ECAI 2014
T. Schaub et al. (Eds.)

doi:10.3233/978-1-61499-419-0-645

A finite-valued solver for disjunctive fuzzy answer set programs

Mushthofa Mushthofa [1] and **Steven Schockaert** [2] and **Martine De Cock**[1 3]

Abstract. Fuzzy Answer Set Programming (FASP) is a declarative programming paradigm which extends the flexibility and expressiveness of classical Answer Set Programming (ASP), with the aim of modeling continuous application domains. In contrast to the availability of efficient ASP solvers, there have been few attempts at implementing FASP solvers. In this paper, we propose an implementation of FASP based on a reduction to classical ASP. We also develop a prototype implementation of this method. To the best of our knowledge, this is the first solver for disjunctive FASP programs. Moreover, we experimentally show that our solver performs well in comparison to an existing solver (under reasonable assumptions) for the more restrictive class of normal FASP programs.

1 Introduction

Answer Set Programming (ASP) has become one of the most popular declarative/logic programming paradigms in recent years [3]. The popularity of ASP has been driven, in part, by the availability of competitive answer set solvers, such as the Potassco suite [10], DLV [14], ASSAT [16], etc. This has enabled a wide variety of practical applications (see e.g., [15], [8], [9] and [7]). Although ASP is well suited for modeling combinatorial search problems [8], it is less suitable for continuous domains. Fuzzy Answer Set Programming (FASP) [20] is a generalization of classical ASP where atoms are allowed to have a graded level of truth (usually between 0 and 1). A general formulation of FASP has been given in [12]. Despite the promising theoretical power of FASP, research on the implementation of FASP solvers has not reached the maturity level of classical ASP solvers. Previous research results related to the implementation of FASP solvers include: (1) a reduction of FASP programs to sets of fuzzy propositional formulas [13], (2) a reduction to bilevel linear programming [5] and (3) a meta-programming approach using HEX-programs [21]. Furthermore, a set of operators for approximating fuzzy answer sets have been proposed [2], which can improve the efficiency of answer set computation using mixed integer or bilevel linear programming. Unfortunately, these existing solutions are concerned with only a limited subset of the full FASP language as defined in [12]. Our aim is to introduce a new method to evaluate FASP programs from a larger fragment, while at the same time being competitive on classes of programs that are already covered by the current methods.

[1] Dept. of Applied Math., Comp. Sc. and Statistics, Ghent University, Belgium email: {Mushthofa.Mushthofa, Martine.DeCock}@UGent.be
[2] School of Computer Science & Informatics, Cardiff University, UK, email: S.Schockaert@cs.cardiff.ac.uk
[3] Center for Data Science, University of Washington Tacoma, USA, email: mdecock@uw.edu

Although fuzzy logic is based on continuous connectives and has a semantics based on infinitely many truth values, satisfiability in many types of fuzzy logics can be checked using finite methods. For example, in [17] and [1], the authors showed that checking validity and satisfiability in infinite-valued Łukasiewicz logic can be reduced to checking validity and satisfiability in suitably chosen sets of finite-valued Łukasiewicz logic. In particular, the satisfiability of a Łukasiewicz formula ϕ can be checked by checking its satisfiability in a k-valued Łukasiewicz logic, where k is an integer which is exponentially bounded by the number of variable occurrences in ϕ. In other words, for each satisfiable formula ϕ, there exists a satisfying truth assignment of ϕ such that each variable has a truth value taken from the set $\mathbb{Q}_k = \{\frac{0}{k}, \frac{1}{k}, \ldots, \frac{k-1}{k}, \frac{k}{k}\}$, for a certain k. However, there is currently no known method to efficiently determine the appropriate k for a given ϕ. In [19] this analysis is extended to obtain a smaller bound for k on many practical instances of Łukasiewicz formulas, by looking at the structure of the formulae, instead of just the number of occurrences of their variables.

In [18], a similar result for fuzzy answer set programming was obtained through the use of the so-called fuzzy equilibrium logic. The result essentially states that every FASP program that has an answer set must have at least one answer set having the truth value of each literal taken from the set $\mathbb{Q}_k$ for a certain k bounded exponentially by the number of atoms in the program. Thus, in principle, one may need to check exponentially many possible values of k to find an answer set. However, as the result in [19] suggested, in many practical cases, fuzzy answer sets may be found after checking only a small number of values of k.

