Extended ASP Tableaux and Rule Redundancy in Normal Logic Programs

Matti Järvisalo and Emilia Oikarinen

Laboratory for Theoretical Computer Science
P.O. Box 5400, FI-02015 Helsinki University of Technology (TKK), Finland

Abstract. We introduce an extended tableau calculus for answer set programming (ASP). The proof system is based on the ASP tableaux defined in [Gebser&Schaub, ICLP 2006], with an added extension rule. We investigate the power of Extended ASP Tableaux both theoretically and empirically. We study the relationship of Extended ASP Tableaux with the Extended Resolution proof system defined by Tseitin for clause sets, and separate Extended ASP Tableaux from ASP Tableaux by giving a polynomial length proof of a family of normal logic programs $\{\Pi_n\}$ for which ASP Tableaux has exponential length minimal proofs with respect to n. Additionally, Extended ASP Tableaux imply interesting insight into the effect of program simplification on the length of proofs in ASP. Closely related to Extended ASP Tableaux, we empirically investigate the effect of redundant rules on the efficiency of ASP solving.

1 Introduction

Answer set programming (ASP) is a declarative problem solving paradigm which has proven successful for a variety of knowledge representation and reasoning tasks. The success has been brought forth by efficient solver implementations bringing the theoretical underpinnings into practice. However, there has been an evident lack of theoretical studies into the reasons for the efficiency of current ASP solvers (e.g. [1–4]). Solver implementations and their inference techniques can be seen as determinisations of the underlying rule-based *proof systems*. Due to this strong interplay between theory and practice, the study of the relative efficiency of these proof systems reveals important new viewpoints and explanations for the successes and failures of particular solver techniques. While such proof complexity [5] studies are frequent in the closely related field of propositional satisfiability (SAT), where typical solvers have been shown to be based on refinements of the well-known Resolution proof system [6], this has not been the case for ASP. Especially, the inference techniques applied in current state-of-the-art ASP solvers have been characterised by a family of tableau-style ASP proof systems for normal logic programs only very recently [7], with some related preliminary proof complexity theoretic investigations [8]. The close relation of ASP and SAT and the respective theoretical underpinning of practical solver techniques has also received little attention up until recently [9, 10], although the fields could gain much by further studies on these connections.

This paper continues in part bridging the gap between ASP and SAT. Influenced by Tseitin's *Extended Resolution* proof system [11] for clausal formulas, we introduce

Extended ASP Tableaux, an extended tableau calculus based on the proof system in [7]. The motivations for Extended ASP Tableaux are many-fold. Theoretically, Extended Resolution has proven to be among the most powerful known proof systems, equivalent to, e.g., extended Frege systems; no exponential lower bounds for the lengths of proofs are known for Extended Resolution. We study the power of Extended ASP Tableaux, showing a tight correspondence with Extended Resolution.

The contributions of this paper are not only of theoretical nature. Extended ASP Tableaux is in fact based on *adding structure* into programs by introducing additional *redundant rules*. On the practical level, structure of problem instances has an important role in both ASP and SAT solving. Typically, it is widely believed that redundancy can and should be removed for practical efficiency. However, the power of Extended ASP Tableaux reveals that this is not generally the case, and such redundancy removing *simplification* mechanism can drastically hinder efficiency. In addition, we contribute by studying the effect of redundancy on the efficiency of a variety of ASP solvers. The results show that the role of redundancy in programs is not as simple as typically believed, and controlled addition of redundancy may in fact prove to be relevant in further strengthening the robustness of current solver techniques.

The paper is organised as follows. After preliminaries on ASP and SAT (Sect. 2), the relationship of Resolution and ASP Tableaux proof systems and concepts related to the complexity of proofs are discussed (Sect. 3). By introducing the Extended ASP Tableaux proof system (Sect. 4), proof complexity and simplification are then studied w.r.t. Extended ASP Tableaux (Sect. 5). Experimental results related to Extended ASP Tableaux and redundant rules in normal logic programs are presented in Sect. 6.

2 Preliminaries

As preliminaries we review basic concepts related to answer set programming (ASP) in the context of normal logic programs, propositional satisfiability (SAT), and translations between ASP and SAT.

2.1 Normal Logic Programs and Stable Models

We consider *normal logic programs* (NLPs) in the *propositional* case. The symbol "$\sim$" denotes *default negation*. A *default literal* is an atom, a, or its default negation, $\sim a$. We define shorthands $L^+ = \{a \mid a \in L\}$ and $L^- = \{a \mid \sim a \in L\}$ for a set of default literals L, and $\sim A = \{\sim a \mid a \in A\}$ for a set of atoms A. A program Π over the set of propositional atoms $\mathsf{atoms}(\Pi)$ consists of a finite set of rules r of the form

$$h \leftarrow a_1, \ldots, a_n, \sim b_1, \ldots, \sim b_m, \tag{1}$$

where $h \in \mathsf{atoms}(\Pi) \cup \{\bot\}$ and $a_i, b_j \in \mathsf{atoms}(\Pi)$. A rule consists of a *head*, $\mathsf{head}(r) = h$, and a *body*, $\mathsf{body}(r) = \{a_1, \ldots, a_n, \sim b_1, \ldots, \sim b_m\}$. This allows the shorthand $\mathsf{head}(r) \leftarrow \mathsf{body}(r)^+ \cup \sim \mathsf{body}(r)^-$ for (1). A rule r is a *fact* if $|\mathsf{body}(r)| = 0$. We define $\mathsf{head}(\Pi) = \bigcup_{r \in \Pi}\{\mathsf{head}(r)\}$ and $\mathsf{body}(\Pi) = \bigcup_{r \in \Pi}\{\mathsf{body}(r)\}$. The set of default literals of a program Π is $\mathsf{dlits}(\Pi) = \{a, \sim a \mid a \in \mathsf{atoms}(\Pi)\}$.

In ASP, we are interested in *stable models* [12] (or *answer sets*) of a program Π. An *interpretation* $M \subseteq \mathsf{atoms}(\Pi)$ defines which atoms of Π are true ($a \in M$) and which are false ($a \notin M$). An interpretation $M \subseteq \mathsf{atoms}(\Pi)$ is a *(classical) model* of Π if and only if $\mathsf{body}(r)^+ \subseteq M$ and $\mathsf{body}(r)^- \cap M = \emptyset$ imply $\mathsf{head}(r) \in M$ for each rule $r \in \Pi$. A model M is a stable model of a program Π if and only if there is no model $M' \subset M$ for Π^M, where

$$\Pi^M = \{\mathsf{head}(r) \leftarrow \mathsf{body}(r)^+ \mid r \in \Pi \text{ and } M \cap \mathsf{body}(r)^- = \emptyset\}$$

is called the *Gelfond-Lifschitz reduct* of Π with respect to M. We say that a program Π is *satisfiable* if it has a stable model, and *unsatisfiable* otherwise.

