Extended ASP Tableaux and Rule Redundancy in Normal Logic Programs∗

MATTI JÄRVISALO and EMILIA OIKARINEN
Helsinki University of Technology (TKK)
Department of Information and Computer Science
P.O. Box 5400, FI-02015 TKK, Finland
(*e-mail:* matti.jarvisalo@tkk.fi, emilia.oikarinen@tkk.fi)

submitted 20 February 2008; revised 12 September 2008; accepted 18 September 2008

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 sets of clauses, and separate Extended ASP Tableaux from ASP Tableaux by giving a polynomial-length proof for 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 lengths of proofs in ASP. Closely related to Extended ASP Tableaux, we empirically investigate the effect of redundant rules on the efficiency of ASP solving.

KEYWORDS: Answer set programming, tableau method, extension rule, proof complexity, problem structure

1 Introduction

Answer set programming (ASP) (Marek and Truszczyński 1999; Niemelä 1999; Gelfond and Leone 2002; Lifschitz 2002; Baral 2003) is a declarative problem solving paradigm which has proven successful for a variety of knowledge representation and reasoning tasks (see (Soininen et al. 2001; Nogueira et al. 2001; Erdem et al. 2006; Brooks et al. 2007) for examples). The success has been brought forth by efficient solver implementations such as smodels (Simons et al. 2002), dlv (Leone et al. 2006), noMore++ (Anger et al. 2005), cmodels (Giunchiglia et al. 2006), assat (Lin and Zhao 2004), and clasp (Gebser et al. 2007). However, there has been an evident lack of theoretical studies into the reasons for the efficiency of ASP solvers.

Solver implementations and their inference techniques can be seen as deterministic implementations of the underlying rule-based *proof systems*. A solver implements

∗ This is an extended version of a paper (Järvisalo and Oikarinen 2007) presented at the 23rd International Conference on Logic Programming (ICLP 2007) in Porto, Portugal.

a particular proof system in the sense that the propagation mechanisms applied by the solver apply the deterministic deduction rules in the proof system, whereas the nondeterministic branching/splitting rule of the proof system is made deterministic through branching heuristics present in typical solvers. From the opposite point of view, a solver can be analyzed by investigating the power of an abstraction of the solver as the proof system the solver implements. 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.

A way of examining the *best-case* performance of solver algorithms is provided by *(propositional) proof complexity theory* (Cook and Reckhow 1979; Beame and Pitassi 1998), which concentrates on studying the relative power of the proof systems underlying solver algorithms in terms of the shortest existing proofs in the systems. A large (superpolynomial) difference in the minimal length of proofs available in different proof systems for a family of Boolean expressions reveals that solver implementations of these systems are inherently different in strength. While such proof complexity theoretic 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 (Beame et al. 2004), this has not been the case for ASP. Especially, the inference techniques applied in current state-of-the-art ASP solvers have been characterized by a family of tableau-style ASP proof systems for normal logic programs only very recently (Gebser and Schaub 2006b), with some related proof complexity theoretic investigations (Anger et al. 2006) and generalizations (Gebser and Schaub 2007). The close relation of ASP and SAT and the respective theoretical underpinning of practical solver techniques has also received little attention up until recently (Giunchiglia and Maratea 2005; Gebser and Schaub 2006a), although the fields could gain much by further studies on these connections.

This work continues in part bridging the gap between ASP and SAT. Influenced by Tseitin's *Extended Resolution* proof system (Tseitin 1969) for clausal formulas, we introduce *Extended ASP Tableaux*, an extended tableau calculus based on the proof system in (Gebser and Schaub 2006b). 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, for example, 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 work 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, the 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* mechanisms 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 rest of this article is organized as follows. After preliminaries on ASP and SAT (Section 2), the relationship of Resolution and ASP Tableaux proof systems and concepts related to the complexity of proofs are discussed (Section 3). By introducing the Extended ASP Tableaux proof system (Section 4), proof complexity and simplification are then studied with respect to Extended ASP Tableaux (Section 5). Experimental results related to Extended ASP Tableaux and redundant rules in normal logic programs are presented in Section 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. In the following we will review some standard concepts related to NLPs and stable models.

A normal logic program Π consists of a finite set of rules of the form

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

where each a_i and b_j is a propositional atom, and h is either a propositional atom, or the symbol $\bot$ that stands for falsity. A rule r consists of a *head*, $\mathsf{head}(r) = h$, and a *body*, $\mathsf{body}(r) = \{a_1, \ldots, a_n, {\sim}b_1, \ldots, {\sim}b_m\}$. The symbol "$\sim$" denotes *default negation*. A *default literal* is an atom a, or its default negation ${\sim}a$.

The set of atoms occurring in a program Π is $\mathsf{atom}(\Pi)$, and

$$\mathsf{dlit}(\Pi) = \{a, {\sim}a \mid a \in \mathsf{atom}(\Pi)\}$$

is the set of default literals in Π. We use the shorthands $L^+ = \{a \mid a \in L\}$ and $L^- = \{a \mid {\sim}a \in L\}$ for a set L of default literals, and ${\sim}A = \{{\sim}a \mid a \in A\}$ for a set A of atoms. 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 $\mathsf{body}(r) = \emptyset$. Furthermore, we use the shorthands

$$\begin{aligned} \mathsf{head}(\Pi) &= \{\mathsf{head}(r) \mid r \in \Pi\} \text{ and} \\ \mathsf{body}(\Pi) &= \{\mathsf{body}(r) \mid r \in \Pi\}. \end{aligned}$$

In ASP, we are interested in *stable models* (Gelfond and Lifschitz 1988) (or *answer sets*) of a program Π. An *interpretation* $M \subseteq \mathsf{atom}(\Pi)$ defines which atoms of Π are true ($a \in M$) and which are false ($a \notin M$). An interpretation $M \subseteq \mathsf{atom}(\Pi)$ 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 of a program Π is a stable model of Π if and only if there is no model $M' \subset M$ of Π^M, where

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

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.

The *positive dependency graph* of Π, denoted by $\mathsf{Dep}^+(\Pi)$, is a directed graph with $\mathsf{atom}(\Pi)$ and

$$\{\langle b, a\rangle \mid \exists r \in \Pi \text{ such that } b = \mathsf{head}(r) \text{ and } a \in \mathsf{body}(r)^+\}$$

as the sets of vertices and edges, respectively. A non-empty set $L \subseteq \mathsf{atom}(\Pi)$ is a loop in $\mathsf{Dep}^+(\Pi)$ if for any $a, b \in L$ there is a path of non-zero length from a to b in $\mathsf{Dep}^+(\Pi)$ such that all vertices in the path are in L. We denote by $\mathsf{loop}(\Pi)$ the set of all loops in $\mathsf{Dep}^+(\Pi)$. A NLP is *tight* if and only if $\mathsf{loop}(\Pi) = \emptyset$. Furthermore, the *external bodies* of a set A of atoms 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 $U \subseteq \mathsf{atom}(\Pi)$ is *unfounded* if $\mathsf{eb}(U) = \emptyset$. We denote the *greatest unfounded set*, that is, the union of all unfounded sets, of Π by $\mathsf{gus}(\Pi)$.

A *splitting set* (Lifschitz and Turner 1994) for a NLP Π is any set $U \subseteq \mathsf{atom}(\Pi)$ such that for every $r \in \Pi$, if $\mathsf{head}(r) \in U$, then $\mathsf{body}(r)^+ \cup \mathsf{body}(r)^- \subseteq U$. The *bottom* of Π relative to U is

$$\mathsf{bottom}(\Pi, U) = \{r \in \Pi \mid \mathsf{atom}(\{r\}) \subseteq U\},$$

and the *top* of Π relative to U is

$$\mathsf{top}(\Pi, U) = \Pi \setminus \mathsf{bottom}(\Pi, U).$$

The top can be partially evaluated with respect to an interpretation $X \subseteq U$. The result is a program $\mathsf{eval}(\mathsf{top}(\Pi, U), X)$ that contains the rule

$$\mathsf{head}(r) \leftarrow (\mathsf{body}(r)^+ \setminus U), {\sim}(\mathsf{body}(r)^- \setminus U)$$

for each $r \in \mathsf{top}(\Pi, U)$ such that $\mathsf{body}(r)^+ \cap U \subseteq X$ and $(\mathsf{body}(r)^- \cap U) \cap X = \emptyset$. Given a splitting set U for a NLP Π, a *solution* to Π with respect to U is a pair $\langle X, Y\rangle$ such that $X \subseteq U$, $Y \subseteq \mathsf{atom}(\Pi) \setminus U$, X is a stable model of $\mathsf{bottom}(\Pi, U)$, and Y is a stable model of $\mathsf{eval}(\mathsf{top}(\Pi, U), X)$. In this work we will apply the *splitting set theorem* (Lifschitz and Turner 1994) that relates solutions with stable models.

Theorem 2.1 ((Lifschitz and Turner 1994)) *Given a normal logic program Π and a splitting set U for Π, an interpretation $M \subseteq \mathsf{atom}(\Pi)$ is a stable model of Π if and only if $\langle M \cap U, M \setminus U\rangle$ is a solution to Π with respect to U.*

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 set of variables appearing in a clause C (a set $\mathcal{C}$ of clauses, respectively) is denoted by $\mathsf{var}(C)$ ($\mathsf{var}(\mathcal{C})$, respectively).