In this paper, we implement a solver based on this idea of looking for finite fuzzy answer sets. In particular, we will show how classical ASP can be used to find finite-valued answer sets of FASP programs. The main idea is to encode the particular semantics of the FASP program into an ASP program (parameterized by the choice of k), in such a way that every answer set of the ASP program corresponds to an answer set of the original FASP program whose truth values are taken from $\mathbb{Q}_k$. We will also discuss a prototype implementation of a FASP solver using this method and experimentally assess its effectiveness. We consider only FASP under Łukasiewicz semantics. However, the main ideas presented here can, in-principle, be adapted to other fuzzy logic semantics.

2 Answer set programming

In this section, we consider the syntax and semantics of a commonly studied class of ASP, namely the disjunctive answer set programs. As usual, let $a \equiv p(t_1, t_2, \ldots, t_n)$ be an atom with arity n, where p is the predicate name and each t_i is either a constant symbol or a

variable. A (classical) literal is an atom a or its classical negation $\neg a$, with $\neg\neg a \equiv a$. A Negation-As-Failure (NAF) literal is an expression of the form **not** a, where a is any classical literal. A NAF literal intuitively denotes the situation where a is not proved to be true. A disjunctive ASP rule is an expression of the form

$$r \equiv a_1 \vee \ldots \vee a_k \leftarrow b_1 \wedge \ldots \wedge b_n \wedge \textbf{not } c_1 \wedge \ldots \wedge \textbf{not } c_m$$

with $k \geq 0, n \geq 0, m \geq 0$ and the a_i's, b_i's and c_i's are classical literals. We say that the set $\{a_1, \ldots, a_k\}$ is the head of the rule, $H(r)$, while $B^+(r) = \{b_1, \ldots, b_n\}$ and $B^-(r) = \{c_1, \ldots, c_m\}$ are the positive and negative body literals, respectively. We denote $B(r) = B^+(r) \cup B^-(r)$.

If $m = 0$, we say that the rule is positive. A rule with $k = 0$ is called a *constraint* rule. A rule with $k = 1$ is called a *normal* rule. If the body is empty (i.e. $m = n = 0$), the rule is also called a fact. A positive normal rule is also called a *Horn* rule. A program is called [definite, positive, normal] iff it contains only [Horn, positive, normal] rules, respectively.

A literal is called a *ground literal* if it contains no variables. The Herbrand base $\mathcal{B}_\mathcal{P}$ of a program $\mathcal{P}$ is the set of all ground literals that can be formed using the predicates and constant symbols appearing in $\mathcal{P}$. An interpretation I of $\mathcal{P}$ is a subset of $\mathcal{B}_\mathcal{P}$. An interpretation I is called consistent iff $\neg a \notin I$ whenever $a \in I$, and inconsistent otherwise. A consistent interpretation I satisfies a rule r iff $H(r) \cap I \neq \emptyset$ whenever $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$. In such a case, we write $I \models r$. An interpretation I satisfies a program $\mathcal{P}$ iff $I \models r$ for every $r \in \mathcal{P}$. We then say that I is a model of $\mathcal{P}$.

For a positive program (i.e. one without NAF literals) $\mathcal{P}$, a model I of $\mathcal{P}$ is called an answer set of $\mathcal{P}$ iff it is a minimal model of $\mathcal{P}$ w.r.t. set-inclusion. For programs with negation-as-failure, we follow the definition of the GL-Reduct from [11]. The *reduct* of $\mathcal{P}$ w.r.t. an interpretation I is the positive program $\mathcal{P}^I$ obtained by: (1) deleting all the rules r with $B^-(r) \cap I \neq \emptyset$, and (2) deleting all the NAF literals in the remaining rules. Then, a model I of $\mathcal{P}$ is an answer set of $\mathcal{P}$ iff it is a minimal model of $\mathcal{P}^I$.

The set of answer sets of $\mathcal{P}$ is denoted by $\mathcal{ANS}(\mathcal{P})$. Definite programs have exactly one answer set. A positive program without constraints has at least one answer set, while a general ASP program may have zero, one or more answer sets. A program is called *consistent* iff it has at least one answer set (i.e. $\mathcal{ANS}(\mathcal{P}) \neq \emptyset$), otherwise it is *inconsistent*. Let $\mathcal{P}$ be any ASP program, and let $\mathcal{P}'$ be a new ASP program obtained from $\mathcal{P}$ by replacing every occurrence of a (classically) negated literal $\neg a$ with a fresh atom symbol a' and adding the constraint $\leftarrow a, a'$. Then every answer set $A \in \mathcal{ANS}(\mathcal{P})$ can be obtained from an answer set $A' \in \mathcal{ANS}(\mathcal{P}')$ by replacing every occurrence of the atoms of the form a' with a. As a result, we only need to consider ASP programs without classical negations. The decision problems associated to ASP programs and their complexity results are discussed in [6].