Given $a, b \in \mathsf{atoms}(\Pi)$, we say that b *depends directly* on a, denoted $a \leq_1 b$, if and only if there is a rule $r \in \Pi$ such that $b = \mathsf{head}(r)$ and $a \in \mathsf{body}(r)^+$. The *positive dependency graph* of Π, denoted by $\mathrm{Dep}^+(\Pi)$, is a directed graph with $\mathsf{atoms}(\Pi)$ and $\{\langle b, a\rangle \mid a \leq_1 b\}$ as the sets of vertices and edges, respectively. A NLP is *tight* if and only if its positive dependency graph is acyclic. We denote by $\mathsf{loop}(\Pi)$ the set of all loops in $\mathrm{Dep}^+(\Pi)$. Furthermore, the *external bodies* of a set of atoms A in Π is $\mathsf{eb}(A) = \{\mathsf{body}(r) \mid r \in \Pi,\ \mathsf{head}(r) \in A,\ \mathsf{body}(r)^+ \cap A = \emptyset\}$. A set $U \subseteq \mathsf{atoms}(\Pi)$ is *unfounded* if $\mathsf{eb}(U) = \emptyset$. We denote the *greatest unfounded set*, i.e., the union of all unfounded sets, of Π by $\mathsf{gus}(\Pi)$.

2.2 Propositional Satisfiability

Let X be a set of Boolean variables. Associated with every variable $x \in X$ there are two *literals*, the positive literal, denoted by x, and the negative literal, denoted by $\bar{x}$. A *clause* is a disjunction of distinct literals. We adopt the standard convention of viewing a clause as a finite set of literals and a CNF formula as a finite set of clauses. The sets of variables and literals appearing in a set of clauses $\mathcal{C}$ are denoted by $\mathsf{vars}(\mathcal{C})$ and $\mathsf{lits}(\mathcal{C})$.

A *truth assignment* τ associates a truth value $\tau(x) \in \{\mathsf{false}, \mathsf{true}\}$ with each variable $x \in X$. A truth assignment *satisfies* a set of clauses if it satisfies every clause in it. A clause is satisfied if it contains at least one satisfied literal, where a literal x (respectively, $\bar{x}$) is satisfied if $\tau(x) = \mathsf{true}$ (respectively, $\tau(x) = \mathsf{false}$). A clause set is *satisfiable* if there is a truth assignment that satisfies it, and *unsatisfiable* otherwise.

2.3 SAT as ASP

There is a natural linear-size translation from sets of clauses to normal logic programs so that the stable models of the encoding represent the satisfying truth assignments of the original clause set [13] *faithfully*, i.e., there is a bijective correspondence between the satisfying truth assignments and stable models of the translation. Given a clause set $\mathcal{C}$, this translation $\mathsf{nlp}(\mathcal{C})$ introduces a new atom c for each clause $C \in \mathcal{C}$, and atoms a_x and $\hat{a}_x$ for each variable $x \in \mathsf{vars}(\mathcal{C})$. The resulting NLP is then

$$\mathsf{nlp}(\mathcal{C}) := \bigcup_{x \in \mathsf{vars}(\mathcal{C})} \{\{a_x \leftarrow {\sim}\hat{a}_x\} \cup \{\hat{a}_x \leftarrow {\sim}a_x\}\} \cup \bigcup_{C \in \mathcal{C}} \{\bot \leftarrow {\sim}c\} \cup \quad (2)$$

$$\bigcup_{C \in \mathcal{C}} \{\{c \leftarrow a_x \mid x \in \mathsf{lits}(C)\} \cup \{c \leftarrow \hat{a}_x \mid \bar{x} \in \mathsf{lits}(C)\}\}. \quad (3)$$

The rules (2) encode the facts that (i) each variable is assigned an unambiguous truth value and that (ii) each clause in $\mathcal{C}$ must be satisfied, while (3) encodes that each clause is satisfied if at least one of its literals is satisfied.

2.4 ASP as SAT

Contrarily to the case of translating SAT into ASP, there is no modular [1] and faithful translation from normal logic programs to propositional logic [13]. Moreover, any faithful translation is potentially of exponential size when additional variables are not allowed [14] [2]. However, if a program Π satisfies the syntactic *tightness* condition, the answer sets of Π can be characterised faithfully by the classical models of a linear-size propositional formula called *Clark's completion* [19, 20] of Π, defined using a Boolean variable x_a for each $a \in \mathsf{atoms}(\Pi)$ as

$$C(\Pi) = \bigwedge_{h \in \mathsf{atoms}(\Pi)} \left(x_h \Leftrightarrow \bigvee_{B \in \mathsf{body}(h)} \left(\bigwedge_{b \in B^+} x_b \wedge \bigwedge_{b \in B^-} \bar{x}_b \right) \right), \tag{4}$$

where $\mathsf{body}(h) = \{\mathsf{body}(r) \mid \mathsf{head}(r) = h\}$. For simplicity, we have the special cases that (i) if x_h is $\perp$ then the equivalence becomes the negation of the right hand side, and (ii) if $h \in \mathsf{facts}(\Pi)$ then the equivalence reduces to the clause $\{x_h\}$.

As in this paper, often one needs to consider the clausal representation of Boolean formulas. For a linear-size clausal translation of $C(\Pi)$, introduce additionally a new Boolean variable x_B for each $B \in \mathsf{body}(\Pi) \setminus \{\emptyset\}$. Using these new variables, we arrive at the *clausal completion*

$$\mathsf{comp}(\Pi) := \bigcup_{B \in \mathsf{body}(\Pi) \setminus \{\emptyset\}} \left\{ x_B \equiv \bigwedge_{b \in B^+} x_b \wedge \bigwedge_{b \in B^-} \bar{x}_b \right\} \cup \bigcup_{B \in \mathsf{body}(\perp)} \{\bar{x}_B\} \tag{5}$$

$$\cup \bigcup_{\substack{h \in \mathsf{head}(\Pi) \setminus \{\perp\} \\ h \notin \mathsf{facts}(\Pi)}} \left\{ x_h \equiv \bigvee_{B \in \mathsf{body}(h)} x_B \right\} \cup \bigcup_{h \in \mathsf{facts}(\Pi)} \{x_h\} \tag{6}$$

$$\cup \bigcup_{a \in \mathsf{atoms}(\Pi) \setminus \mathsf{head}(\Pi)} \{\bar{x}_a\}, \tag{7}$$

where the shorthands $x \equiv \bigwedge_{x_i \in X} x_i$ and $x \equiv \bigvee_{x_i \in X} x_i$ stand for the sets of clauses $\{\bar{x}_1, \ldots \bar{x}_n, x\} \cup \bigcup_{x_i \in X} \{x_i, \bar{x}\}$ and $\{x_1, \ldots x_n, \bar{x}\} \cup \bigcup_{x_i \in X} \{\bar{x}_i, x\}$, respectively. For an example of a logic program's clausal completion, see Fig. 1(left).

[1] Intuitively, for a modular translation, adding an atom to a program leads to a local change not involving the translation of the rest of the program [13].

[2] However, polynomial size propositional encodings using extra variables are known, e.g. [15, 16]. Also, ASP as Propositional Satisfiability approaches for solving normal logic programs have been developed, e.g., ASSAT [17] (based on incrementally adding loop formulas) and ASP-SAT [18] (based on generating a classical model and testing its minimality).