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 and only if it satisfies every clause in it. A clause is satisfied if and only if it contains at least one satisfied literal, where a literal x ($\bar{x}$, respectively) is satisfied if $\tau(x) = \mathsf{true}$ ($\tau(x) = \mathsf{false}$, respectively). A set of clauses 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 set of clauses *faithfully*, that is, there is a bijective correspondence between the satisfying truth assignments and stable models of the translation (Niemelä 1999). Given a set $\mathcal{C}$ of clauses, 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 $x \in \mathsf{var}(\mathcal{C})$. The resulting NLP is then

$$
\begin{aligned}
\mathsf{nlp}(\mathcal{C}) \quad = \quad & \{a_x \leftarrow {\sim}\hat{a}_x.\ \ \hat{a}_x \leftarrow {\sim}a_x \mid x \in \mathsf{var}(\mathcal{C})\} \cup && (2)\\
& \{\bot \leftarrow {\sim}c \mid C \in \mathcal{C}\} \cup && (3)\\
& \{c \leftarrow a_x \mid x \in C,\ C \in \mathcal{C},\ x \in \mathsf{var}(C)\} \cup && (4)\\
& \{c \leftarrow {\sim}a_x \mid \bar{x} \in C,\ C \in \mathcal{C},\ x \in \mathsf{var}(C)\}. && (5)
\end{aligned}
$$

The rules (2) encode that each variable must be assigned an unambiguous truth value, the rules in (3) that each clause in $\mathcal{C}$ must be satisfied, while (4) and (5) encode that each clause is satisfied if at least one of its literals is satisfied.

Example 2.2 *The set* $\mathcal{C} = \{\{x, y\}, \{x, \bar{y}\}, \{\bar{x}, y\}, \{\bar{x}, \bar{y}\}\}$ *of clauses is represented by the normal logic program*

$$
\begin{aligned}
\mathsf{nlp}(\mathcal{C}) \quad = \quad \{ \ & a_x \leftarrow {\sim}\hat{a}_x.\ \hat{a}_x \leftarrow {\sim}a_x.\ a_y \leftarrow {\sim}\hat{a}_y.\ \hat{a}_y \leftarrow {\sim}a_y.\\
& \bot \leftarrow {\sim}c_1.\ \bot \leftarrow {\sim}c_2.\ \bot \leftarrow {\sim}c_3.\ \bot \leftarrow {\sim}c_4.\\
& c_1 \leftarrow a_x.\ c_1 \leftarrow a_y.\ c_2 \leftarrow a_x.\ c_2 \leftarrow {\sim}a_y.\\
& c_3 \leftarrow {\sim}a_x.\ c_3 \leftarrow a_y.\ c_4 \leftarrow {\sim}a_x.\ c_4 \leftarrow {\sim}a_y \ \}.
\end{aligned}
$$

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 (Niemelä 1999).

[1] Intuitively, for a modular translation, adding a set of facts to a program leads to a local change not involving the translation of the rest of the program (Niemelä 1999).

Moreover, any faithful translation is potentially of exponential size when additional variables are not allowed (Lifschitz and Razborov 2006)[2]. However, for any tight program Π it holds that the answer sets of Π can be characterized faithfully by the satisfying truth assignments of a linear-size propositional formula called *Clark's completion* (Clark 1978; Fages 1994) of Π, defined using a Boolean variable x_a for each $a \in \mathsf{atom}(\Pi)$ as

$$C(\Pi) \quad = \bigwedge_{h \in \mathsf{atom}(\Pi) \cup \{\bot\}} \left(x_h \leftrightarrow \bigvee_{r \in \mathsf{rule}(h)} \left(\bigwedge_{b \in \mathsf{body}(r)^+} x_b \wedge \bigwedge_{b \in \mathsf{body}(r)^-} \bar{x}_b \right) \right), \tag{6}$$

where $\mathsf{rule}(h) = \{r \in \Pi \mid \mathsf{head}(r) = h\}$. Notice that there are the special cases (i) if h is $\bot$ then the equivalence becomes the negation of the right hand side, (ii) if h is a fact, then the equivalence reduces to the clause $\{x_h\}$, and (iii) if an atom h does not appear in the head of any rule then the equivalence reduces to the clause $\{\bar{x}_h\}$.

In this work, we will consider the clausal representation of Boolean formulas. A linear-size clausal translation of $C(\Pi)$ is achieved by introducing additionally a new Boolean variable x_B for each $B \in \mathsf{body}(\Pi)$. Using the new variables for the bodies, we arrive at the *clausal completion*

$$\begin{aligned} \mathsf{comp}(\Pi) \quad = & \bigcup_{B \in \mathsf{body}(\Pi)} \left\{ x_B \equiv \bigwedge_{a \in B^+} x_a \wedge \bigwedge_{b \in B^-} \bar{x}_b \right\} \cup \bigcup_{B \in \mathsf{body}(\mathsf{rule}(\bot))} \{\{\bar{x}_B\}\} & (7) \\ & \cup \bigcup_{h \in \mathsf{head}(\Pi) \setminus \{\bot\}} \left\{ x_h \equiv \bigvee_{B \in \mathsf{body}(\mathsf{rule}(h))} x_B \right\} & (8) \\ & \cup \bigcup_{a \in \mathsf{atom}(\Pi) \setminus \mathsf{head}(\Pi)} \{\{\bar{x}_a\}\}, & (9) \end{aligned}$$

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 $\{x, \bar{x}_1, \ldots, \bar{x}_n\} \cup \bigcup_{x_i \in X} \{\bar{x}, x_i\}$ and $\bigcup_{x_i \in X} \{x, \bar{x}_i\} \cup \{\bar{x}, x_1, \ldots, x_n\}$, respectively.

Example 2.3 *For the normal logic program* $\Pi = \{a \leftarrow b, {\sim}a.\ b \leftarrow c.\ c \leftarrow {\sim}b\}$*, the clausal completion is*

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

[2] However, polynomial-size propositional encodings using extra variables are known, see (Ben-Eliyahu and Dechter 1994; Lin and Zhao 2003; Janhunen 2006). Also, ASP as Propositional Satisfiability approaches for solving normal logic programs have been developed, for example, assat (Lin and Zhao 2004) (based on incrementally adding—possibly exponentially many—loop formulas) and asp-sat (Giunchiglia et al. 2006) (based on generating a *supported model* (Brass and Dix 1995) of the program and testing its minimality—thus avoiding exponential space consumption).

3 Proof Systems for ASP and SAT

In this section we review concepts related to proof complexity (Cook and Reckhow 1979; Beame and Pitassi 1998) in the context of this work, and discuss the relationship of Resolution and ASP Tableaux (Gebser and Schaub 2006b).

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 $S(E, P)$ holds. 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 for 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 and 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'. If S simulates S', and there is a family $\{E_n\}$ separating S from S', then S is *more powerful* than S'.

3.2 Resolution

The well-known Resolution proof system (RES) for sets of clauses 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 set $\mathcal{C}$ of clauses 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* for (the unsatisfiability of) $\mathcal{C}$.

Any RES proof $\pi = (C_1, C_2, \ldots, C_n = \emptyset)$ can be represented as a directed acyclic graph, in which the leafs are initial clauses and other nodes are derived clauses. There are edges from C_i and C_j to C_k if and only if 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* ($\mathsf{T\text{-}RES}$), 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, that is, RES is more powerful than T-RES (Ben-Sasson et al. 2004). On the other hand, it is well known that the DPLL method (Davis and Putnam 1960; Davis et al. 1962), the basis of most state-of-the-art SAT solvers, is polynomially equivalent to T-RES. However, conflict-learning DPLL is more powerful than T-RES, and polynomially equivalent to RES under a slight generalization (Beame et al. 2004).

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 or ASP-T (Gebser and Schaub 2006b).

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 $a \in \mathsf{atom}(\Pi) \cup \mathsf{body}(\Pi)$. 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 Figure 1; 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 use shorthands $\mathbf{t}l$ and $\mathbf{f}l$ for default literals:

$$\begin{aligned}
\mathbf{t}l &= \begin{cases}\mathbf{T}a, & \text{if } l = a \text{ is positive,}\\ \mathbf{F}a, & \text{if } l = {\sim}a \text{ is negative; and}\end{cases}\\
\mathbf{f}l &= \begin{cases}\mathbf{T}a, & \text{if } l = {\sim}a \text{ is negative,}\\ \mathbf{F}a, & \text{if } l = a \text{ is positive.}\end{cases}
\end{aligned}$$

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 the entries $\mathbf{T}a$ and $\mathbf{F}a$ 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 $a \in \mathsf{atom}(\Pi) \cup \mathsf{body}(\Pi)$ and the branch is closed under the deduction rules (b)–(i). A tableau is contradictory, if all its branches in are contradictory, and *non-contradictory* otherwise. A tableau is complete if all its branches are complete. A contradictory tableau from Π is an ASP-T proof for (the unsatisfiability of) Π. The *length* of an ASP-T proof is the number of entries in it.

Example 3.1 *An* ASP-T *proof for the NLP* $\Pi = \{a \leftarrow b, {\sim}a.\ b \leftarrow c.\ c \leftarrow {\sim}b\}$ *is shown in Figure 2, with the rule applied for deducing each entry given in parentheses. For example, the entry* $\mathbf{F}a$ *has been deduced from* $a \leftarrow b, {\sim}a$ *in* Π *and the entry* $\mathbf{T}\{b, {\sim}a\}$ *in the left branch by applying the rule (g) Backward True Body. On the other hand,* $\mathbf{T}\{b, {\sim}a\}$ *has been deduced from* $a \leftarrow b, {\sim}a$ *in* Π *and the entry* $\mathbf{T}a$ *in the left branch by applying the rule (i§), that is, rule (i) by the fact that the condition § "Backward True Atom" is fulfilled (in* Π*, the only body with atom* a *in*

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

(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)

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

(i)

($\natural$): Applicable when $\phi \in \mathsf{atom}(\Pi) \cup \mathsf{body}(\Pi)$.
($\flat$): Applicable when one of the following conditions holds:
§ ("Forward False Atom"), † ("Well-Founded Negation"), or ‡ ("Forward Loop").
($\sharp$): Applicable when one of the following conditions holds:
§ ("Backward True Atom"), † ("Well-Founded Justification"), or ‡ ("Backward Loop").
(§): Applicable when $\mathsf{body}(\mathsf{rule}(h)) = \{B_1, \ldots, B_m\}$.
(†): Applicable when
$\{B_1, \ldots, B_m\} \subseteq \mathsf{body}(\Pi)$ and $h \in \mathsf{gus}(\{r \in \Pi \mid \mathsf{body}(r) \notin \{B_1, \ldots B_m\}\})$.
(‡): Applicable when $h \in L$, $L \in \mathsf{loop}(\Pi)$, and $\mathsf{eb}(L) = \{B_1, \ldots, B_m\}$ all hold.

