $HD(r)$, for instance the following:
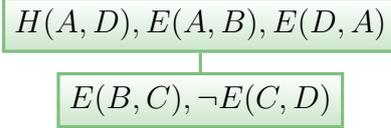
$$\boxed{H(A, D), E(A, B), E(D, A)}$$
$$\boxed{E(B, C), \neg E(C, D)}$$

This decomposition then yields the following set of rules r^*, when applying the steps discussed above:

$$\Sigma_D(D) \leftarrow E(D, A)$$
$$T_1(B, D) \leftarrow E(B, C), \neg E(C, D), \Sigma_D(D)$$
$$H(A, D) \leftarrow E(A, B), E(D, A), T_1(B, D)$$

The resulting set of rules is equivalent to the rule r in the sense of Theorem 3.3 below, however the number of possible ground rules is now only in $O(2^{max_{n \in N}|\chi(n)|})$ instead of $O(2^{|\mathbf{X}|})$, with $\mathbf{X}$ the variables in r. □

Once we have preprocessed a rule (or, every rule in a logic program), it is easy to recreate the answer sets of the original program, as the following theorem states:

Theorem 3.3. *Let Π be a logic program. Then for every answer set A of Π there exists exactly one answer set $A_{r^*} \supseteq A$ of Π_{r^*} and for every answer set A_{r^*} of Π_{r^*} there exists exactly one answer set $A \subseteq A_{r^*}$ of Π, such that in both cases it holds that $B_\Pi \cap A_{r^*} = A$.*

In order to proof this theorem, we first need to establish the following two technical lemmas:

Lemma 3.4. *Let Π be a logic program, $S \subseteq B_\Pi$ and $r \in \Pi$. Then, there exists a uniquely determined set $T \subseteq B_{r^*} \setminus B_\Pi$ of ground temporary atoms from r^*, such that the following conditions hold:*

(a) For every $t \in T$ there exists a rule $s \in Gr(r^*, \Pi_{r^*})$ such that $H(s) = \{t\}$ and $B(s)$ is satisfied by $S \cup T$.

(b) For every rule $s \in Gr(r^*, \Pi_{r^*})$ it holds that if $B(s)$ is satisfied by $S \cup T$ and its head atom t is a temporary atom (i.e. s is instantiated from a rule in r^* different from the head rule), then $t \in T$.

Proof. Let V be the set of variables occurring in r. For every ground instance of r we define a function $g : V \mapsto \Delta$ that maps the variables in r to the appropriate constant of Δ (i.e., the ground instance can be obtained by applying g to r).

We set $T_g = \emptyset$ and apply the following step exhaustively (i.e., until a fixpoint is reached):

- Add to T_g all temporary atoms from r^* grounded by g, where $S \cup T_g$ satisfies the corresponding rule body.

Note that by construction every rule with the exception of the head rule in r^* is identified by its unique temporary head atom, therefore, for a specific grounding g there can only be exactly one corresponding rule body per (grounded) temporary head atom.

We now set

$$T = \bigcup_g T_g$$

for all possible groundings g. T indeed satisfies conditions (a) and (b): From the construction of T it is clear that a temporary ground atom is only added to T if a corresponding rule body is satisfied. Therefore condition (a) is clearly satisfied. Also, whenever a rule body is satisfied, by construction the corresponding head atom (if temporary) is added to T. This clearly satisfies condition (b).

It remains to show that T is indeed the only set that satisfies conditions (a) and (b). The atoms contained in T must occur in any set that satisfies conditions (a) and (b), as the absence of just one atom from T would lead to a violation of at least one of those conditions: By construction of r^*, temporary atoms never occur negated and we could always run the fixpoint operation above, which would add them. We therefore only have to show that there exists no set of ground temporary atoms T' disjoint from T, such that $T \cup T'$ still satisfies conditions (a) and (b).

Suppose to the contrary that there exists such a set T'. Then by condition (a), for every $t \in T'$ there is at least one (ground) rule r with t as its head and its body satisfied by $S \cup T \cup T'$. However if the body of r consisted only of non-temporary atoms, or atoms from T, then the fixpoint construction of T would already have added them.

Therefore, the only possibility is that there is another ground temporary atom $t' \in T'$, causing the body of r to get satisfied (i.e., there is an edge from t' to t in the dependency graph of r). The same holds for t' itself, thus the only way to construct T' is by having one or more sequences of atoms $t_1, \ldots, t_n$, such that t_1 is needed to satisfy a rule with head t_2 and so forth, and finally t_n is needed to satisfy a rule with head t_1.