3 Proof Systems for ASP and SAT

In this section we review concepts related to proof complexity (see, e.g., [5]) in the context of this paper, and discuss the relationship of Resolution and ASP Tableaux [7].

3.1 Propositional Proof Systems and Complexity

Formally, a *(propositional) proof system* is a polynomial-time computable predicate S such that a propositional expression E is unsatisfiable if and only if there is a *proof* p for which $S(E, p)$. A proof system is thus a polynomial-time procedure for checking the correctness of proofs in a certain format. While proof checking is efficient, finding short proofs may be difficult, or, generally, impossible since short proofs may not exist for a too weak proof system. As a measure of hardness of proving unsatisfiability of an expression E in a proof system S, the *(proof) complexity* of E in S is the *length* of the shortest proof of E in S. For a family $\{E_n\}$ of unsatisfiable expressions over increasing number of variables, the (asymptotic) complexity of $\{E_n\}$ is measured with respect to the sizes of E_n.

For two proof systems S,S', we say that S' *(polynomially) simulates* S if for all families $\{E_n\}$ it holds that $C_{S'}(E_n) \leq p(C_S(E_n))$ for all E_n, where p is a polynomial, and C_S and $C_{S'}$ are the complexities in S and S', respectively. If S simulates S' and vice versa, then S and S' are *polynomially equivalent*. If there is a family $\{E_n\}$ for which S' does not polynomially simulate S, we say that $\{E_n\}$ *separates* S from S', and S is *stronger* than S'.

3.2 Resolution

The well-known Resolution proof system (RES) for clause sets is based on the *resolution rule*. Let C, D be clauses, and x a Boolean variable. The resolution rule states that we can *directly derive* $C \cup D$ from $\{x\} \cup C$ and $\{\bar{x}\} \cup D$ by *resolving on* x.

A RES *derivation* of a clause C from a clause set $\mathcal{C}$ is a sequence of clauses $\pi = (C_1, C_2, \ldots, C_n)$, where $C_n = C$ and each C_i, where $1 \leq i < n$, is either (i) a clause in $\mathcal{C}$ (an *initial clause*), or (ii) derived with the resolution rule from two clauses C_j, C_k where $j, k < i$ (a *derived clause*). The *length* of π is n, the number of clauses occurring in it. Any derivation of the empty clause $\emptyset$ from $\mathcal{C}$ is a RES *proof* of $\mathcal{C}$.

Any RES proof $\pi = (C_1, C_2, \ldots, C_n)$ can be presented as a directed acyclic graph, in which the leafs are initial clauses, inner nodes are derived clauses, and the root is the empty clause. There are edges from C_i and C_j to C_k iff C_k has been directly derived from C_i and C_j using the resolution rule. Many *Resolution refinements*, in which the structure of the graph representation is restricted, have been proposed and studied. Of particular interest here is *Tree-like Resolution* (T-RES), in which it is required that proofs are represented by trees. This implies that a derived clause, if subsequently used multiple times in the proof, must be derived anew each time from initial clauses.

T-RES is a *proper* RES refinement, i.e., RES is stronger than T-RES [21]. On the other hand, it is well known that the DPLL method [22], the basis of most state-of-the-art SAT solvers, is polynomially equivalent to T-RES. However, conflict-learning DPLL is stronger than T-RES, and polynomially equivalent to RES under a slight generalisation [6].

3.3 ASP Tableaux

Although ASP solvers for normal logic programs have been available for many years, the deduction rules applied in such solvers have only recently been formally defined as a proof system, which we will here refer to as ASP Tableaux [7] (ASP-T).

An ASP tableau for a NLP Π is a binary tree of the following structure. The *root* of the tableau consists of the rules Π and the *entry* $\mathbf{F}\bot$ for capturing that $\bot$ is always false. The non-root nodes of the tableau are single *entries* of the form $\mathbf{T}a$ or $\mathbf{F}a$, where $a \in \mathsf{atoms}(\Pi) \cup \mathsf{body}(\Pi)$. As typical for tableau methods, entries are generated by *extending* a *branch* (a path from the root to a leaf node) by applying one of the rules in Fig.2; if the prerequisites of a rule hold in a branch, the branch can be extended with the entries specified by the rule. For convenience, we have the shorthand $\mathbf{t}l$ ($\mathbf{f}l$) for literals, defined as $\mathbf{T}l$ if l is positive (negative), and $\mathbf{F}l$ if l is negative (positive).

A branch is *closed under* the *deduction rules* (b)-(i) if the branch cannot be extended using the rules. A branch is *contradictory* if there are entries $\mathbf{T}a, \mathbf{F}a$ for some a. A branch is *complete* if it is contradictory or if there is the entry $\mathbf{T}a$ or $\mathbf{F}a$ for each $a \in \mathsf{atoms}(\Pi) \cup \mathsf{body}(\Pi)$ and the branch is closed under the deduction rules. A tableau is complete if all its branches are complete. A complete tableau from Π in which all branches are contradictory is an ASP-T proof of the unsatisfiability of Π. The *length* of an ASP-T proof is the number of entries in it. In Fig. 1 an ASP-T proof is presented for the program Π given on the left of the proof, with the rule applied for deducing each entry given in parenthesis.

$$
\begin{aligned}
\Pi &= \{a \leftarrow {\sim}a, b.\ b \leftarrow c.\ c \leftarrow {\sim}b\} \\
\mathsf{comp}(\Pi) &= \{\{\bar{x}_{\{\sim a,b\}}, \bar{x}_a\}, \{\bar{x}_{\{\sim a,b\}}, x_b\}, \\
&\quad \{x_{\{\sim a,b\}}, x_a, \bar{x}_b\}, \{\bar{x}_{\{c\}}, x_c\}, \\
&\quad \{x_{\{c\}}, \bar{x}_c\}, \{\bar{x}_{\{\sim b\}}, \bar{x}_b\}, \\
&\quad \{x_{\{\sim b\}}, x_b\}, \{\bar{x}_a, x_{\{\sim a,b\}}\}, \\
&\quad \{x_a, \bar{x}_{\{\sim a,b\}}\}, \{\bar{x}_b, x_{\{c\}}\}, \\
&\quad \{x_b, \bar{x}_{\{c\}}\}, \{\bar{x}_c, x_{\{\sim b\}}\}, \{x_c, \bar{x}_{\{\sim b\}}\}\}
\end{aligned}
$$

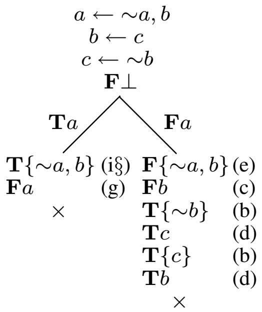


Fig. 1. A logic program Π, its clausal completion $\mathsf{comp}(\Pi)$, and an ASP-T proof for Π.