Fig. 1. Rules in ASP Tableaux.

the head is $\{b, {\sim}a\}$*). The tableau in Figure 2 has two closed branches:*

$$(\Pi \cup \{\mathbf{F}\bot\}, \mathbf{T}a, \mathbf{T}\{b, {\sim}a\}, \mathbf{F}a) \text{ and}$$

$$(\Pi \cup \{\mathbf{F}\bot\}, \mathbf{F}a, \mathbf{F}\{b, {\sim}a\}, \mathbf{F}b, \mathbf{T}\{{\sim}b\}, \mathbf{T}c, \mathbf{T}\{c\}, \mathbf{T}b).$$

These branches share the common prefix $(\Pi \cup \{\mathbf{F}\bot\})$.

Any branch B describes a *partial assignment* $\mathcal{A}$ on $\mathsf{atom}(\Pi) \cup \mathsf{body}(\Pi)$ in a natural way, that is, if there is an entry $\mathbf{T}a$ ($\mathbf{F}a$, respectively) in B for $a \in \mathsf{atom}(\Pi) \cup \mathsf{body}(\Pi)$, 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, that is, there is a complete non-

$$
\begin{array}{c}
a \leftarrow b, \sim a \\
b \leftarrow c \\
c \leftarrow \sim b \\
\mathbf{F}\bot
\end{array}
$$

$$
\begin{array}{ll|ll}
\multicolumn{2}{c|}{\mathbf{T}a} & \multicolumn{2}{c}{\mathbf{F}a} \\
\mathbf{T}\{b, \sim a\} & \text{(i§)} & \mathbf{F}\{b, \sim a\} & \text{(e)} \\
\mathbf{F}a & \text{(g)} & \mathbf{F}b & \text{(c)} \\
\times & & \mathbf{T}\{\sim b\} & \text{(b)} \\
 & & \mathbf{T}c & \text{(d)} \\
 & & \mathbf{T}\{c\} & \text{(b)} \\
 & & \mathbf{T}b & \text{(d)} \\
 & & \times &
\end{array}
$$

Fig. 2. An ASP-T proof for $\Pi = \{a \leftarrow b, \sim a.\ b \leftarrow c.\ c \leftarrow \sim b\}$.

contradictory ASP tableau from Π if and only if Π is satisfiable (Gebser and Schaub 2006b). Thus the assignment $\mathcal{A}$ described by a complete non-contradictory branch gives a stable model $M = \{a \in \mathsf{atom}(\Pi) \mid (a, \mathsf{true}) \in \mathcal{A}\}$ of Π.

As argued in (Gebser and Schaub 2006b), current ASP solver implementations are tightly related to ASP-T, with the intuition that the cut rule is made deterministic with decision heuristics, while the deduction rules describe the propagation mechanism in ASP solvers. For instance, the noMore++ system (Anger et al. 2005) is a deterministic implementation of the rules (a)–(g),(h§),(h†), and (i§), while smodels (Simons et al. 2002) applies the same rules with the cut rule restricted to $\mathsf{atom}(\Pi)$.

Interestingly, ASP-T and T-RES are polynomially equivalent under the translations comp and nlp. Although the similarity of unit propagation in DPLL and propagation in ASP solvers is discussed in (Giunchiglia and Maratea 2005; Gebser and Schaub 2006a), here we want to stress the direct connection between ASP-T and T-RES. In detail, T-RES and ASP-T are equivalent in the sense that (i) given an arbitrary NLP Π, the length of minimal T-RES proofs for $\mathsf{comp}(\Pi)$ is polynomially bounded in the the length of minimal ASP-T proofs for Π, and (ii) given an arbitrary set $\mathcal{C}$ of clauses, the length of minimal ASP-T proofs for $\mathsf{nlp}(\mathcal{C})$ is polynomially bounded in the length of minimal T-RES proofs for $\mathcal{C}$.

Theorem 3.2 T-RES *and* ASP-T *are polynomially equivalent proof systems in the sense that*

(i) *considering tight normal logic programs,* T-RES *under the translation* comp *polynomially simulates* ASP-T*, and*

(ii) *considering sets of clauses,* ASP-T *under the translation* nlp *polynomially simulates* T-RES*.*

In the following we give detailed proofs for the two parts of Theorem 3.2 followed by illustrating examples.

In the proof of the first part of Theorem 3.2, we use a concept of a *(binary) cut tree* corresponding to an ASP-T proof. Given an ASP-T proof T for a normal logic program Π, the corresponding cut tree is obtained as follows. Starting from the root of T, we replace each non-leaf entry generated by a deduction rule in T by an application of the cut rule on the corresponding entry. For example, the cut tree T' corresponding to the ASP-T proof T in Figure 2 is given in Figure 3 (left).

Proof of Theorem 3.2 (i)
Let T be an ASP-T proof for a tight normal logic program Π. Without loss of generality, we will assume that branches in T have not been extended further after they have become contradictory. We now show that we can construct a T-RES proof π for $\mathsf{comp}(\Pi)$ using the cut tree T' corresponding to T. Furthermore, we show that for such a proof π it holds that, given any prefix p of an arbitrary branch B in T' there is a clause $C \in \pi$ contradictory to the partial assignment in p, that is, there is the entry $\mathbf{F}a$ ($\mathbf{T}a$) for $a \in \mathsf{atom}(\Pi) \cup \mathsf{body}(\Pi)$ in p for each corresponding positive literal x_a (negative literal $\bar{x}_a$) in C.

Consider first the partial assignment in an arbitrary (full) branch B in T'. Assume that there is no clause in $\mathsf{comp}(\Pi)$ contradictory to the partial assignment in B, that is, we can obtain a truth assignment τ based on the entries in B such that every clause in $\mathsf{comp}(\Pi)$ is satisfied in τ. But this leads to contradiction since $\mathsf{comp}(\Pi)$ is satisfied if and only if Π is satisfied. Thus there is a clause $C \in \mathsf{comp}(\Pi)$ contradictory to the partial assignment in B, and we take the clause C into our resolution proof π.

Assume that we have constructed π such that for any prefix p of length n for any branch B in T', there is a clause $C \in \pi$ contradictory to the partial assignment in p. Consider an arbitrary prefix p of length $n-1$. Now, in T' we have the prefixes p' and p'' of length n which have been obtained through extending p by applying the cut rule on some $a \in \mathsf{atom}(\Pi) \cup \mathsf{body}(\Pi)$. In other words, p' is p with $\mathbf{T}a$ appended in the end (p'' is p with $\mathbf{F}a$ appended in the end). Since p' (p'', respectively) is of length n, there is a clause C (D, respectively) in π contradictory to the partial assignment in p' (p'', respectively). Now there are two possibilities. If $C = \{\bar{x}_a\} \cup C'$ and $D = \{x_a\} \cup D'$, we can resolve on x_a adding $C' \cup D'$ to π. Thus we have a clause $C' \cup D' \in \pi$ contradictory to the partial assignment in the prefix p. Otherwise we have that $\bar{x}_a \notin C$ or $x_a \notin D$, and hence either $C \in \pi$ or $D \in \pi$ is contradictory to the partial assignment in the prefix p.

When reaching the root of T', we must have derived $\emptyset$ since it is the only clause contradictory with the empty assignment. Furthermore, the T-RES derivation π is of polynomial length with respect to T' (and T). $\square$

The following example illustrates the RES proof construction used above in the proof of Theorem 3.2 (i).

Example 3.3 *Consider again the tight NLP* $\Pi = \{a \leftarrow b, {\sim}a.\ b \leftarrow c.\ c \leftarrow {\sim}b\}$ *from Example 2.3 and the* ASP-T *proof* T *for* Π *in Figure 2. We now construct a* T-RES *proof for the completion* $\mathsf{comp}(\Pi)$ *(see Example 2.3 for details) using the strategy from the proof of Theorem 3.2 (i). First,* T *is transformed into a cut tree* T' *given in Figure 3 (left). Consider now the two leftmost branches in* T'*. The partial assignment in the branch with entries* $\mathbf{T}a$ *and* $\mathbf{F}\{b, {\sim}a\}$ *is contradictory*

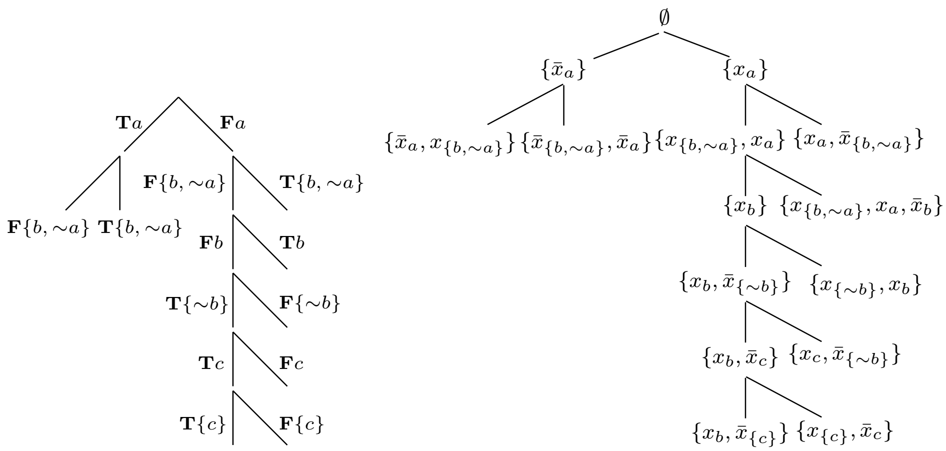


Fig. 3. Left: cut tree based on the ASP-T proof in Figure 2. Right: resulting T-RES proof.