Note that this is the case if and only if there is a directed circle $t_1 \to t_2 \to \cdots \to t_n \to t_1$ in the dependency graph of the grounding of r^*. However, when running Preprocess, all leaf rules have only non-temporary atoms in their bodies. As the hypertree decomposition is a tree, any circle in

the dependency graph of the grounding of r^* must have an edge from a head atom of r to such a non-temporary body atom, making a circle consisting only of temporary atoms impossible, which yields a contradiction. Therefore no set T' can exist and T is indeed unique. □

Lemma 3.5. *Let Π be a logic program, $r \in \Pi$ a rule and $S \subseteq B_\Pi$ a set of atoms. Then S is a model of Π if and only if $S \cup T$ is a model of Π_{r^*}, with T being the set defined in Lemma 3.4 with respect to the set S.*

Proof. First observe that for any rule $s \in \Pi \setminus \{r\}$, $Gr(s, \Pi) = Gr(s, \Pi_{r^*})$ holds, since Π and Π_{r^*} have the same domain. It thus is sufficient to show that S satisfies r if and only if $S \cup T$ satisfies r^*. For every ground instance of r, let g be the function mapping the variables in r to appropriate domain elements. We show that S satisfies $g(r)$ if and only if S satisfies $g(r^*)$.

($\Leftarrow$): Assume that all the rules in $g(r^*)$ are satisfied by $S \cup T$. By construction, the bodies of the rules in r^* togehter consist of all the body atoms from rule r, and temporary atoms. Therefore, if all the rules in $g(r^*)$ are satisfied by $S \cup T$, either one or more of their bodies are not satisfied, which immediately implies (by the properties of T established in Lemma 3.4) that the body of $g(r)$ is not satisfied either; or all the rule-heads of $g(r^*)$ occur in $S \cup T$, immediately implying that the head of $g(r)$ occurs in S.

($\Rightarrow$): Assume that $g(r)$ is satisfied by S. The properties of T guarantee that in any case, every rule with a temporary head atom is satisfied. Therefore, we only have to show that also the head rule or $g(r)$ is satisfied. In case that $H(g(r)) \subseteq S$ (i.e., S satisfies the head of $g(r)$), this is trivially the case.

In the other case, S does not satisfy the body $B(g(r))$ of r. In this case, at least one rule body in $g(r^*)$ must be unsatisfied as well. Either this is the rule body of the head rule of $g(r^*)$, or it is the body of one of the other rules in $g(r^*)$ (say s'). In the former case, the head rule is trivially satisfied. In the latter case, by the properties of T, we know that the temporary head atom $(t = H(s')) \notin T$. Therefore, all rule bodies in $g(r^*)$ that contain t are also unsatisfied. By a simple inductive argument, it can now be shown that then at least one temporary atom in the body of the head rule of $g(r^*)$ does not occur in T, as the body of the head rule can only be satisfied, if all the temporary atoms of $g(r^*)$ are contained in T. This completes the proof.

□

We can now prove Theorem 3.3:

Proof. We first show that an answer set for Π_{r^*} can be constructed from an answer set for Π: Given an answer set S of Π, we obtain the desired answer set S^* of Π_{r^*} by setting $S^* = S \cup T$, with T being the set defined in Lemma 3.4 with respect to S. We need to show that S^* is indeed an answer set of Π_{r^*}. From Lemma 3.5, we already know that S^* is a model of Π_{r^*} and thus we need to show that S^* is a minimal model of $(Gr(\Pi_{r^*}))^{S^*}$. Observe that $(Gr(\Pi_{r^*}))^{S^*} = (Gr(\Pi_{r^*}))^S$ since temporary atoms occur only positively in Π_{r^*}. Moreover, recall that $Gr(\Pi_{r^*} \setminus r^*) = Gr(\Pi \setminus \{r\})$. Towards a contradiction, let $U \subset S^*$ be a model of $(Gr(\Pi_{r^*}))^{S^*}$. From Lemma 3.4, we can conclude that $U \cap B_\Pi \subset S^* \cap B_\Pi = S$ has to hold (otherwise U only misses some temporary atoms compared to S). By the observation that $Gr(\Pi_{r^*} \setminus r^*) = Gr(\Pi \setminus \{r\})$, we get that $U \cap B_\Pi$ cannot satisfy $\{r\}^S$ (since S is an answer set of Π). Hence there is a ground instance of r such that

8

$B^+(r)$ is satisfied by U, $B^-(r)$ is satisfied by S and $H(r) \cap U = \emptyset$. By the construction of r^* it can be seen that then U cannot satisfy the correspondingly grounded head rule of r^*; a contradiction to our assumption that U is a model of $(Gr(\Pi_{r^*}))^{S^*}$.