Any branch B describes a *partial assignment* $\mathcal{A}$ on $\mathsf{atoms}(\Pi)$ in a natural way, i.e., if there is an entry $\mathbf{T}a$ ($\mathbf{F}a$, respectively) in B for $a \in \mathsf{atoms}(\Pi)$, then $(a, \mathsf{true}) \in \mathcal{A}$ ($(a, \mathsf{false}) \in \mathcal{A}$, respectively). ASP-T is a sound and complete proof system for normal logic programs [7], i.e., there is a complete non-contradictory ASP tableau from Π if and only if Π is satisfiable. Thus the assignment $\mathcal{A}$ described by a complete non-contradictory branch gives a stable model $M = \{a \in \mathsf{atoms}(\Pi) \mid (a, \mathsf{true}) \in \mathcal{A}\}$. As argumented in [7], current ASP solver implementations are tightly related to ASP-T, with the intuition that the cut rule is determinised with decision heuristics, while the deduction rules describe the propagation mechanism in ASP solvers. For instance, the noMore++ system [2] is a determinisation of the rules (a)-(g),(h§),(h†),(i§), while smodels [1] applies the same rules with the cut rule restricted to $\mathsf{atoms}(\Pi)$.

$$\frac{}{\mathbf{T}\phi \mid \mathbf{F}\phi}\ (\natural)$$

(a) Cut

$$\frac{\begin{array}{c} h \leftarrow l_1, \ldots, l_n \\ \mathbf{t}l_1, \ldots, \mathbf{t}l_n \end{array}}{\mathbf{T}\{l_1, \ldots, l_n\}}$$

(b) Forward True Body

$$\frac{\begin{array}{c} \mathbf{F}\{l_1, \ldots, l_i, \ldots, l_n\} \\ \mathbf{t}l_1, \ldots, \mathbf{t}l_{i-1}, \mathbf{t}l_{i+1}, \ldots, \mathbf{t}l_n \end{array}}{\mathbf{f}l_i}$$

(c) Backward False Body

$$\frac{\begin{array}{c} h \leftarrow l_1, \ldots, l_n \\ \mathbf{T}\{l_1, \ldots, l_n\} \end{array}}{\mathbf{T}h}$$

(d) Forward True Atom

$$\frac{\begin{array}{c} h \leftarrow l_1, \ldots, l_n \\ \mathbf{F}h \end{array}}{\mathbf{F}\{l_1, \ldots, l_n\}}$$

(e) Backward False Atom

$$\frac{\begin{array}{c} h \leftarrow l_1, \ldots, l_i, \ldots, l_n \\ \mathbf{f}l_i \end{array}}{\mathbf{F}\{l_1, \ldots, l_i, \ldots, l_n\}}$$

(f) Forward False Body

$$\frac{\mathbf{T}\{l_1, \ldots, l_i, \ldots, l_n\}}{\mathbf{t}l_i}$$

(g) Backward True Body

$$\frac{\mathbf{F}B_1, \ldots, \mathbf{F}B_m}{\mathbf{F}h}\ (\flat)$$

(h)

$$\frac{\begin{array}{c} \mathbf{T}h \\ \mathbf{F}B_1, \ldots, \mathbf{F}B_{i-1}, \mathbf{F}B_{i+1}, \ldots, \mathbf{F}B_m \end{array}}{\mathbf{T}B_i}\ (\sharp)$$

(i)

($\natural$): $\phi \in \mathsf{atoms}(\Pi) \cup \mathsf{body}(\Pi)$
($\flat$): § (Forward False Atom) or † (Well-Founded Negation) or ‡ (Forward Loop)
($\sharp$): § (Backward True Atom) or † (Well-Founded Justification) or ‡ (Backward Loop)
(§): $\mathsf{body}(h) = \{B_1, \ldots, B_m\}$
(†): $\{B_1, \ldots, B_m\} \subseteq \mathsf{body}(\Pi)$ and $h \in \mathsf{gus}(\{r \in \Pi \mid \mathsf{body}(r) \notin \{B_1, \ldots B_m\}\})$
(‡): $h \in L$, $L \in \mathsf{loop}(\Pi)$, $\mathsf{eb}(L) = \{B_1, \ldots, B_m\}$

Fig. 2. Rules in ASP Tableaux

Interestingly, ASP-T and T-RES are polynomially equivalent under the translations comp and nlp. Although the similarity of DPLL's unit propagation and propagation in ASP solvers is discussed in [9, 10], here we want to stress the direct connection between ASP-T and T-RES.

Theorem 1. *Considering tight programs,* T-RES *under the translation* comp *can polynomially simulate* ASP-T.

The intuitive idea of the proof of Theorem 1 is the following. Consider again the tight NLP Π and the ASP-T proof T in Fig. 1. The completion $\mathsf{comp}(\Pi)$ is also shown in Fig. 1. We transform T into a binary *cut tree* T' where every entry generated by a deduction rule in T is replaced by an application of the cut rule on the corresponding entry. See Fig. 3 (left) for the cut tree corresponding to the ASP-T proof in Fig. 1. Now there is a T-RES proof of $\mathsf{comp}(\Pi)$ such that for any prefix p of an arbitrary branch B in T' there is a clause $C \in \pi$ contradictory to the partial assignment in p, i.e., there is the entry $\mathbf{F}a$ ($\mathbf{T}a$) in p for each corresponding positive literal x_a (negative literal $\bar{x}_a$) in C. Furthermore, each such C has a Tree-like Resolution derivation from $\mathsf{comp}(\Pi)$ of polynomial length w.r.t. the postfix of B starting directly after p. When reaching the root of T', we must have derived $\emptyset$ since it is contradictory with the empty assignment. The T-RES proof resulting from the cut tree in Fig. 3 (left) is shown in Fig. 3 (right).

The reverse direction, as stated by Theorem 2, follows from a similar argument.

Theorem 2. *Considering clause sets,* ASP-T *under the translation* nlp *can polynomially simulate* T-RES.

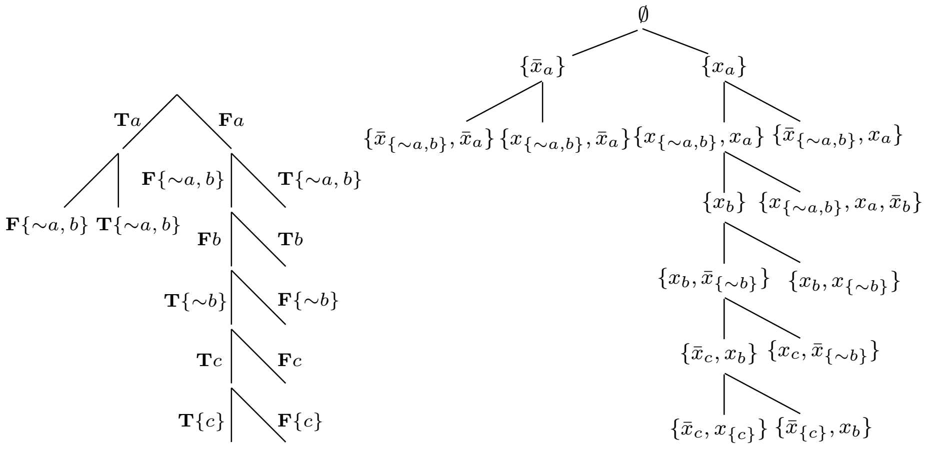


Fig. 3. Left: cut tree based on the ASP-T proof in Fig. 1. Right: resulting T-RES proof.