to clause $\{\bar{x}_a, x_{\{b,\sim a\}}\}$ *in* $\mathsf{comp}(\Pi)$*, and the partial assignment in the branch with entries* $\mathbf{T}a$ *and* $\mathbf{T}\{b, \sim a\}$ *is contradictory to clause* $\{\bar{x}_{\{b,\sim a\}}, \bar{x}_a\}$ *in* $\mathsf{comp}(\Pi)$*. Thus we resolve on* $x_{\{b,\sim a\}}$ *and obtain the clause* $\{\bar{x}_a\}$*, which is contradictory to the single entry* $\mathbf{T}a$ *in the prefix of the two leftmost branches in* T'*. Similarly, we can construct a resolution tree for clause* $\{x_a\}$ *corresponding to the right side of* T'*. We finish the proof by resolving on* x_a*. The complete* T-RES *proof corresponding to the cut tree* T' *is shown in Figure 3 (right).*

Proof of Theorem 3.2 (ii)
Let $\pi = (C_1, \ldots, C_n = \emptyset)$ be a T-RES refutation of a set $\mathcal{C}$ of clauses. Recall that each derived clause C_i in π is obtained by resolving on x from $C_j = C \cup \{x\}$ and $C_k = D \cup \{\bar{x}\}$ for some $j, k < i$.

An ASP-T proof T for $\mathsf{nlp}(\mathcal{C})$ is obtained from π as follows. We start from C_n, which is obtained from clauses $C_j = \{x\}$ and $C_k = \{\bar{x}\}$ by resolving on $x \in \mathsf{var}(\mathcal{C})$, and apply in T the cut rule on a_x corresponding to x. Then we recursively continue the same way with C_j (C_k, respectively) in the generated branch with $\mathbf{F}a_x$ ($\mathbf{T}a_x$, respectively). Since π is tree-like, each clause in the prefix $(C_1, \ldots, C_{\max\{j,k\}})$ of π is either used in the derivation of C_j or C_k, but not in both. By construction when reaching C_1 the branches of T correspond one-to-one to the paths in π (seen as a tree) from C_n to the leaf clauses of π. For a particular leaf clause C, we have for each literal $l \in C$ ($l = x$ or $l = \bar{x}$) contradicting entries for a_x in the corresponding branch of T, that is, $\mathbf{F}a_x$ if $l = x$ and $\mathbf{T}a_x$ if $l = \bar{x}$. Now we can directly deduce for each $\mathbf{F}a_x$ the entry $\mathbf{F}\{a_x\}$ and for each $\mathbf{T}a_x$ the entry $\mathbf{F}\{\sim a_x\}$. These entries together will allow us to directly deduce $\mathbf{F}c$ (all the bodies of rules with atom c as the head are false). Since we have $\bot \leftarrow \sim c \in \mathsf{nlp}(\mathcal{C})$, we can deduce $\mathbf{T}c$, and the branch becomes contradictory. $\square$

The following example illustrates the strategy used in the proof of Theorem 3.2 (ii).

Example 3.4 *Recall the set $\mathcal{C} = \{\{x,y\},\{x,\bar{y}\},\{\bar{x},y\},\{\bar{x},\bar{y}\}\}$ of clauses and the corresponding normal logic program $\mathsf{nlp}(\mathcal{C})$ presented in Example 2.2. The set $\mathcal{C}$ of clauses has a* T-RES *refutation $\pi = (\{x,y\},\{x,\bar{y}\},\{\bar{x},y\},\{\bar{x},\bar{y}\},\{y\},\{\bar{y}\},\emptyset)$. Now we construct an* ASP-T *proof T for $\mathsf{nlp}(\mathcal{C})$ from π as done in the proof of Theorem 3.2 (ii). The resulting* ASP-T *proof T is presented in Figure 4. In the tableau we have omitted entries of the form $\mathbf{T}\{l\}$ and $\mathbf{F}\{l\}$ for bodies consisting of a single default literal. The empty clause is obtained resolving on y from $\{y\}$ and $\{\bar{y}\}$, and thus we start with applying the cut rule on a_y. The clause $\{\bar{y}\}$ is obtained resolving on x from $\{x,\bar{y}\}$ and $\{\bar{x},\bar{y}\}$. We continue in the branch with $\mathbf{T}a_y$ by applying the cut rule on a_x. Consider now the branch with $\mathbf{T}a_y$ and $\mathbf{T}a_x$ in the tableau. The branch corresponds to the clause $\{\bar{x},\bar{y}\}$ in $\mathcal{C}$. Thus we arrive in a contradiction by deducing $\mathbf{F}c_4$ from $c_4 \leftarrow {\sim}a_x$ and $c_4 \leftarrow {\sim}a_y$, and $\mathbf{T}c_4$ from $\bot \leftarrow {\sim}c_4$. Other branches become contradictory similarly.*

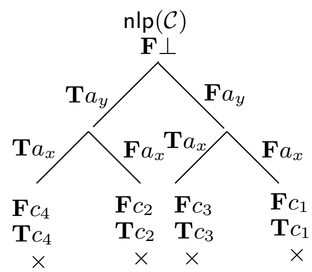


Fig. 4. An ASP-T proof for $\mathsf{nlp}(\mathcal{C})$ resulting from a T-RES proof $\pi = (\{x,y\},\{x,\bar{y}\},\{\bar{x},y\},\{\bar{x},\bar{y}\},\{y\},\{\bar{y}\},\emptyset)$ for $\mathcal{C}$ in Example 3.4.

4 Extended ASP Tableaux

We will now introduce an *extension rule*[3] 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.

Definition 4.1 *Given a normal logic program Π and two literals $l_1, l_2 \in \mathsf{dlit}(\Pi)$, the (elementary)* extension rule *in* E-ASP-T *adds the rule $p \leftarrow l_1, l_2$ to Π, where $p \notin \mathsf{atom}(\Pi) \cup \{\bot\}$.*

It is essential that p is a new atom for preserving satisfiability. After an application of the extension rule one considers the program $\Pi' = \Pi \cup \{p \leftarrow l_1, l_2\}$ instead of the original program Π. Notice that $\mathsf{atom}(\Pi') = \mathsf{atom}(\Pi) \cup \{p\}$. Thus when the extension rule is applied several times, the atoms introduced in previous applications of the rule can be used in defining further new atoms (see the forthcoming Example 4.2).

[3] Notice that the extension rule introduced here differs from the one proposed in (Hai et al. 2003) in the context of theorem proving.

When convenient, we will apply a generalization of the elementary extension rule. By allowing one to introduce multiple bodies for p, the *general extension rule* adds a set of rules

$$\bigcup_i \{p \leftarrow l_{i,1}, \ldots, l_{i,k_i} \mid p \notin \mathsf{atom}(\Pi) \cup \{\bot\} \text{ and } l_{i,k} \in \mathsf{dlit}(\Pi) \text{ for all } 1 \leq k \leq k_i\}$$

into Π. Notice that equivalent constructs can be introduced with the elementary extension rule. For example, bodies with more than two literals can be decomposed with balanced parentheses using additional new atoms.

Example 4.2 *Consider a normal logic program* Π *such that* $\mathsf{atom}(\Pi) = \{a, b\}$. *We apply the general extension rule and add a definition for the disjunction of atoms* a *and* b*, resulting in a program* $\Pi \cup \{c \leftarrow a.\ c \leftarrow b\}$. *An equivalent construct can be introduced by applying the elementary extension rule twice: add first a rule* $d \leftarrow {\sim}a, {\sim}b$*, and then add a rule* $c \leftarrow {\sim}d, {\sim}d$.

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

A key point is that applications of the extension rule do not affect the existence of stable models.

Theorem 4.3 *Extended ASP Tableaux is a sound and complete proof system for normal logic programs.*

Proof

Let T be an $\mathsf{E\text{-}ASP\text{-}T}$ proof for normal logic program Π with the set E of extending rules, that is, an $\mathsf{ASP\text{-}T}$ proof for $\Pi \cup E$. Since $\mathsf{ASP\text{-}T}$ is sound and complete, there is a complete non-contradictory branch in T if and only if $\Pi \cup E$ is satisfiable. The set $\mathsf{atom}(\Pi)$ is a splitting set for $\Pi \cup E$, since $\mathsf{head}(r) \notin \mathsf{atom}(\Pi) \cup \{\bot\}$ for every extending rule $r \in E$. Furthermore, $\mathsf{bottom}(\Pi \cup E, \mathsf{atom}(\Pi)) = \Pi$ and $\mathsf{top}(\Pi \cup E, \mathsf{atom}(\Pi)) = E$. By Theorem 2.1, $\Pi \cup E$ is satisfiable if and only if there is a solution to $\Pi \cup E$ with respect to $\mathsf{atom}(\Pi)$, that is, there is a stable model $M \subseteq \mathsf{atom}(\Pi)$ for Π and a stable model N for $\mathsf{eval}(E, M)$. Since the rules in E are generated using the extension rule (recall also $\bot \notin \mathsf{head}(E)$), there is a unique stable model for $\mathsf{eval}(E, M)$ for each $M \subseteq \mathsf{atom}(\Pi)$. Thus there is a solution to $\Pi \cup E$ with respect to $\mathsf{atom}(\Pi)$ if and only if Π is satisfiable, and moreover, $\Pi \cup E$ is satisfiable if and only if Π is satisfiable, and $\mathsf{E\text{-}ASP\text{-}T}$ is sound and complete. □

4.1 The Extension Rule and Well-Founded Deduction

An interesting question regarding the possible gains of applying the extension rule in $\mathsf{E\text{-}ASP\text{-}T}$ with the ASP tableau rules is whether the additional extension rule allows one to simulate well-founded deduction (rules (h†),(h‡),(i†), and (i‡)) with the other deduction rules ((b)–(g),(h§),(i§))[4]. We now show that this is not the case; the extension rule does not allow us to simulate reasoning related to unfounded sets and loops. This is implied by Theorem 4.4, which states that, by removing rules (h†),(h‡),(i†), and (i‡) from E-ASP-T, the resulting tableau method becomes incomplete for NLPs.