It remains to show the other direction: Given an answer set S^* of Π_{r^*} we obtain as the desired answer set $S = S^* \cap B_\Pi$ of Π. By Lemma 3.5, S is a model of Π. To show that S is an answer set of P assume towards a contradiction that there exists an $U \subset S$ being a model of $Gr(\Pi)^S$. As such, by Lemma 3.5, there exists a set T of temporary atoms, such that $U \cup T$ is a model of $(Gr(\Pi_{r^*}))^{S^*}$. However, Lemma 3.4 clearly implies that the set T with respect to U must be a subset of T with respect to S. Therefore we obtain a model $(U \cup T) \subset S^*$ of $(Gr(\Pi_{r^*}))^{S^*}$; a contradiction to the assumption that S^* is an answer set of Π_{r^*}. $\qquad\square$

Note that Theorem 3.3 also shows that we can replace in a program Π step-by-step each rule r by the corresponding replacement r^* and obtain a program equivalent to Π in the sense of Theorem 3.3 where each rule has been decomposed.

This leads to a decrease in grounding size, depending on the treewidth of the rules in the program. We define the size of a rule to be the size of its hypergraph representation. Then we can state the following theorem:

Theorem 3.6. *Let Π be a logic program and $r \in \Pi$ a rule of size n. If r has bounded treewidth, then the size of $Gr(r^*, \Pi_{r^*})$ is linear in the size of the rule; and, in fact, is bounded by the function $O(2^k \cdot n)$, where k is the treewidth of r.*

Proof. For every node m in $HD(r)$, a rule with at most k variables is generated. The number of groundings for this rule is bounded by $O(2^k)$, the number of all size-k subsets of the domain of Π. As the size of a hypertree decomposition is linear in the size of the input graph, we have that there are at most $O(2^k \cdot n)$ groundings for these rules. In the worst case, for every variable X, a domain closure rule $\Sigma(X)$ is generated. For bounded treewidths, the maximum number of instantiations of such a rule is in $O(2^k)$, as the body predicate may have k variables at most. Therefore we have that the maximum number of groundings of r^*, with respect to Π_{r^*}, is bounded by $O(2^k \cdot n + 2^k \cdot n) = O(2^k \cdot n)$. $\qquad\square$

Corollary 3.7. *Let Π be a logic program. If every rule in Π has bounded treewidth, then the size of $Gr(\Pi)$ is linear in the size of Π.*

The implications of the above theorem, as we will show in Section 4, can lead to serious speedups in the time it takes current-generation grounders to ground a logic program.

4 Experimental Evaluation

In order to empirically test our projected runtime behaviour, we have implemented a prototypical rule-preprocessing system available at

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

This tools makes use of the **SHARP** framework for hypertree decomposition-based algorithms[2]. Our system handles all basic ASP rules, including inequality as well as comparisons. However, arithmetical operations are currently not implemented.

Using our prototype, we performed a series of tests on a set of benchmarks from the third ASP competition[3] (see also [7]). We selected the following four problems from the competition

- Sokoban Decision
- Stable Marriage
- Minimal Diagnosis
- Partner Units Polynomial

This particular selection is motivated by the fact that these encodings do not use any arithmetical operations, choice rules or other ASP extensions, thus our first prototype is able to process them. Currently, this prototype is only able to handle the core ASP syntax, excluding arithmetical operations.

We chose problems from the ASP competition to show that, even though the encodings have been extensively hand-tuned, by intelligently splitting rules according to our algorithms, it is still possible to obtain improved grounding results. This also signifies the usefulness of our algorithm, because employing it would eliminate the need for extensive, time-consuming and notoriously imperfect hand-tuning.

In the following plots, red dots represent the value measured for the original benchmark instance and blue triangles represent the value measured for the preprocessed benchmark instance. Only the non-ground encoding was preprocessed, afterwards it was passed to gringo [15], together with the actual problem instance from the third ASP competition website, and the output was fed into claspd[4] [14]. For each problem a sample of 50 problem instances was selected. The time for preprocessing was not recorded in our plots, as for our benchmark instances it was not measurable (i.e. always below 0.1 seconds). The time limit for both gringo and claspd was 600 seconds each. If a timeout occurred, then no point was plotted for the respective instance. The "size" of the grounded program was measured by recording the number of variables, as determined by running claspd. As claspd introduces variables not only for atoms but also for rule bodies, this gives a useful impression of the actual problem size.