4 Extended ASP Tableaux

We will now introduce an *extension rule* to ASP-T, which results in *Extended ASP Tableaux* (E-ASP-T), an extended tableau proof system for ASP. The idea is that one can define names for conjunctions of default literals, i.e., given two $l_1, l_2 \in \mathsf{dlits}(\Pi)$, the (elementary) extension rule allows adding the rule $p \leftarrow l_1, l_2$ to Π, where $p \notin \mathsf{atoms}(\Pi) \cup \{\bot\}$. It is essential that p is a new atom for preserving satisfiability.

When convenient, we will apply a generalisation of the elementary extension. By allowing one to introduce multiple bodies for p, the general extension rule [3] is

$$\Pi := \Pi \cup \bigcup_i \{p \leftarrow l_{i,1}, \ldots, l_{i,k_i} \mid l_{j,k} \in \mathsf{dlits}(\Pi),\ p \notin \mathsf{atoms}(\Pi) \cup \{\bot\}\}.$$

An E-ASP-T proof of program Π is an ASP-T proof T of $\Pi \cup E$, where E is a set of *extending rules* generated with the extension rule. The length of an E-ASP-T proof is the length of T plus the number of rules in E.

Since $\mathsf{head}(r) \notin \mathsf{atoms}(\Pi) \cup \{\bot\}$ for all extending rules $r \in E$, the extension rule does not affect the existence of stable models, i.e., for each stable model M of Π, there is a unique $N \subseteq \mathsf{atoms}(E) \setminus \mathsf{atoms}(\Pi)$ such that $M \cup N$ is a stable model of $\Pi \cup E$. Thus E-ASP-T is a sound and complete proof system.

5 Proof Complexity

In this section we study proof complexity theoretic issues related to E-ASP-T from several viewpoints: we (i) consider the relationship between E-ASP-T and Tseitin's

[3] Notice equivalent constructs can be introduced with the elementary rule. For example, using additional new atoms, bodies with more than two literals can be decomposed with balanced parentheses.

Extended Resolution, (ii) give an explicit separation of E-ASP-T from ASP-T, and (iii) relate the extension rule to the effect of program simplification on proof lengths.

5.1 Relationship with Extended Resolution

E-ASP-T is motivated by Extended Resolution (E-RES), a proof system by Tseitin [11]. E-RES consists of the resolution rule and an extension rule which allows one to introduce equivalences of the form $x \equiv l_1 \wedge l_2$, where x is a new variable and l_1, l_2 literals in the clause set. In other words, given a clause set $\mathcal{C}$, one application of the extension rule adds the clauses $\{\bar{x}, l_1\}$, $\{\bar{x}, l_2\}$, and $\{x, \bar{l}_1, \bar{l}_2\}$ to $\mathcal{C}$. E-RES is known to be more powerful than RES; in fact, E-RES is polynomially equivalent with, e.g., extended Frege systems, and no superpolynomial proof complexity lower bounds are known for E-RES. We will now relate E-ASP-T with E-RES, and show that they are polynomially equivalent under the translations comp and nlp.

Theorem 3. E-RES *and* E-ASP-T *are polynomially equivalent proof systems in the sense that*

(i) considering tight normal logic programs, E-RES *under the translation* comp *polynomially simulates* E-ASP-T*, and*
(ii) considering clause sets, E-ASP-T *under the translation* nlp *polynomially simulates* E-RES.

Proof. (i): Let T be an E-ASP-T proof for a tight NLP Π, i.e., T is an ASP-T proof of $\Pi \cup E$, where E is the extension of Π. We use the shorthand x_l for the variable corresponding to default literal l in $\mathsf{comp}(\Pi)$, i.e., $x_l = x_a$ ($x_l = \bar{x}_a$) if $l = a$ ($l = {\sim}a$) for $a \in \mathsf{atoms}(\Pi)$. By Theorem 1 there is a polynomial T-RES proof for $\mathsf{comp}(\Pi \cup E)$. Since $\mathsf{head}(E) \cap (\mathsf{atoms}(\Pi) \cup \{\perp\}) = \emptyset$, the clauses introduced for $\mathsf{head}(E)$ in $\mathsf{comp}(\Pi \cup E)$ can be seen as extensions in E-RES, i.e., for each $h \leftarrow l_1, l_2 \in E$ there are the clauses $x_h \equiv x_{l_1} \wedge x_{l_2}$ in $\mathsf{comp}(\Pi \cup E)$. Thus there is an extension E' for $\mathsf{comp}(\Pi)$ such that the T-RES proof of $\mathsf{comp}(\Pi \cup E)$ is an E-RES proof of $\mathsf{comp}(\Pi)$.

(ii): Let $\pi = (C_1, \ldots, C_n = \emptyset)$ be an E-RES proof of a set of clauses $\mathcal{C}$. Let E be the set of clauses generated with the extension rule in π. We introduce shorthands for atoms corresponding to literals, i.e., $a_l = a_x$ ($a_l = {\sim}a_x$) if $l = x$ ($l = \bar{x}$) for $x \in \mathsf{atoms}(\mathcal{C})$. We add the following rules to $\mathsf{nlp}(\mathcal{C})$ with the ASP extension rule: $a_x \leftarrow a_{l_1}, a_{l_2}$ for each extension $x \equiv l_1 \wedge l_2$; $c \leftarrow a_l$ for each $l \in C$ in π such that $C \notin \mathcal{C}$; and $p_1 \leftarrow c_1$ and $p_i \leftarrow c_i, p_{i-1}$ for each $C_i \in \pi$ and $2 \leq i < n$.

An E-ASP-T proof for $\mathsf{nlp}(\mathcal{C})$ is generated as follows. From $i = 1$ to $n-1$ apply the cut rule on p_i in the branch with $\mathbf{T}p_j$ for all $j < i$. We notice that each branch with $\mathbf{F}p_i$ and $\mathbf{T}p_j$ for all $j < i$ closes without further application of the cut. We can deduce $\mathbf{F}c_i$ from $\mathbf{F}p_i$. Now either (i) $C_i \in \mathcal{C}$, (ii) C_i is a derived clause, or (iii) $C_i \in E$. For instance, if $C_i = \{\bar{x}, l_1\}$ from the extension $x \equiv l_1 \wedge l_2$, then from $c_i \leftarrow {\sim}a_x$ and $c_i \leftarrow a_{l_1}$ we deduce $\mathbf{T}a_x$ and $\mathbf{F}a_{l_1}$. The branch closes as $\mathbf{T}\{a_{l_1}, a_{l_2}\}$ and $\mathbf{T}a_{l_1}$ are deduced from $a_x \leftarrow a_{l_1}, a_{l_2}$. Other cases are similar.