Theorem 4.4 *Using only tableau rules (a)–(g), (h§) and (i§), and the extension rule does not result in a complete proof system for normal logic programs.*

Proof

Consider the NLP $\Pi = \{\bot \leftarrow {\sim}a.\ a \leftarrow b.\ b \leftarrow a\}$. Although Π is unsatisfiable, in the proof system having only the tableau rules (a)–(g),(h§), and (i§), we can construct a complete and *non-contradictory* tableau with a single branch

$$T = (\Pi \cup \{\mathbf{F}\bot\}, \mathbf{F}\{{\sim}a\}\ (\mathrm{e}), \mathbf{T}a\ (\mathrm{c}), \mathbf{T}\{b\}\ (\mathrm{i}\S), \mathbf{T}b\ (\mathrm{g}), \mathbf{T}\{a\}\ (\mathrm{i}\S))$$

for Π.

Consider an arbitrary set E of extending rules generated using the extension rule in E-ASP-T. Recall that $\mathsf{head}(E) \cap (\mathsf{atom}(\Pi) \cup \{\bot\}) = \emptyset$. We can form a complete non-contradictory tableau T' for $\Pi \cup E$ as follows.

First, define $H_0 = \mathsf{atom}(\Pi) \cup \{\bot\}$ and

$$H_i = \{h \in \mathsf{head}(E) \mid \bigcup_{r \in \mathsf{rule}(h)} (\mathsf{body}(r)^+ \cup \mathsf{body}(r)^-) \subseteq \bigcup_{j<i} H_j\}.$$

Thus the sets H_i are used to define a level numbering for the atoms defined in the extension E. Furthermore, we define

$$E_i = \{r \in \Pi \cup E \mid \mathsf{head}(r) \in \bigcup_{j \le i} H_j\}$$

for all $i \ge 0$. Notice that $E_0 = \Pi$, and $\Pi \cup E = \bigcup_{i \ge 0} E_i$. We now show using induction that for each $i \ge 0$, the only branch B in T can be extended into a complete non-contradictory branch for E_i using tableau rules (b)–(g), (h§), and (i§).

The base case ($i = 0$) holds by definition. Assume that the claim holds for $i - 1$, that is, B can be extended into a complete non-contradictory branch B' for E_{i-1}. Consider now arbitrary $r \in E_i$. By definition $\mathsf{body}(r)^+ \cup \mathsf{body}(r)^- \subseteq \mathsf{atom}(E_{i-1})$ for each $r \in E_i$. Since B' is complete, it contains entries for each $a \in \mathsf{atom}(E_{i-1})$, and we can deduce an entry for $\mathsf{body}(r)$ using ASP tableau rule (b) or (f) (depending on the entries in B'). If the entry $\mathbf{T}(\mathsf{body}(r))$ has been deduced, we can deduce $\mathbf{T}h$ for $h = \mathsf{head}(r)$ using (d). Otherwise, we have deduced the entries $\mathbf{F}(\mathsf{body}(r'))$ for every $r' \in E_i$ such that $h = \mathsf{head}(r')$, and we can deduce $\mathbf{F}h$ using (h§). Thus we have deduced entries for all $a \in \mathsf{atom}(E_i) \cup \mathsf{body}(E_i)$ and the branch is non-contradictory. Furthermore it is easy to check that the branch is closed under the tableau rules (b)–(g),(h§), and (i§).

[4] Notice that the proof system consisting of tableau rules (a)–(g),(h§), and (i§) amounts to computing supported models (Gebser and Schaub 2006b).

Thus we obtain a complete and non-contradictory tableau for $\Pi \cup E$. Since we cannot generate a contradictory tableau for Π with tableau rules (a)–(g),(h§), and (i§), we cannot generate one for $\Pi \cup E$ either. This is in contradiction with the fact that Π is unsatisfiable. $\square$

5 Proof Complexity

In this section we study proof complexity theoretic issues related to E-ASP-T from several viewpoints: we will

- consider the relationship between E-ASP-T and the Extended Resolution proof system (Tseitin 1969),
- give an explicit separation of E-ASP-T from ASP-T, and
- relate the extension rule to the effect of program simplification on proof lengths in ASP-T.

5.1 Relationship with Extended Resolution

The system E-ASP-T is motivated by Extended Resolution (E-RES), a proof system originally introduced in (Tseitin 1969). The system E-RES consists of the resolution rule and an extension rule that allows one to expand a set of clauses by iteratively introducing equivalences of the form $x \equiv l_1 \wedge l_2$, where x is a new variable, and l_1 and l_2 are literals in the current set of clauses. In other words, given a set $\mathcal{C}$ of clauses, one application of the extension rule adds the clauses $\{x, \bar{l}_1, \bar{l}_2\}$, $\{\bar{x}, l_1\}$, and $\{\bar{x}, l_2\}$ to $\mathcal{C}$. The system E-RES is known to be more powerful than RES; in fact, E-RES is polynomially equivalent to, for example, 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 5.1 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 sets of clauses,* 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, that is, T is an ASP-T proof for $\Pi \cup E$, where E is the set of extending rules generated in the proof. We use the shorthand x_l for the variable corresponding to default literal l in comp($\Pi \cup E$), that is, $x_l = x_a$ ($x_l = \bar{x}_a$, respectively) if $l = a$ ($l = {\sim}a$, respectively) for $a \in$ atom($\Pi \cup E$). By Theorem 3.2 there is a polynomial RES proof for comp($\Pi \cup E$). Now consider comp(Π). We apply the extension rule in E-RES in the same order in which the extension rule in E-ASP-T is applied when generating the set E of

extending rules. In other words, we apply the extension rule in E-RES as follows for each rule $r = h \leftarrow l_1, l_2$ in E. If $\mathsf{body}(r) = \{l_1, l_2\} \in \mathsf{body}(\Pi)$, then there are the clauses $x_{\{l_1,l_2\}} \equiv x_{l_1} \wedge x_{l_2}$ in $\mathsf{comp}(\Pi)$. If this is the case, we generate the clauses $x_h \equiv x_{\{l_1,l_2\}}$ with the extension rule in E-RES. Otherwise, that is, if $\mathsf{body}(r)$ does not have a corresponding propositional variable in $\mathsf{comp}(\Pi)$, we generate the clauses $x_h \equiv x_{\{l_1,l_2\}}$ and $x_{\{l_1,l_2\}} \equiv x_{l_1} \wedge x_{l_2}$. Denote the resulting set of extending clauses by E'. Now we notice that $\mathsf{comp}(\Pi) \cup E' = \mathsf{comp}(\Pi \cup E)$, and therefore the RES proof for $\mathsf{comp}(\Pi \cup E)$ is an E-RES proof for $\mathsf{comp}(\Pi)$ in which the extension rule in E-RES is applied to generate the clauses in E'.

(ii): Let $\pi = (C_1, \dots, C_n = \emptyset)$ be an E-RES proof for a set $\mathcal{C}$ of clauses. Let E be the set of clauses in π generated with the extension rule. We introduce shorthands for atoms corresponding to literals, that is, $a_l = a_x$ ($a_l = {\sim} a_x$) if $l = x$ ($l = \bar{x}$) for $x \in \mathsf{var}(\mathcal{C} \cup E)$. Now, an E-ASP-T proof for $\mathsf{nlp}(\mathcal{C})$ is generated as follows. First, we add the following rules to $\mathsf{nlp}(\mathcal{C})$ with the extension rule in E-ASP-T:

$$a_x \leftarrow a_{l_1}, a_{l_2} \text{ for each extension } x \equiv l_1 \wedge l_2; \tag{10}$$

$$c \leftarrow a_l \text{ for each literal } l \in C \text{ for a clause } C \in \pi \text{ such that } C \notin \mathcal{C}; \text{ and} \tag{11}$$

$$p_1 \leftarrow c_1 \text{ and } p_i \leftarrow c_i, p_{i-1} \text{ for each } C_i \in \pi \text{ and } 2 \leq i < n. \tag{12}$$

Then, 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 now show that for each i the branch with $\mathbf{F}p_i$ and $\mathbf{T}p_j$ for all $j < i$ becomes contradictory without further application of the cut rule. First, deduce $\mathbf{F}c_i$ from $\mathbf{F}p_i$ using the rule (12) for i. One of the following holds for $C_i \in \pi$: either (a) $C_i \in \mathcal{C}$, (b) C_i is a derived clause, or (c) $C_i \in E$.

(a) If $C_i \in \mathcal{C}$ we can deduce $\mathbf{T}c_i$ from $\perp \leftarrow {\sim} c_i \in \mathsf{nlp}(\mathcal{C})$, and the branch becomes contradictory.