Figure 1(a) shows the size of the preprocessed grounded Sokoban Decision program that was output by gringo in relation to the size of the grounding of the original. As can be seen the grounding size can be reduced dramatically. On average, the size of the ground program was reduced by 78%.

Figure 1(b) shows the number of atoms in the grounded Sokoban Decision problem. Given that our preprocessing strategy introduces a number of temporary predicates in the non-ground encoding, the number of actual atoms in the ground program increases by a linear factor, as each hypertree decomposition itself is linear in the size of the respective rule, and at each node, a

---

[2]http://www.dbai.tuwien.ac.at/research/project/sharp
[3]http://aspcomp2011.mat.unical.it
[4]In short test-runs we obtained similar results for the well-known DLV solver [20]
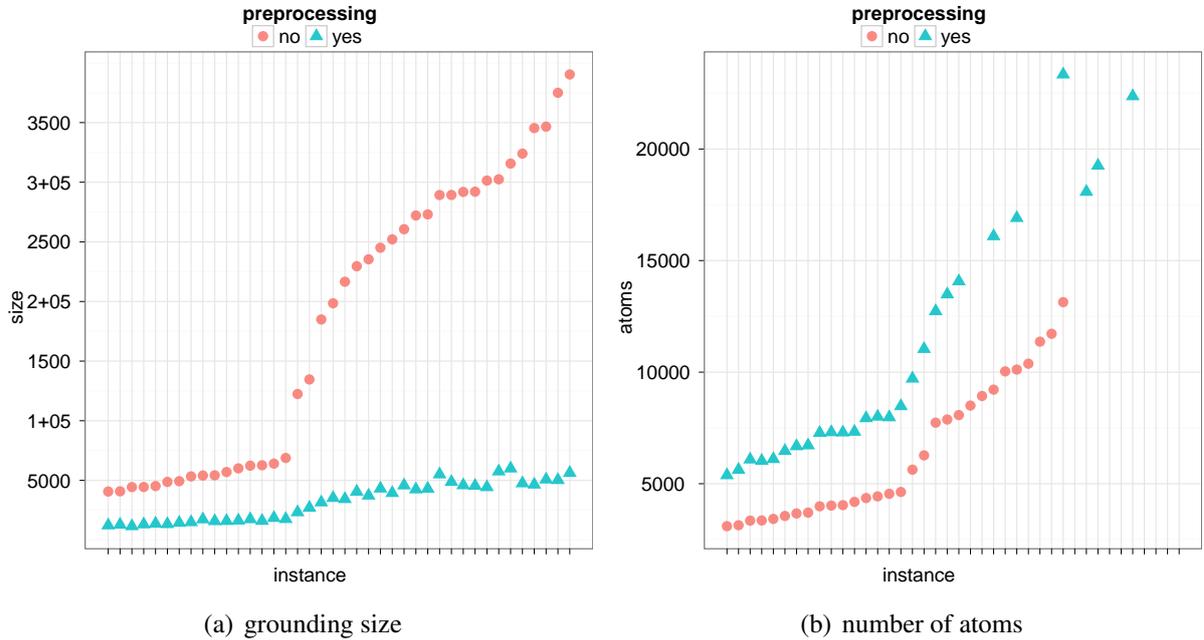
(a) grounding size

(b) number of atoms

Figure 1: Grounding size and number of ground atoms for the Sokoban Decision problem.

single new temporary predicate is introduced. However, because of the nature of our preprocessing method, the number of rules decreases, and the decrease in the number of rules corresponds well with the decrease in size of the grounding.

Figure 2 shows the time in seconds needed by claspd for solving the whole grounded problem, as well as the number of conflicts it encountered while doing so. Except for a few cases, the solving time of claspd, when combined with our preprocessing algorithm, is slightly increased, despite the much smaller size of the ground program. In rare cases however, there is a substantial slowdown of claspd. However we also noticed that for a number of instances, the smaller size of the ground program enabled claspd to solve the problem without hitting the time limit (see the topmost few instances in Figure 2(b)). The number of conflicts, shown in Figure 2(a) exhibit a similar behaviour. In most cases, an increased number of conflicts also entails an increased number of restarts of claspd.

Note that this increase in solving time could be easily eliminated if the solver (clasdp or otherwise) would be aware of the nature our the temporary atoms. The increase is mainly due to the solver making lots of unnecessary guesses about which temporary atoms should be in the answer set and which ones shouldn't. However, by Lemma 3.4, given a set of non-temporary atoms, the temporary atoms for this set can always be deterministically calculated with minimal overhead. Therefore the solver could (a) ignore all rules with temporary head atoms, as by Lemma 3.4 those are always satisfied, (b) for a guessed (partial) answer set, compute the corresponding temporary atoms as per the proof of Lemma 3.4 and (c) check, whether the head rule is satisfied.