3 Fuzzy answer set programming

Here, we adopt the formulation of FASP as described in [5], which is based on the formulation by [12], but focuses specifically on Łukasiewicz semantics. Similar to classical ASP programs, in FASP we assume the availability of a set of predicate symbols from which we construct atoms. A (classical) literal is either a constant symbol $\overline{c}$ where $c \in [0,1] \cap \mathbb{Q}$, an atom a or a classical negation literal $\neg a$. An extended literal is a classical literal a or a NAF literal **not** a. A *head/body expression* is a formula defined recursively as follows:

- a constant $\overline{c}, c \in [0,1] \cap \mathbb{Q}$, and a classical literal a are head expressions.
- a constant $\overline{c}, c \in [0,1] \cap \mathbb{Q}$, and an extended literal a are body expressions.
- if α and β are head/body expressions, then $\alpha \otimes \beta$, $\alpha \oplus \beta$, $\alpha \veebar \beta$ and $\alpha \barwedge \beta$ are also head/body expressions, respectively.

Constants and (classical) literals are also called *atomic* expressions, as opposed to *composite* expressions, which are expressions that contain the application of one or more of the logical operators. A FASP program is a finite set of rules of the form:

$$r \equiv \alpha \leftarrow \beta$$

where α is a head expression (called the *head* of r) and β is a body expression (called the *body* of r). As in classical ASP, we also write $H(r)$ and $B(r)$ to denote the head and body of a rule r, respectively. A FASP rule of the form $a \leftarrow \overline{c}$ for an atom a and a constant c is called a fact. A FASP rule of the form $\overline{c} \leftarrow \beta$ is called a constraint. A rule which does not contain any application of the operator **not** is called a *positive* rule. A rule which has only one literal in the head is called a *normal* rule. A FASP program is called [*positive*, *normal*] iff it contains only [positive, normal] rules, respectively. A positive normal program which has no constraints is called a *simple* program.

The semantics of FASP is traditionally defined on a complete truth-lattice $\mathcal{L} = \langle L, \leq_L \rangle$ [4]. In this paper, we consider two types of truth-lattice: the infinite-valued lattice $\mathcal{L}_\infty = \langle [0,1], \leq \rangle$ and the finite-valued lattices $\mathcal{L}_k = \langle \mathbb{Q}_k, \leq \rangle$, for integer $k \geq 1$. An interpretation of a FASP program $\mathcal{P}$ is a function $I : \mathcal{B}_\mathcal{P} \mapsto \mathcal{L}$ which can be extended to to expressions and rules as follows:

- $I(\overline{c}) = c$, for a constant $c \in \mathcal{L}$.
- $I(\alpha \otimes \beta) = \max(I(\alpha) + I(\beta) - 1, 0)$.
- $I(\alpha \oplus \beta) = \min(I(\alpha) + I(\beta), 1)$.
- $I(\alpha \veebar \beta) = \max(I(\alpha), I(\beta))$.
- $I(\alpha \barwedge \beta) = \min(I(\alpha), I(\beta))$.
- $I(\textbf{not } \alpha) = 1 - I(\alpha)$.
- $I(\alpha \leftarrow \beta) = \min(1 - I(\beta) + I(\alpha), 1)$.

for appropriate expressions α and β. Here, the operators **not**, $\otimes, \oplus, \veebar, \barwedge$ and $\leftarrow$ denote the Łukasiewicz negation, t-norm, t-conorm, maximum, minimum and implication, respectively.

An interpretation I is consistent iff $I(l) + I(\neg l) \leq 1$ for each $l \in \mathcal{B}_\mathcal{P}$. We say that a consistent interpretation I of $\mathcal{P}$ satisfies a FASP rule r iff $I(r) = 1$. This condition is equivalent to $I(H(r)) \geq I(B(r))$. An interpretation is a model of a program $\mathcal{P}$ iff it satisfies every rule of $\mathcal{P}$. For interpretations I_1, I_2, we write $I_1 \leq I_2$ iff $I_1(l) \leq I_2(l)$ for each $l \in \mathcal{B}_\mathcal{P}$, and $I_1 < I_2$ iff $I_1 \leq I_2$ and $I_1 \neq I_2$. We call a fuzzy model I of $\mathcal{P}$ a *minimal* model if there is no other fuzzy model J of $\mathcal{P}$ such that $J < I$.