(b) If C_i is a derived clause, that is, C_i is obtained from C_j and C_k for $j, k < i$ resolving on x, then $C_i = (C_k \cup C_j) \setminus \{x, \bar{x}\}$. For all the literals $l \in C_i$ we deduce $\mathbf{f}a_l$ from the rules (11) in the extension. From $\mathbf{T}p_j$ and $\mathbf{T}p_k$ we deduce $\mathbf{T}c_j$ and $\mathbf{T}c_k$ using the rule (12) in the extension for j and k, respectively. Furthermore because we have entries $\mathbf{f}a_l$ for each l in $(C_k \cup C_j) \setminus \{x, \bar{x}\}$, we deduce $\mathbf{T}a_x$ and $\mathbf{F}a_x$ and the branch becomes contradictory. Recall that there is a rule $c \leftarrow a_l$ for each clause $C \in \pi$ and literal $l \in C$ either in $\mathsf{nlp}(\mathcal{C})$ or in the extension (rules in (11)).

(c) If $C_i \in E$, then C_i is of the form $\{x, \bar{l}_1, \bar{l}_2\}$, $\{\bar{x}, l_1\}$, or $\{\bar{x}, l_2\}$ for $x \equiv l_1 \wedge l_2$. For instance, if $C_i = \{\bar{x}, l_1\}$, 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 becomes contradictory as $\mathbf{T}\{a_{l_1}, a_{l_2}\}$ and $\mathbf{t}a_{l_1}$ are deduced from a rule (10) in the extension. The branch becomes contradictory similarly, if C_i is of the form $\{x, \bar{l}_1, \bar{l}_2\}$ or $\{\bar{x}, l_2\}$.

Finally, consider the branch with $\mathbf{T}p_i$ for all $i = 1 \dots n-1$. The empty clause C_n in π is obtained by resolving $C_j = \{x\}$ and $C_k = \{\bar{x}\}$ in π for some $j, k < n$. Thus we can deduce $\mathbf{T}c_j$ and $\mathbf{T}c_k$ from rules (12) for j and k, respectively, and furthermore, $\mathbf{T}a_x$ and $\mathbf{F}a_x$ from $c_j \leftarrow a_x$ and $c_k \leftarrow {\sim} a_x$, resulting in a contradiction in the branch. The obtained contradictory ASP tableau is of linear length with respect to π. $\square$

5.2 Pigeonhole Principle Separates Extended ASP Tableaux from ASP Tableaux

To exemplify the strength of E-ASP-T, we now consider a family of normal logic programs $\{\Pi_n\}$ which separates E-ASP-T from ASP-T, that is, we give an explicit polynomial-length proof for Π_n for which ASP-T has exponential-length minimal proofs with respect to n. We will consider this family also in the experiments reported in this article.

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

$$\begin{aligned}
\mathrm{PHP}_n^{n+1} \quad = \quad & \{\bot \leftarrow {\sim}p_{i,1}, \ldots, {\sim}p_{i,n} \mid 1 \le i \le n+1\} \;\cup & (13)\\
& \{\bot \leftarrow p_{i,k}, p_{j,k} \mid 1 \le i < j \le n+1,\; 1 \le k \le n\} \;\cup & (14)\\
& \{p_{i,j} \leftarrow {\sim}p'_{i,j}.\;\; p'_{i,j} \leftarrow {\sim}p_{i,j} \mid 1 \le i \le n+1,\; 1 \le j \le n\}. & (15)
\end{aligned}$$

In the program above, $p_{i,j}$ has the interpretation that pigeon i sits in hole j. The rules in (13) require that each pigeon must sit in some hole, and the rules in (14) require that no two pigeons can sit in the same hole. The rules in (15) 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 5.2 *The complexity of* $\{\mathrm{PHP}_n^{n+1}\}$ *with respect to* n *is*

(i) *polynomial in* E-ASP-T*, and*
(ii) *exponential in* ASP-T.

Proof
(i): In (Cook 1976) an extending set of clauses is added to a clausal encoding $\mathcal{C}_{\mathrm{PHP}}$ of the pigeonhole principle[5] so that RES has polynomial-length proofs for the resulting set of clauses. By Theorem 5.1 (ii) there is a polynomial-length E-ASP-T proof for

$$\begin{aligned}
\mathsf{nlp}(\mathcal{C}_{\mathrm{PHP}}) \quad = \quad & \{p_{i,j} \leftarrow {\sim}p'_{i,j}.\;\; p'_{i,j} \leftarrow {\sim}p_{i,j} \mid 1 \le i \le n+1,\; 1 \le j \le n\} \cup \\
& \{\bot \leftarrow {\sim}c_i \mid 1 \le i \le n+1\} \cup \\
& \{\bot \leftarrow {\sim}c_{ijk} \mid 1 \le i < j \le n+1, 1 \le k \le n\} \cup \\
& \{c_i \leftarrow p_{i,j} \mid 1 \le j \le n, 1 \le i \le n+1\} \cup \\
& \{c_{ijk} \leftarrow {\sim}p_{i,k}.\;\; c_{ijk} \leftarrow {\sim}p_{j,k} \mid 1 \le i < j \le n+1, 1 \le k \le n\}.
\end{aligned}$$

For simplicity, we keep the names of the atoms $p_{i,j}$ unchanged in the translation.

In more detail, let $\pi = (C_1, C_2, \ldots, C_m = \emptyset)$ be the polynomial-length E-RES

[5] The particular encoding, for which there are no polynomial-length RES proofs (Haken 1985), is $\mathcal{C}_{\mathrm{PHP}} = \bigcup_{1\le i\le n+1}\{\{\bigvee_{j=1}^{n} p_{i,j}\}\} \cup \bigcup_{1\le i<j\le n+1, 1\le k\le n}\{\{\neg p_{i,k} \vee \neg p_{j,k}\}\}$.

proof[6] for the clausal representation $\mathcal{C}_{\text{PHP}}$. Let

$$\text{EXT}^l \quad = \quad \{e^l_{i,j} \leftarrow e^{l+1}_{i,j}.\ e^l_{i,j} \leftarrow e^{l+1}_{i,l}, e^{l+1}_{l+1,j} \mid 1 \leq i \leq l \text{ and } 1 \leq j \leq l-1\}$$

for $1 < l \leq n$, where each $e^{n+1}_{i,j}$ is $p_{i,j}$. The extension EXT^l corresponds the set of extending clauses in (Cook 1976) similarly to the set of rules (10) in part (ii) of the proof of Theorem 5.1. Furthermore, $\text{E}(\pi)$ consists of the sets of rules (11) and (12) defined in the proof of Theorem 5.1 (ii). By applying the strategy from the proof of Theorem 5.1 (ii), we obtain a polynomial-length ASP-T proof for

$$\mathsf{nlp}(\mathcal{C}_{\text{PHP}}) \cup \bigcup_{1<l\leq n} \text{EXT}^l \cup \text{E}(\pi).$$

Now, we use the same strategy to construct a polynomial ASP-T proof for the program

$$\text{EPHP}^{n+1}_n = \text{PHP}^{n+1}_n \cup \bigcup_{1<l\leq n} \text{EXT}^l \cup \text{E}'(\pi),$$

where $\text{E}'(\pi)$ consists of rules $c \leftarrow a_l$ for each literal $l \in C$ for each clause $C \in \pi$ (that is, rules as in (11) but without the restriction $C \notin \mathcal{C}_{\text{PHP}}$) together with the rules in (12). The only difference comes in step (a) in the proof of Theorem 5.1 (ii), that is, when we have deduced $\mathbf{F}c$ corresponding to $C \in \mathcal{C}_{\text{PHP}}$. Since we do not have the rule $\perp \leftarrow {\sim}c$ in EPHP^{n+1}_n, we cannot deduce $\mathbf{T}c$ to obtain a contradiction. Instead, we can deduce a contradiction without using the ASP-T cut rule through a program rule in PHP^{n+1}_n that corresponds to the clause C. For instance, if $C = \{\neg p_{i,k}, \neg p_{j,k}\}$, we have the rules $c \leftarrow {\sim}p_{i,k}$ and $c \leftarrow {\sim}p_{j,k}$ in $\text{E}'(\pi)$ and the rule $\perp \leftarrow p_{i,k}, p_{j,k}$ in PHP^{n+1}_n. From $\mathbf{F}c$, we deduce $\mathbf{T}p_{i,k}$ and $\mathbf{T}p_{j,k}$. From $\mathbf{F}\perp$ and $\perp \leftarrow p_{i,k}, p_{j,k}$, we deduce $\mathbf{F}\{p_{i,k}, p_{j,k}\}$, and furthermore, from $\mathbf{T}p_{i,k}$ and $\mathbf{F}\{p_{i,k}, p_{j,k}\}$, we deduce $\mathbf{F}p_{j,k}$. This results in a polynomial-length E-ASP-T proof for PHP^{n+1}_n.

(ii): Assume now that there is a polynomial ASP-T proof for PHP^{n+1}_n. By Theorem 3.2, there is a polynomial T-RES proof for $\mathsf{comp}(\text{PHP}^{n+1}_n)$. Notice that the completion $\mathsf{comp}(\text{PHP}^{n+1}_n)$ consists of the clausal encoding $\mathcal{C}_{\text{PHP}}$ of the pigeonhole principle and additional clauses (tautologies) for rules of the form $p_{i,j} \leftarrow {\sim}p'_{i,j}$, $p'_{i,j} \leftarrow {\sim}p_{i,j}$. It is easy to see that these additional tautologies do not affect the length of the minimal T-RES proofs for $\mathsf{comp}(\text{PHP}^{n+1}_n)$. 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 with respect to n for (Tree-like) Resolution (Haken 1985). $\square$

We can also easily obtain a *non-tight* program family to witness the separation demonstrated in Theorem 5.2. Consider the family

$$\{\text{PHP}^{n+1}_n \cup \{p_{i,j} \leftarrow p_{i,j} \mid 1 \leq i \leq n+1, 1 \leq j \leq n\}\},$$

[6] The polynomial-length E-RES proof for $\mathcal{C}_{\text{PHP}}$ is not described in detail in (Cook 1976). Details on the structure of the RES proof can be found in (Järvisalo and Junttila 2008). The intuitive idea is that the extension allows for reducing PHP^{n+1}_n to PHP^n_{n-1} with a polynomial number of resolution steps.

which is non-tight with the additional self-loops $\{p_{i,j} \leftarrow p_{i,j}\}$, but preserves (un)satisfiability of PHP_n^{n+1} for all n. Since the self-loops do not contribute to the proofs for PHP_n^{n+1}, ASP-T still has exponential-length minimal proofs for these programs, while the polynomial-length E-ASP-T proof presented in the proof of Theorem 5.2 is still valid.