The Sokoban Decision problem is the only problem in our benchmark selection that involves a
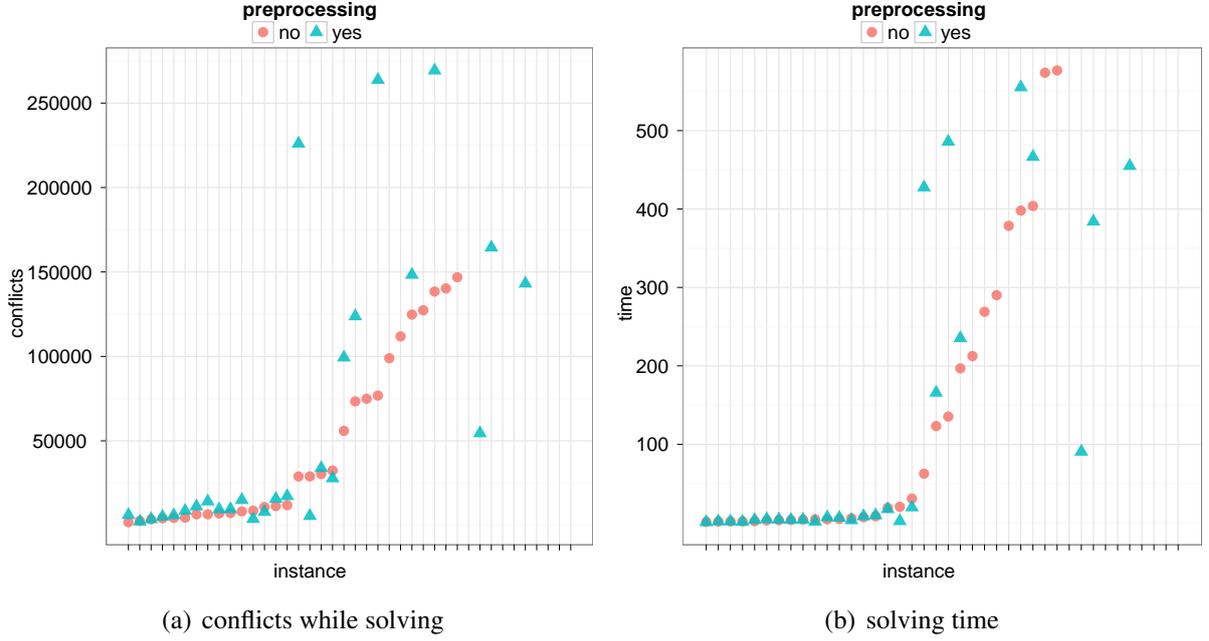
(a) conflicts while solving

(b) solving time

Figure 2: The number of conflicts encountered and the time in seconds needed by claspd for solving the grounded Sokoban Decision problem.

solving phase. The other three problems that we discuss in the following are in fact solved by the grounder itself, therefore only the grounding size and grounding time plots are relevant for these problems.

Figure 3 shows the size of the grounding of the Minimal Diagnosis and Partner Units Polynomial problems. In the latter, only a single rule is split up, which is a rule with an all-positive body (i.e. no default negation). In this case our approach works best, because no domain closure predicates (Σ) are needed. This simple split-up rule already decreases the grounding size by an average of 4%, as seen in Figure 3(b).

On the other hand, for the Minimal Diagnosis problem in Figure 3(a), all the rules that are split up are of the form

$$a(U, V) \leftarrow b(U, S), b(V, T), S \mathrel{!}= T$$

and therefore get split up into the following three rules:

$$T_1(V, S) \leftarrow b(V, T), S \mathrel{!}= T, \Sigma_S(S)$$
$$a(U, V) \leftarrow b(U, S), T_1(V, S)$$
$$\Sigma_S(S) \leftarrow b(U, S)$$

In this case, with our approach there is a chance that the actual grounding size increases, especially if many valid groundings for the fact $b(U, S)$ exist. Note that the grounding size with our preprocessing algorithm is always upper-bounded by $O(2^{\max_{n \in N} |\chi(n)|})$, as opposed to exponential