Now, consider the branch with $\mathbf{T}p_i$ for all $i = 1 \ldots n-1$. The empty clause C_n is obtained by resolving $C_j = \{x\}$ and $C_k = \{\bar{x}\}$, $j, k < n$. Thus we can deduce $\mathbf{T}c_j$ and $\mathbf{T}c_k$ from $p_j \leftarrow c_j, p_{j-1}$ and $p_k \leftarrow c_k, p_{k-1}$, respectively, and furthermore, $\mathbf{T}a_x$

and $\mathbf{F}a_x$ from $c_j \leftarrow l$ and $c_k \leftarrow \bar{l}$ ($l = x$ or $l = \bar{x}$), closing the branch. The obtained contradictory ASP tableau is of linear length w.r.t. π. □

5.2 Pigeonhole Principle Separates Extended ASP Tableaux from ASP Tableaux

As an example, we now consider a family of normal logic programs $\{\Pi_n\}$ which separates E-ASP-T from ASP-T, i.e., we give an explicit polynomial length proof of Π_n for which ASP-T has exponential length minimal proofs with respect to n. We will consider this family also in the experiments of this paper.

The program family $\{\mathrm{PHP}_n^{n+1}\}$ in question is the following typical encoding of the *pigeon-hole principle* as a normal logic program:

$$\mathrm{PHP}_n^{n+1} := \bigcup_{1\le i\le n+1} \{\bot \leftarrow {\sim}p_{i,1}, \dots, {\sim}p_{i,n}\} \cup \bigcup_{\substack{1\le i<j\le n+1\\1\le k\le n}} \{\bot \leftarrow p_{i,k}, p_{j,k}\} \quad (8)$$

$$\cup \bigcup_{\substack{1\le i\le n+1\\1\le j\le n}} \{\{p_{i,j} \leftarrow {\sim}p'_{i,j}\} \cup \{p'_{i,j} \leftarrow {\sim}p_{i,j}\}\} \quad (9)$$

In the above, $p_{i,j}$ has the interpretation that pigeon i sits in hole j. The rules in (8) require that (i) each pigeon must sit in some hole and that (ii) no two pigeons can sit in the same hole. The rules in (9) enforce that for each pigeon and each hole, the pigeon either sits in the hole or does not sit in the hole. Each PHP_n^{n+1} is unsatisfiable since there is no bijective mapping from an $(n+1)$-element set to an n-element set.

Theorem 4. *The complexity of* $\{\mathrm{PHP}_n^{n+1}\}$ *with respect to* n *is polynomial in* E-ASP-T *and exponential in* ASP-T

Proof. (i): Following Cook's extension [23] for achieving a polynomial-length E-RES proof of a clausal encoding of the pigeonhole principle[4], we define the polynomial size program extension

$$\mathrm{EXT}^l := \bigcup_{\substack{1\le i\le l\\1\le j\le l-1}} \{\{e^l_{i,j} \leftarrow e^{l+1}_{i,j}\} \cup \{e^l_{i,j} \leftarrow e^{l+1}_{i,l}, e^{l+1}_{l+1,j}\}\} \quad (10)$$

for $1 \le l \le n$, where each $e^{n+1}_{i,j}$ is interpreted as $p_{i,j}$.

Although not explicitly given by Cook, the extension given in [23] does not seem to yield a polynomial length *tree-like* proof of the clausal representation, so Theorem 3 does not directly imply a polynomial length ASP-T proof for $\mathrm{PHP}_n^{n+1} \cup \bigcup_{1\le l\le n} \mathrm{EXT}^l$. However, given the polynomial length E-RES proof[5] $\pi = (C_1, C_2, \dots, C_n = \emptyset)$ of the clausal representation, we can follow the general strategy given in the proof of Theorem 3 for defining an additional extension $E(\pi)$ which allows a polynomial length ASP-T proof for the resulting program

$$\mathrm{EPHP}_n^{n+1} := \mathrm{PHP}_n^{n+1} \cup \bigcup_{1\le l\le n} \mathrm{EXT}^l \cup E(\pi).$$

---

[4] The particular encoding is $\bigcup_{1\le i\le n+1}\{\bigvee_{j=1}^n x_{i,j}\} \cup \bigcup_{1\le i<i'\le n+1, 1\le j\le n}\{\neg x_{i,j} \vee \neg x_{i',j}\}$.

[5] The intuitive idea is that the extension allows for reducing PHP_n^{n+1} to PHP_{n-1}^n with a polynomial number of Resolution steps. Due to space constraints we do not give π explicitly here.

(ii): $\mathsf{comp}(\mathrm{PHP}_n^{n+1})$ consists of the clausal encoding of the pigeon-hole principle and additional clauses (tautologies) for rules of the form $a \leftarrow \sim a'$, $a' \leftarrow \sim a$. Assume now that there is a polynomial ASP-T proof for PHP_n^{n+1}. By Theorem 1 there is a polynomial T-RES of $\mathsf{comp}(\mathrm{PHP}_n^{n+1})$. It is easy to see that the additional tautologies in $\mathsf{comp}(\mathrm{PHP}_n^{n+1})$ do not help in the resolution proof. Thus there is a polynomial length T-RES proof for the clausal pigeonhole encoding. However, this contradicts the fact that the complexity of the clausal pigeonhole principle is exponential w.r.t. n for (Tree-like) Resolution [24]. $\square$

In fact, Theorem 4 is also witnessed by *non-tight* programs. Consider the family $\{\mathrm{PHP}_n^{n+1} \cup \{p_{i,j} \leftarrow p_{i,j} \mid 1 \leq i \leq n+1, 1 \leq j \leq n\}\}$, which is non-tight with the additional self-loops $\{p_{i,j} \leftarrow p_{i,j}\}$, but preserves (un)satisfiability of PHP_n^m for all n, m. Since the self-loops do not contribute to the proofs of PHP_n^{n+1}, ASP-T still has exponential length minimal proofs for these programs, while the E-ASP-T proof presented in the proof of Theorem 4 is still valid.

5.3 Program Simplification and Complexity

We will now give an interesting corollary of Theorem 4, addressing the effect of program simplification on the length of proofs.

Tightly related to the development of efficient solver implementations for resolving ASP programs arising from practical applications is the development of techniques for *simplifying* programs. Efficient program simplification through *local transformation rules* becomes especially important as practically relevant programs are often produced automatically, because often a high number of redundant constraints is produced in the process. While various satisfiability-preserving local transformation rules for simplifying logic programs have been introduced (see, e.g., [25]), the effect of applying such transformations on the lengths of proofs has not received attention.

Taking a first step into this direction, we now show that even simple transformation rules may have a drastic negative effect on proof complexity. Consider the local transformation rule $\mathsf{red}(\Pi) := \Pi \setminus \{r \in \Pi \mid \mathsf{head}(r) \notin \mathsf{body}(\Pi)\}$. The rules removed by red are redundant with respect to satisfiability of the program in the sense that red preserves *visible equivalence* [16]. The visible equivalence relation takes the interfaces of programs into account: $\mathsf{atoms}(\Pi)$ is partitioned into $\mathsf{v}(\Pi)$ and $\mathsf{h}(\Pi)$ determining the *visible* and the *hidden* atoms in Π, respectively. Programs Π_1 and Π_2 are visibly equivalent, denoted by $\Pi_1 \equiv_{\mathrm{v}} \Pi_2$, if and only if $\mathsf{v}(\Pi_1) = \mathsf{v}(\Pi_2)$ and there is a bijective correspondence between the stable models of Π_1 and Π_2 mapping each $a \in \mathsf{v}(\Pi_1)$ onto itself. Defining $\mathsf{v}(\Pi) = \mathsf{v}(\mathsf{red}(\Pi)) = \mathsf{atoms}(\mathsf{red}(\Pi))$, one can see that $\mathsf{red}(\Pi) \equiv_{\mathrm{v}} \Pi$.