For a positive FASP program $\mathcal{P}$, a fuzzy model I of $\mathcal{P}$ is called a *fuzzy answer set* of $\mathcal{P}$ iff it is a minimal model of $\mathcal{P}$. For a non-positive FASP program $\mathcal{P}$, a generalization of the GL reduct is defined in [12] as follows: the reduct of a rule r w.r.t. an interpretation I is the positive rule r^I obtained by replacing each occurrence of **not** a by the constant $\overline{I(\textbf{not } a)}$. The reduct of a FASP program $\mathcal{P}$ w.r.t. a fuzzy interpretation I is then defined as the positive program $\mathcal{P}^I = \{r^I \mid r \in \mathcal{P}\}$. A fuzzy model I of $\mathcal{P}$ is called a fuzzy answer set of $\mathcal{P}$ iff I is a fuzzy answer set of $\mathcal{P}^I$. The set of all the fuzzy answer sets of a FASP program $\mathcal{P}$ is denoted by $\mathcal{ANS}(\mathcal{P})$. A simple FASP program has exactly one fuzzy answer set. A positive FASP program may have no, one or several fuzzy answer sets. A FASP program is called *consistent* iff it has at least one fuzzy answer set, and *inconsistent* otherwise.

Example 3.1. Consider the FASP program $\mathcal{P}_1$ which has the following rules:

$$\{a \leftarrow \textbf{not } c, b \leftarrow \textbf{not } c, c \leftarrow a \oplus b\}$$

One can check that under both the truth-lattice $\mathcal{L}_3$ and $\mathcal{L}_\infty$, the fuzzy interpretation $I_1 = \{(a, \frac{1}{3}), (b, \frac{1}{3}), (c, \frac{2}{3})\}$ is a minimal model of $\mathcal{P}_1^{I_1}$, and hence it is an answer set of $\mathcal{P}_1$. However, one can see that the program admits no answer sets under any $\mathcal{L}_k$, where k is a positive integer not divisible by 3.

As noted in [5], one can also "simulate" the classical negation by replacing every occurrence of a classically negated atom $\neg a$ with a fresh atom symbol a' and adding the constraint $\overline{0} \leftarrow a \otimes a'$. This allows us to focus only on FASP program without classical negations. In [5], the decision problems associated to FASP programs and their complexity results under Łukasiewicz semantics are also discussed. Note that, similar to the case in classical ASP, the decision problems associated to FASP programs can be reduced to the problem of deciding whether the program has an answer set or not, i.e., deciding satisfiability. This allows us to focus only on the problem of deciding satisfiability of FASP programs.

4 Solving finite satisfiability of FASP using ASP

The results in [1] and [19] suggest that solving FASP programs using finite methods could potentially be useful. Call a fuzzy answer set A of $\mathcal{P}$ a k-answer set of $\mathcal{P}$ iff the truth values of the atoms in A are taken from the set $\mathbb{Q}_k$. Then it can be seen that every k-answer set of a FASP program $\mathcal{P}$ under the infinite-valued truth-lattice $\mathcal{L}_\infty$ is also an answer set of $\mathcal{P}$ under the finite-valued truth-lattice $\mathcal{L}_k$. This means that we can find every answer set of $\mathcal{P}$ under $\mathcal{L}_\infty$ by iteratively finding its answer sets under $\mathcal{L}_k$, for $k \geq 1$. A result in [1] shows that exponentially many k need to be checked to exhaustively find all answer sets of the program under $\mathcal{L}_\infty$. As we will see in Section 5, however, in practice usually only a small number of values for k needs to be checked.

We will show how answer sets of FASP programs under a finite-valued truth lattice $\mathcal{L}_k$ can be found using a reduction to classical ASP. In the next sections, we will show how we can rewrite FASP rules into equivalent forms prior to the translation (to make the translation process more efficient) as well as the details of the translation itself. Finally, we will analyse the conditions under which this approach is also successful for finding answer sets in the infinitely-valued truth lattice $\mathcal{L}_\infty$.