(a) Minimal Diagnosis grounding size
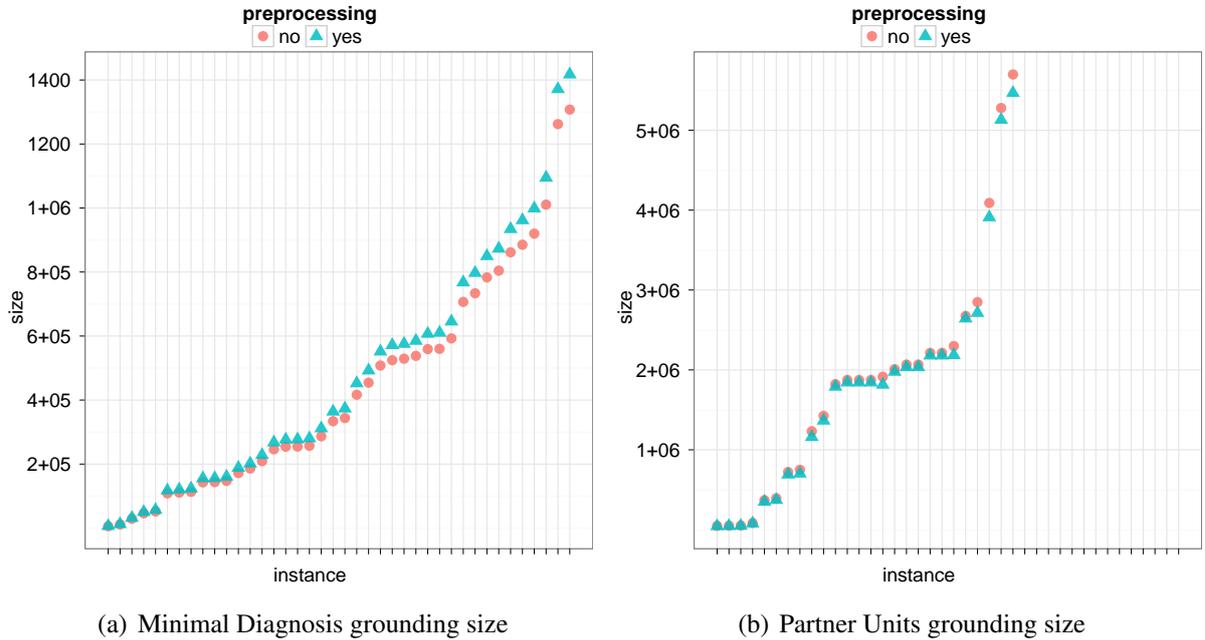
(b) Partner Units grounding size

Figure 3: The plots above show the grounding size of the Minimal Diagnosis and the Partner Units Polynomial problems.

in the number of variables of the whole rule. However these worst-case bounds are seldom exhausted. Whether a rule that gets split up as described above is actually beneficial to the overall grounding size, heavily depends on the configuration of the ground facts that are supplied to the grounder.

Note also that if our preprocessing approach would be integrated directly into the grounder, it would eliminate the need for domain closure predicates as the grounder already knows about the domain anyway. In this case it would be impossible for the grounding size to increase when employing our preprocessing approach and thus the only potential disadvantage could be eliminated.