A polynomial time, satisfiability-preserving simplification algorithm $\mathsf{red}^*(\Pi)$ is obtained by closing program Π under red. However, notice that, in the worst case when we define $\mathsf{v}(\mathrm{EPHP}_n^{n+1}) = \mathsf{v}(\mathrm{PHP}_n^{n+1}) = \mathsf{atoms}(\mathrm{PHP}_n^{n+1})$, we have $\mathsf{red}^*(\mathrm{EPHP}_n^{n+1}) = \mathrm{PHP}_n^{n+1}$. Thus, by Theorem 4, red^* transforms a program family having polynomial complexity in ASP Tableaux into one with exponential complexity with respect to n.

6 Experiments

We evaluate empirically how well current state-of-the-art ASP solvers can make use of the additional structure introduced to programs using the extension rule. We run the solvers smodels [1] (version 2.32, a widely used lookahead solver), clasp [4] (rc4, with many techniques—including conflict learning—adopted from DPLL-based SAT solvers), and cmodels [18] (version 3.66, a SAT-based ASP solver running the conflict-learning SAT solver zChaff version 2004.11.15 as the back-end). The experiments are run on standard PCs with 2-GHz AMD 3200+ processors under Linux.

First, we investigate whether ASP solvers are able to benefit from the extension in EPHP_n^{n+1}. We compare the number of decisions and running times of each of the solvers on PHP_n^{n+1}, $\mathrm{CPHP}_n^{n+1} := \mathrm{PHP}_n^{n+1} \cup \bigcup_{1 \le l \le n} \mathrm{EXT}^l$, and EPHP_n^{n+1}. By Theorem 4 the solvers should in theory be able to exhibit polynomially scaling number of decisions for EPHP_n^{n+1}. In fact with conflict-learning this might also be possible for CPHP_n^{n+1} due to the tight correspondence with conflict-learning SAT solvers and Resolution. The results for $n = 10 \ldots 12$ are shown in Table 1. While the number of decisions for the conflict-learning clasp and cmodels is somewhat reduced by the extensions, the solvers do not seem to be able to reproduce the polynomial size proofs, and we do not observe a dramatic change in the running times. With a timeout of 2 hours, smodels gives no answer for $n = 12$ on PHP_n^{n+1} or CPHP_n^{n+1}. However, for EPHP_n^{n+1} smodels returns without any branching, which should be due to the fact that smodels's complete lookahead notices that by branching on the critical extension atoms (as in part (ii) of the proof of Theorem 4) the false branch closes immediately. With this in mind, an interesting further study out of the scope of this paper would be the possibilities of integrating conflict learning techniques with (partial) lookahead.

In the second experiment, we study the effect of having a modest number of redundant rules on the behaviour of ASP solvers. For this we apply the following procedure $\textsc{AddRandomRedundancy}(\mathit{\Pi}, n, p)$:

1. **For** $i = 1$ **to** $\lfloor \frac{p}{100} n \rfloor$:
   1a. Randomly select $l_1, l_2 \in \mathsf{dlits}(\mathit{\Pi})$ such that $l_1 \neq l_2$.
   1b. $\mathit{\Pi} := \mathit{\Pi} \cup \{r_i \leftarrow l_1, l_2\}$, where $r_i \notin \mathsf{atoms}(\mathit{\Pi})$
2. **Return** $\mathit{\Pi}$

Given a program $\mathit{\Pi}$, the procedure iteratively adds rules of the form $r_i \leftarrow l_1, l_2$ to $\mathit{\Pi}$, where l_1, l_2 are random default literals currently in the program and r_i is a new atom. The number of introduced rules is $p\%$ of the integer n.

Table 1. Results on PHP_n^{n+1}, CPHP_n^{n+1}, and EPHP_n^{n+1} with timeout (-) of 2 hours.

		Time (s)			Decisions		
Solver	n	PHP_n^{n+1}	CPHP_n^{n+1}	EPHP_n^{n+1}	PHP_n^{n+1}	CPHP_n^{n+1}	EPHP_n^{n+1}
smodels	10	32.28	120.24	9.28	158878	141177	0
smodels	11	471.54	1828.40	23.07	1885949	1619703	0
smodels	12	-	-	52.20	-	-	0
clasp	10	8.60	7.78	19.26	197982	114840	38842
clasp	11	72.78	62.74	97.23	1072358	574874	116534
clasp	12	900.33	1046.86	881.90	7787578	4964309	646278
cmodels	10	1.91	2.23	27.42	9455	9916	20615
cmodels	11	7.99	10.28	70.39	23058	26283	38648
cmodels	12	48.36	56.70	270.63	87864	98994	97745

In Fig. 4, the median, minimum, and maximum number of decisions and running times for the solvers on $\textsc{AddRandomRedundancy}(\mathrm{PHP}_n^{n+1}, n, p)$ are shown for $p = 50, 100 \ldots, 450$ over 15 trials at each data point. The mean number of decisions (left) and running times (right) on the original PHP_n^{n+1} are presented by the horizontal lines. Notice that the number of added atoms and rules is linear to n, which is negligible to the number of atoms (in the order of n^2) and rules (n^3) in PHP_n^{n+1}. For similar running times, the number of holes n is 10 for clasp and smodels and 11 for cmodels. The results are very interesting: each of the solvers seems to react individually to the added redundancy. For cmodels (b), only a few added redundant rules are enough to worsen its behaviour. For smodels (c), the number of decisions decreases linearly with the number of added rules. However, the running times grow fast at the same time, most probably due to smodels's lookahead. We also ran the experiment for smodels (d) without using lookahead. This had a visible effect on the number of decisions, showing a benefit from the added rules compared to smodels on PHP_n^{n+1}.

The most interesting effect is seen for clasp; clasp benefits from the added rules w.r.t. the number of decision, while the running times stay similar on the average, contrarily to the other solvers. In addition to this robustness against redundancy, we believe that this shows promise for further exploiting redundancy added in a controlled way during search; the added rules give new possibilities to branch on definitions which were not available in the original program. However, for benefiting from redundancy with running times in mind, optimised lightweight propagation mechanisms are essential.

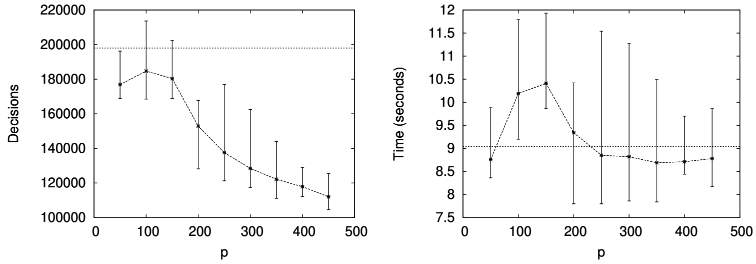
As a final remark, an interesting observation is that the effect of the transformation presented in [8], which enables smodels to branch on the bodies of rules, having an exponential effect on the proof complexity of a particular program family, can be equivalently obtained by applying the ASP extension rule. This may in part explain the effect on adding redundancy on the number of decision made by smodels.