The generality of the arguments used in the proof of Theorem 5.2 is not limited to the specific family PHP_n^{n+1} of NLPs. For understanding the general idea behind the explicit construction of EPHP_n^{n+1}, it is informative to notice the following. Instead of considering PHP_n^{n+1}, one can apply the argument in the proof Theorem 5.2 using any tight NLP Π which represents a set of clauses $\mathcal{C}$ for which (i) there is no polynomial-length RES proof, but for which (ii) there is a polynomial-length E-RES proof . By property (ii) we know from Theorem 5.1 (ii) that there is a polynomial-length E-ASP-T proof for Π.

5.3 Program Simplification and Complexity

We will now give an interesting corollary of Theorem 5.2, addressing the effect of program simplification on the length of proofs in ASP-T.

Tightly related to the development of efficient solver implementations for ASP programs arising from practical applications is the development of techniques for *simplifying* programs. Practically relevant programs are often generated automatically, and in the process a large number of redundant constraints is produced. Therefore efficient program simplification through *local transformation rules* is important. While various satisfiability-preserving local transformation rules for simplifying logic programs have been introduced (see (Eiter et al. 2004) for example), 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

$$\mathsf{red}(\Pi) \quad = \quad \Pi \setminus \{r \in \Pi \mid \mathsf{head}(r) \notin \bigcup_{B \in \mathsf{body}(\Pi)} (B^+ \cup B^-) \text{ and } \mathsf{head}(r) \neq \bot\}.$$

A polynomial-time simplification algorithm $\mathsf{red}^*(\Pi)$ is obtained by closing program Π under red. Notice that we have $\mathsf{red}^*(\mathrm{EPHP}_n^{n+1}) = \mathrm{PHP}_n^{n+1}$. Thus, by Theorem 5.2, red^* transforms a program family having polynomial complexity in ASP Tableaux into one with exponential complexity with respect to n.

The rules removed by red^* are redundant with respect to satisfiability of the program in the sense that red^* preserves *visible equivalence* (Janhunen 2006). The visible equivalence relation takes the interfaces of programs into account: $\mathsf{atom}(\Pi)$ 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. Now if one defines $\mathsf{v}(\Pi) = \mathsf{atom}(\mathsf{red}^*(\Pi)) = \mathsf{v}(\mathsf{red}^*(\Pi))$, that is, assuming that the atoms removed by red^* are hidden in Π, one can see that $\mathsf{red}^*(\Pi) \equiv_{\mathrm{v}} \Pi$. Hence, even though there is a bijective correspondence between the stable models of EPHP_n^{n+1} and $\mathsf{red}^*(\mathrm{EPHP}_n^{n+1}) = \mathrm{PHP}_n^{n+1}$, red^* causes a superpolynomial blow-up in the length of proofs in ASP-T and the related solvers, if applied before actually proving EPHP_n^{n+1}.

6 Experiments

We experimentally evaluate how well current state-of-the-art ASP solvers can make use of the additional structure introduced to programs using the extension rule. For the experiments, we ran the solvers [7] smodels (Simons et al. 2002) (version 2.33, a widely used lookahead solver), clasp (Gebser et al. 2007) (version 1.1.0, with many techniques—including conflict learning—adopted from DPLL-based SAT solvers), and cmodels (Giunchiglia et al. 2006) (version 3.77, a SAT-based ASP solver running the conflict-learning SAT solver zChaff (Moskewicz et al. 2001) version 2007.3.12 as the back-end). The experiments were run on standard PCs with 2-GHz AMD 3200+ processors under Linux. Running times were measured using `/usr/bin/time`.

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}, $\mathrm{CPHP}_n^{n+1} = \mathrm{PHP}_n^{n+1} \cup \bigcup_{1<l\leq n} \mathrm{EXT}^l$, and EPHP_n^{n+1}. By Theorem 5.2 the solvers should in theory be able to exhibit polynomially scaling numbers 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 RES (Beame et al. 2004). The results for $n = 10 \ldots 12$ are shown in Table 1. While the number of decisions for the conflict-learning solvers clasp

[7] We note that the detailed results reported here differ somewhat from those reported in the conference version of this work (Järvisalo and Oikarinen 2007). This is due to the fact that, for the current article, we used more recent versions of the solvers.

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	34.02	119.69	8.65	164382	144416	0
smodels	11	486.44	1833.48	21.70	1899598	1584488	0
smodels	12	-	-	49.28	-	-	0
clasp	10	6.81	7.29	10.05	337818	216894	38863
clasp	11	58.48	45.00	82.07	1840605	882393	203466
clasp	12	579.28	509.43	941.23	12338982	6434939	1467623
cmodels	10	1.60	1.69	7.87	8755	8579	12706
cmodels	11	8.20	8.51	43.96	24318	23758	42782
cmodels	12	46.33	54.26	122.72	88419	94917	88499

and cmodels is somewhat reduced by the extensions, the solvers do not seem to be able to reproduce the polynomial-length 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 is due to the fact that smodels' complete lookahead notices that by branching on the critical extension atoms (as in part (ii) of the proof of Theorem 5.2) the false branch becomes contradictory immediately. With this in mind, an interesting further study out of the scope of this work would be the possibilities of integrating conflict learning techniques with (partial) lookahead.

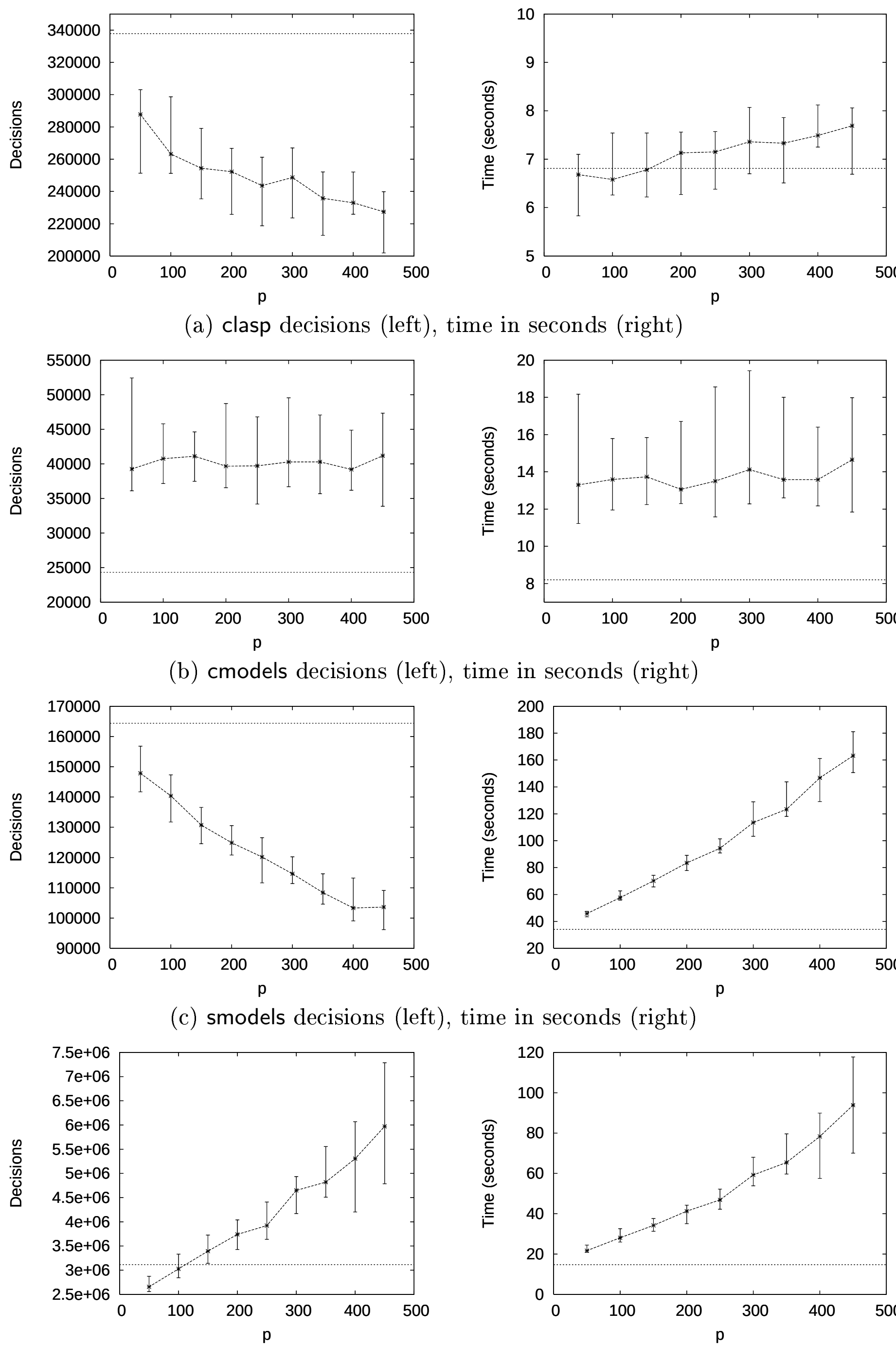
In the second experiment, we study the effect of having a modest number of redundant rules on the behavior of ASP solvers. For this we apply the procedure AddRandomRedundancy(Π, n, p) shown in Algorithm 1. Given a program Π, the procedure iteratively adds rules of the form $r_i \leftarrow l_1, l_2$ to Π, 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.