Lastly, the Stable Marriage problem in Figure 4 shows the strength of our preprocessing algorithm. Here the non-ground rules contain many free variables and many predicates are joined together which forms the ideal basis for our algorithm. The non-ground rules force gringo to output almost exponentially many groundings for each rule. Figure 4(a) shows that a significant speedup in all cases can here be gained, for the worst-case instances, cutting the grounding time from over 300 seconds to about 5 seconds. Also the grounding size decreases dramatically. In Figure 4(b) it can also be seen, that for the topmost 15 instances, clasp could not even finish parsing the gringo output within the timeout limit of 600 seconds. In case of our significantly reduced grounding size, this was however easily possible.
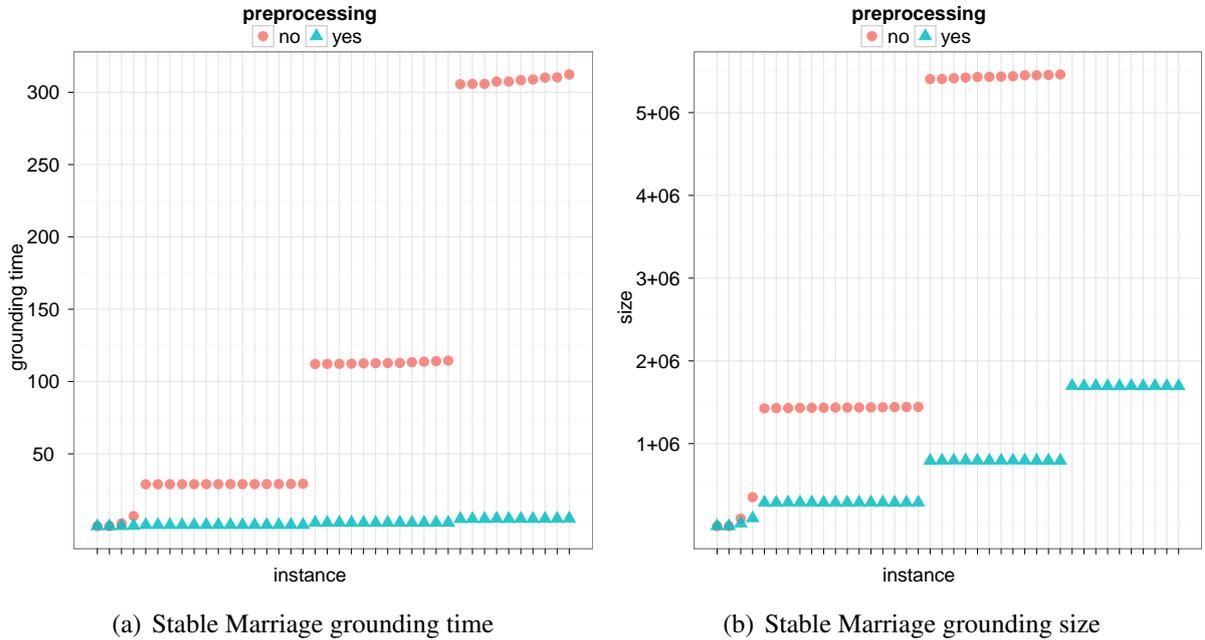
(a) Stable Marriage grounding time

(b) Stable Marriage grounding size

Figure 4: The plots above show the grounding time and grounding sizes for the Stable Marriage problem.

5 Conclusion

In this paper, we have presented a novel preprocessing strategy for non-ground rules in answer set programming. The preprocessing intelligently splits up non-ground rules into smaller ones by means of a hypertree decomposition in order to decrease the maximum number of variables per rule (and thus to reduce the size of the entire grounding). This technique follows the rule of thumb experienced ASP users will apply when encoding their problems. However, for complex rules, manual splitting becomes increasingly difficult and the readability of the encoding may suffer considerably. Also, programs may be automatically generated or specified for the purpose of presentation rather than for optimization (for instance, specifications in general game playing, see, e.g., [21]).

Benchmarks performed on problems used in the well-established answer set programming competition show significant potential of our strategy and thus warrant inclusion of such a method into existing grounders. The speedup of the grounding process is due to two factors:

Firstly, if the number of rule instantiations is reduced significantly, also the time it takes to compute and output each of these instantiations is reduced by the same amount. This effect can clearly be seen for the Stable Marriage problem in the previous section.

Secondly, by splitting up rules into smaller, equivalent ones, the number of joins between non-ground predicates is reduced. Therefore the grounder does not have to perform as many join operations as before, which also leads to a speedup of the grounding process.

Future Work. In order to use the demonstrated positive effects of our algorithm in state-of-the-art ASP grounders and solvers, there are two approaches worth investigating.

Firstly, if this preprocessing approach is directly incorporated to a grounder, the grounder may use the information about temporary predicates in order to speed up the grounding process further. Also, the domain closure predicates (Σ) are currently only a workaround, as currently our preprocessing algorithm has no information about the domain of specific variables in a non-ground rule. However, if included directly into the grounder, the domain closure predicates would become obsolete, as the grounder can immediately fill the respective variables with their now known domain, as the grounder has full information about the ground facts and domains of the various predicates and variables. This would not only lead to a speedup, but also would further decrease the size of the grounding, as the domain predicates do no longer exist, eliminating also the increase in size of the Minimal Diagnosis grounding.

Secondly, even though the size of the ground program decreases in all our benchmark cases except the Minimal Diagnosis problem, the solving time actually increases. This means that claspd is currently not aware of the tree-like structure of the split-up rules in the preprocessed and grounded instance. If the grounder could pass information about the temporary predicates to the solver, this could significantly speed up the solving process, as the temporary predicates could be automatically dismissed from the computation and the answer sets.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[3] Hans L. Bodlaender. A tourist guide through treewidth. *Acta Cybern.*, 11(1-2):1–22, 1993.