7 Conclusions

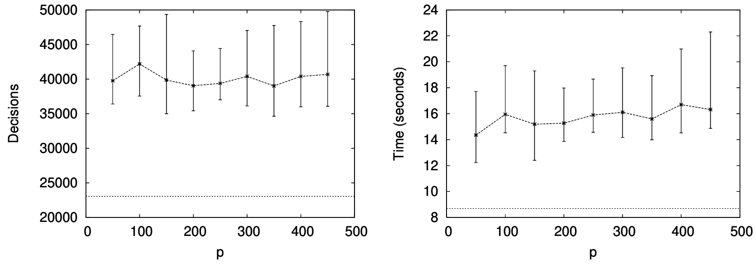
We introduce Extended ASP Tableaux, an extended tableau calculus for normal logic programs under the stable model semantics. We study the strength of the calculus, showing a tight correspondence with Extended Resolution, which is among the most powerful known propositional proof systems. This sheds further light on the relation of ASP and propositional satisfiability solving and their underlying proof systems, something which we believe is for the benefit of both of the communities.

Furthermore, this work shows the intricate nature of the interplay of structure and the hardness of solving ASP instances. We anticipate that controlled use of the extension rule is possible and will yield performance gains by considering in more detail the structural properties of programs in particular problem domains. One could also consider implementing branching on any possible formula *inside* a solver. However, this would require novel heuristics, since choosing the formula to branch on from the exponentially many alternatives is nontrivial and is not applied in current solvers. We find this an interesting future direction of research. Another important research direction set forth by this study is a more in-depth investigation into the effect of program simplification on the hardness of solving ASP instances.

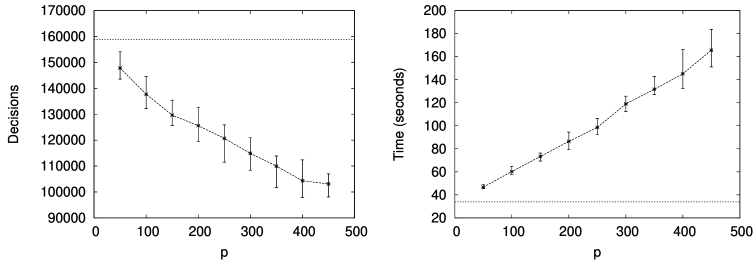
Acknowledgements. Financial support from Academy of Finland (grant #211025), Helsinki Graduate School in Computer Science and Engineering, Emil Aaltonen Foundation, the Finnish Cultural Foundation (EO), the Technological Foundation TES, and the Nokia Foundation (EO) is gratefully acknowledged.



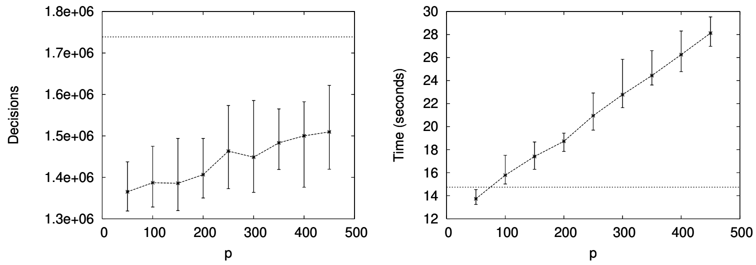
(a) clasp decisions (left), time in seconds (right)



(b) cmodels decisions (left), time in seconds (right)



(c) smodels decisions (left), time in seconds (right)



(d) smodels without lookahead: decisions (left), time in seconds (right)

Fig. 4. Effects of adding randomly generated redundant rules to PHP_n^{n+1}

References

1. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1-2) (2002) 181–234
2. Anger, C., Gebser, M., Linke, T., Neumann, A., Schaub, T.: The nomore++ approach to answer set solving. In: LPAR. Volume 3835 of LNCS., Springer (2005) 95–109
3. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM TOCL **7**(3) (2006) 499–562
4. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: IJCAI. (2007) 286–392
5. Beame, P., Pitassi, T.: Propositional proof complexity: Past, present, and future. Bulletin of the EATCS **65** (1998) 66–89
6. Beame, P., Kautz, H., Sabharwal, A.: Towards understanding and harnessing the potential of clause learning. Journal of Artificial Intelligence Research **22** (2004) 319–351
7. Gebser, M., Schaub, T.: Tableau calculi for answer set programming. In: ICLP. Volume 4079 of LNCS., Springer (2006) 11–25
8. Anger, C., Gebser, M., Janhunen, T., Schaub, T.: What's a head without a body? In: ECAI, IOS Press (2006) 769–770
9. Giunchiglia, E., Maratea, M.: On the relation between answer set and SAT procedures (or, between cmodels and smodels). In: ICLP. Volume 3668 of LNCS., Springer (2005) 37–51
10. Gebser, M., Schaub, T.: Characterizing ASP inferences by unit propagation. In: LaSh ICLP Workshop. (2006) 41–56
11. Tseitin, G.: On the complexity of derivation in propositional calculus. In: Automation of Reasoning 2: Classical Papers on Computational Logic. Springer (1983) 466–483
12. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: ICLP, MIT Press (1988) 1070–1080
13. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence **25**(3-4) (1999) 241–273
14. Lifschitz, V., Razborov, A.: Why are there so many loop formulas? ACM Transactions on Computational Logic **7**(2) (2006) 261–268
15. Ben-Eliyahu, R., Dechter, R.: Propositional semantics for disjunctive logic programs. Annals of Mathematics and Artificial Intelligence **12**(1-2) (1994) 53–87
16. Janhunen, T.: Some (in)translatability results for normal logic programs and propositional theories. Journal of Applied Non-Classical Logics **16**(1-2) (2006) 35–86
17. Lin, F., Zhao, Y.: ASSAT: Computing answer sets of a logic program by SAT solvers. Artificial Intelligence **157**(1–2) (2004) 115–137
18. Giunchiglia, E., Lierler, Y., Maratea, M.: Answer set programming based on propositional satisfiability. Journal of Automated Reasoning **36**(4) (2006) 345–377
19. Clark, K.: Negation as failure. In: Readings in nonmonotonic reasoning. Morgan Kaufmann Publishers (1987) 311–325
20. Fages, F.: Consistency of Clark's completion and existence of stable models. Journal of Methods of Logic in Computer Science **1** (1994) 51–60
21. Ben-Sasson, E., Impagliazzo, R., Wigderson, A.: Near optimal separation of tree-like and general resolution. Combinatorica **24**(4) (2004) 585–603
22. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem proving. Communications of the ACM **5**(7) (1962) 394–397
23. Cook, S.A.: A short proof of the pigeon hole principle using extended resolution. SIGACT News **8**(4) (1976) 28–32
24. Haken, A.: The intractability of resolution. TCS **39**(2-3) (1985) 297–308
25. Eiter, T., Fink, M., Tompits, H., Woltran, S.: Simplifying logic programs under uniform and strong equivalence. In: LPNMR. Volume 2923 of LNCS., Springer (2004) 87–99