Algorithm 1 AddRandomRedundancy(Π, n, p)

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

In Figure 5, the median, minimum, and maximum number of decisions and running times for the solvers on AddRandomRedundancy$(\mathrm{PHP}_n^{n+1}, n, p)$ are shown for $p = 50, 100, \ldots, 450$ over 15 trials for each value of p. 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 behavior. 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 likely due to smodels' lookahead. We also ran the experiment for smodels without using lookahead (d). This had a visible effect on the number of decisions compared to smodels on PHP_n^{n+1}.

The most interesting effect is seen for clasp (a); clasp benefits from the added rules with respect to the number of decisions, 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,



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

for benefiting from redundancy with running times in mind, optimized lightweight propagation mechanisms are essential.

As a final remark, an interesting observation is that the effect of the transformation presented in (Anger et al. 2006), 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 of 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, which we believe to be for the benefit of both of the communities.

Our experiments show the intricate nature of the interplay between redundant problem structure and the hardness of solving ASP instances. We conjecture that more systematic use of the extension rule is possible and may even 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.

8 Acknowledgements

The authors thank Ilkka Niemelä for comments on a manuscript of this article. Financial support from Helsinki Graduate School in Computer Science and Engineering, Academy of Finland (grants #211025 and #122399), Emil Aaltonen Foundation, Nokia Foundation, Finnish Foundation for Technology Promotion TES, Jenny and Antti Wihuri Foundation (MJ), and Finnish Cultural Foundation (EO) is gratefully acknowledged.

References

Anger, C., Gebser, M., Janhunen, T., and Schaub, T. 2006. What's a head without a body? In *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI 2006)*, G. Brewka, S. Coradeschi, A. Perini, and P. Traverso, Eds. IOS Press, 769–770.

Anger, C., Gebser, M., Linke, T., Neumann, A., and Schaub, T. 2005. The nomore++ approach to answer set solving. In *Proceedings of the 12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR*

2005), G. Sutcliffe and A. Voronkov, Eds. Lecture Notes in Computer Science, vol. 3835. Springer, 95–109.

Baral, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving.* Cambridge University Press.

Beame, P., Kautz, H., and Sabharwal, A. 2004. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research 22*, 319–351.

Beame, P. and Pitassi, T. 1998. Propositional proof complexity: Past, present, and future. *Bulletin of the EATCS 65*, 66–89.

Ben-Eliyahu, R. and Dechter, R. 1994. Propositional semantics for disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence 12,* 1–2, 53–87.

Ben-Sasson, E., Impagliazzo, R., and Wigderson, A. 2004. Near optimal separation of tree-like and general resolution. *Combinatorica 24,* 4, 585–603.

Brass, S. and Dix, J. 1995. Characterizations of the stable semantics by partial evaluation. In *Proceedings of the 3rd International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 1995)*, V. W. Marek and A. Nerode, Eds. Lecture Notes in Computer Science, vol. 928. Springer, 85–98.

Brooks, D. R., Erdem, E., Erdogan, S. T., Minett, J. W., and Ringe, D. 2007. Inferring phylogenetic trees using answer set programming. *Journal of Automated Reasoning 39,* 4, 471–511.

Clark, K. L. 1978. Negation as failure. In *Logic and Data Bases*, H. Gallaire and J. Minker, Eds. Plenum Press, 293–322.

Cook, S. A. 1976. A short proof of the pigeon hole principle using extended resolution. *SIGACT News 8,* 4, 28–32.

Cook, S. A. and Reckhow, R. A. 1979. The relative efficiency of propositional proof systems. *Journal of Symbolic Logic 44,* 1, 36–50.

Davis, M., Logemann, G., and Loveland, D. 1962. A machine program for theorem proving. *Communications of the ACM 5,* 7, 394–397.

Davis, M. and Putnam, H. 1960. A computing procedure for quantification theory. *Journal of the ACM 7,* 3, 201–215.

Eiter, T., Fink, M., Tompits, H., and Woltran, S. 2004. Simplifying logic programs under uniform and strong equivalence. In *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2004)*, V. Lifschitz and I. Niemelä, Eds. Lecture Notes in Computer Science, vol. 2923. Springer, 87–99.

Erdem, E., Lifschitz, V., and Ringe, D. 2006. Temporal phylogenetic networks and logic programming. *Theory and Practice of Logic Programming 6,* 5, 539–558.

Fages, F. 1994. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science 1*, 51–60.

Gebser, M., Kaufmann, B., Neumann, A., and Schaub, T. 2007. Conflict-driven answer set solving. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, M. M. Veloso, Ed. 286–392.

Gebser, M. and Schaub, T. 2006a. Characterizing ASP inferences by unit propagation. In *ICLP Workshop on Search and Logic: Answer Set Programming and SAT*, E. Giunchiglia, V. Marek, D. Mitchell, and E. Ternovska, Eds. 41–56.

Gebser, M. and Schaub, T. 2006b. Tableau calculi for answer set programming. In *Proceedings of the 22nd International Conference on Logic Programming (ICLP 2006)*, S. Etalle and M. Truszczynski, Eds. Lecture Notes in Computer Science, vol. 4079. Springer, 11–25.

Gebser, M. and Schaub, T. 2007. Generic tableaux for answer set programming. In *Proceedings of the 23rd International Conference on Logic Programming (ICLP 2007)*,

V. Dahl and I. Niemelä, Eds. Lecture Notes in Computer Science, vol. 4670. Springer, 119–133.

Gelfond, M. and Leone, N. 2002. Logic programming and knowledge representation – the A-Prolog perspective. *Artificial Intelligence 138,* 1–2, 3–38.

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference and Symposium on Logic Programming (ICLP/SLP 1988)*, R. A. Kowalski and K. A. Bowen, Eds. MIT Press, 1070–1080.

Giunchiglia, E., Lierler, Y., and Maratea, M. 2006. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning 36,* 4, 345–377.

Giunchiglia, E. and Maratea, M. 2005. On the relation between answer set and SAT procedures (or, between CMODELS and SMODELS). In *Proceedings of the 21st International Conference on Logic Programming (ICLP 2005)*, M. Gabbrielli and G. Gupta, Eds. Lecture Notes in Computer Science, vol. 3668. Springer, 37–51.

Hai, L., Jigui, S., and Yimin, Z. 2003. Theorem proving based on the extension rule. *Journal of Automated Reasononing 31,* 1, 11–21.

Haken, A. 1985. The intractability of resolution. *Theoretical Computer Science 39,* 2–3, 297–308.

Janhunen, T. 2006. Some (in)translatability results for normal logic programs and propositional theories. *Journal of Applied Non-Classical Logics 16,* 1-2, 35–86.

Järvisalo, M. and Junttila, T. 2008. Limitations of restricted branching in clause learning. *Constraints*. in press.

Järvisalo, M. and Oikarinen, E. 2007. Extended ASP tableaux and rule redundancy in normal logic programs. In *Proceedings of the 23rd International Conference on Logic Programming (ICLP 2007)*, V. Dahl and I. Niemelä, Eds. Lecture Notes in Computer Science, vol. 4670. Springer, 134–148.

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., and Scarcello, F. 2006. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic 7,* 3, 499–562.

Lifschitz, V. 2002. Answer set programming and plan generation. *Artificial Intelligence 138,* 1–2, 39–54.

Lifschitz, V. and Razborov, A. 2006. Why are there so many loop formulas? *ACM Transactions on Computational Logic 7,* 2, 261–268.

Lifschitz, V. and Turner, H. 1994. Splitting a logic program. In *Proceedings of the 11th International Conference on Logic Programming*, P. V. Hentenryck, Ed. MIT Press, 23–37.

Lin, F. and Zhao, J. 2003. On tight logic programs and yet another translation from normal logic programs to propositional logic. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI 2003)*, G. Gottlob and T. Walsh, Eds. Morgan Kaufmann, 853–858.

Lin, F. and Zhao, Y. 2004. ASSAT: Computing answer sets of a logic program by SAT solvers. *Artificial Intelligence 157,* 1–2, 115–137.

Marek, V. W. and Truszczyński, M. 1999. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*, K. R. Apt, V. W. Marek, M. Truszczyński, and D. S. Warren, Eds. Springer, 375–398.

Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. 2001. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC 2001)*. ACM, 530–535.

Niemelä, I. 1999. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence 25,* 3–4, 241–273.

NOGUEIRA, M., BALDUCCINI, M., GELFOND, M., WATSON, R., AND BARRY, M. 2001. An A-Prolog decision support system for the space shuttle. In *Proceedings of the 3rd International Symposium on Practical Aspects of Declarative Languages (PADL 2001)*, I. V. Ramakrishnan, Ed. Lecture Notes in Computer Science, vol. 1990. Springer, 169–183.

SIMONS, P., NIEMELÄ, I., AND SOININEN, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence 138,* 1–2, 181–234.

SOININEN, T., NIEMELÄ, I., TIIHONEN, J., AND SULONEN, R. 2001. Representing configuration knowledge with weight constraint rules. In *Proceedings of the 1st International Workshop on Answer Set Programming: Towards Efficient and Scalable Knowledge (ASP 2001)*, A. Provetti and T. C. Son, Eds.

TSEITIN, G. S. 1969. On the complexity of derivation in propositional calculus. In *Studies in Constructive Mathematics and Mathematical Logic, Part II*, A. Slisenko, Ed. Seminars in Mathematics, V.A. Steklov Mathematical Institute, Leningrad, vol. 8. Consultants Bureau, 115–125. English translation appears in *Automation of Reasoning 2: Classical Papers on Computational Logic 1967–1970* J. Siekmann and G. Wrightson, Eds. Springer (1983), 466–483.