[4] Hans L. Bodlaender. Discovering treewidth. In Peter Vojtás, Mária Bieliková, Bernadette Charron-Bost, and Ondrej Sýkora, editors, *SOFSEM 2005: 31st Conference on Current Trends in Theory and Practice of Computer Science. Proceedings*, volume 3381 of *LNCS*, pages 1–16. Springer, 2005.

[5] Hans L. Bodlaender and Arie M. C. A. Koster. Treewidth computations I. Upper bounds. *Inf. Comput.*, 208(3):259–275, 2010.

[6] Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, 2011.

[7] Francesco Calimeri, Giovambattista Ianni, Francesco Ricca, Mario Alviano, Annamaria Bria, Gelsomina Catalano, Susanna Cozza, Wolfgang Faber, Onofrio Febbraro, Nicola Leone, Marco Manna, Alessandra Martello, Claudio Panetta, Simona Perri, Kristian Reale,

Maria Carmela Santoro, Marco Sirianni, Giorgio Terracina, and Pierfrancesco Veltri. The third answer set programming competition: Preliminary report of the system competition track. In James P. Delgrande and Wolfgang Faber, editors, *11th Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR 2011. Proceedings*, volume 6645 of *LNCS*, pages 388–403. Springer, 2011.

[8] Artan Dermaku, Tobias Ganzow, Georg Gottlob, Benjamin J. McMahan, Nysret Musliu, and Marko Samer. Heuristic methods for hypertree decomposition. In Alexander F. Gelbukh and Eduardo F. Morales, editors, *MICAI 2008: 7th Mexican International Conference on Artificial Intelligence, Proceedings*, volume 5317 of *LNCS*, pages 1–11. Springer, 2008.

[9] Thomas Eiter, Wolfgang Faber, Michael Fink, and Stefan Woltran. Complexity results for answer set programming with bounded predicate arities and implications. *Ann. Math. Artif. Intell.*, 51(2-4):123–165, 2007.

[10] Wolfgang Faber, Nicola Leone, Cristinel Mateis, and Gerald Pfeifer. Using database optimization techniques for nonmonotonic reasoning. In *Proc. 7th International Workshop on Deductive Databases and Logic Programming (DDLP'99)*, pages 135–139, 1999.

[11] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. Challenges in answer set solving. In M. Balduccini and T. Son, editors, *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning: Essays in Honor of Michael Gelfond*, volume 6565, pages 74–90. Springer, 2011.

[12] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. *clasp* : A conflict-driven answer set solver. In Chitta Baral, Gerhard Brewka, and John S. Schlipf, editors, *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007. Proceedings*, volume 4483 of *LNCS*, pages 260–265. Springer, 2007.

[13] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. Advanced preprocessing for answer set solving. In Malik Ghallab, Constantine D. Spyropoulos, Nikos Fakotakis, and Nikolaos M. Avouris, editors, *ECAI 2008 - 18th European Conference on Artificial Intelligence, Proceedings*, volume 178 of *Frontiers in Artificial Intelligence and Applications*, pages 15–19. IOS Press, 2008.

[14] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. The conflict-driven answer set solver clasp: Progress report. In Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors, *Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings*, volume 5753 of *LNCS*, pages 509–514. Springer, 2009.

[15] Martin Gebser, Torsten Schaub, and Sven Thiele. Gringo : A new grounder for answer set programming. In Chitta Baral, Gerhard Brewka, and John S. Schlipf, editors, *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007,*

Tempe, AZ, USA, May 15-17, 2007, Proceedings, volume 4483 of *LNCS*, pages 266–271. Springer, 2007.

[16] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proc. ICLP/SLP*, pages 1070–1080, 1988.

[17] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Comput.*, 9(3/4):365–386, 1991.

[18] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 1999*, pages 21–32. ACM Press, 1999.

[19] Georg Gottlob, Nicola Leone, and Francesco Scarcello. The complexity of acyclic conjunctive queries. *J. ACM*, 48(3):431–498, 2001.

[20] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The dlv system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, 2006.

[21] Maximilian Möller, Marius Thomas Schneider, Martin Wegner, and Torsten Schaub. Centurio, a general game player: Parallel, Java- and ASP-based. *Künstliche Intelligenz*, 25(1):17–24, 2011.

[22] Ilkka Niemelä and Patrik Simons. Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Logic Programming and Nonmonotonic Reasoning, 4th International Conference, LPNMR'97, Dagstuhl Castle, Germany. Proceedings*, volume 1265 of *Lecture Notes in Computer Science*, pages 421–430. Springer, 1997.

[23] Neil Robertson and Paul D. Seymour. Graph minors. III. Planar tree-width. *J. Comb. Theory, Ser. B*, 36(1):49–64, 1984.