4.1 FASP rule rewriting

Before we perform the translation to ASP, we rewrite the FASP rules into an equivalent set of rules which follow a certain "standardized form", in order to make the translation simpler and more efficient. First, if a rule $r \in \mathcal{P}$ contains a constant symbol $\overline{c}$ in the body, replace r with

$$H(r) \leftarrow B(r)[\overline{c}/p]$$

where p is a fresh atom symbol. Here, $x[y/z]$ is obtained by replacing each occurrence of y in x with z. If $c > 0$, add the rule $p \leftarrow c$ to the program. If a rule $r \in \mathcal{P}$ contains a constant symbol $\overline{c}$ in the head, replace r with

$$H(r)[\overline{c}/p] \leftarrow B(r)$$

where a fresh atom symbol p is used for every constant appearing in the program. If $c < 1$, add the rules $\{p \leftarrow \overline{c}, \overline{c} \leftarrow p\}$ to the program. It is not hard to see that these replacements do not change the meaning of the program.

Next, we rewrite each rule such that the resulting rules contain at most one application of the logical operators ($\oplus, \otimes, \overline{\wedge}, \underline{\vee}$, **not**). The idea is to recursively split each application of the operators on composite expressions by defining new auxiliary atoms to capture the truth value of each of the composite expressions, and then replace the original rule with a set of equivalent rules. For example, for a rule $r \in \mathcal{P}$ of the form $a \leftarrow \beta * \gamma$ where $* \in \{\oplus, \otimes, \underline{\vee}, \overline{\wedge}\}$, a is a classical literal and β and γ are composite expressions, we replace r with the following set of rules $\{a \leftarrow p * q, p \leftarrow \beta, q \leftarrow \gamma\}$ where p and q are fresh atom symbols. While, for a rule $r \in \mathcal{P}$ of the form $\alpha * \beta \leftarrow c$ where $* \in \{\oplus, \otimes, \underline{\vee}, \overline{\wedge}\}$ and α and β are composite expressions and c is a classical literal, we replace r with the following rules: $\{p * q \leftarrow c, p \leftarrow \alpha, \alpha \leftarrow p, q \leftarrow \beta, \beta \leftarrow q\}$ where p and q are fresh atom symbols. Due to space constraints, we omit the proof that these rewriting steps result in a FASP program which is equivalent to the original program, in the sense that every answer set of the original program can be extended to an answer set of the new program (by assigning a truth value to each of the newly introduced atoms) and conversely, that the restriction of any answer set from the new program to the atoms which occur in the original program is indeed an answer set of the original program. The following proposition holds.

Proposition 1. Using a finite number of rewriting steps, we can convert any program $\mathcal{P}$ into an equivalent program $Rw(\mathcal{P})$ containing only rules of the following forms:

1. A fact $a \leftarrow \overline{c}$ for an atom a and a constant $\overline{c}, c \in (0, 1]$.
2. A constraint $\overline{c} \leftarrow a$ for an atom a and a constant $\overline{c}, c \in [0, 1)$.
3. A rule with no operator in the body nor in the head $a \leftarrow b$.
4. A rule with NAF-literal in the body $a \leftarrow \textbf{not } b$ for atoms a and b.
5. A rule with a binary operator in the body $a \leftarrow b * c$, with a, b and c atoms and $* \in \{\otimes, \oplus, \underline{\vee}, \overline{\wedge}\}$.
6. A rule with a binary operator in the head $a * b \leftarrow c$, with a, b and c atoms and $* \in \{\otimes, \oplus, \underline{\vee}, \overline{\wedge}\}$.

and that the size of $Rw(\mathcal{P})$ is $O(n \cdot m)$, where n is the number of rules in $\mathcal{P}$ and m is the maximum number of atom occurrences in the rules of $\mathcal{P}$.

4.2 Translation to classical ASP

To find the answer sets of $\mathcal{P}$ under $\mathcal{L}_k$, we perform a translation of each rule of $\mathcal{P}$ into ASP rules parametrized by k. Consider a FASP program $\mathcal{P}$ and an integer k. Assume that each rule of $\mathcal{P}$ follows the "standardized" forms as described in Proposition 1. We will translate $\mathcal{P}$ into a classical ASP program $Tr(\mathcal{P}, k)$. First, we assume the availability of atom symbols a_i for every $a \in \mathcal{B}_\mathcal{P}$ and $1 \leq i \leq k$ to be used in $Tr(\mathcal{P}, k)$.

We translate each rule of $\mathcal{P}$ as follows:

1. For a fact $r \in \mathcal{P}$ of the form: $a \leftarrow \overline{c}, c \in (0, 1]$ we add the fact $a_j \leftarrow$ to $Tr(\mathcal{P}, k)$, where $j = k * c$.
2. For a constraint $r \in \mathcal{P}$ of the form $\overline{c} \leftarrow a, c \in [0, 1)$ we add a constraint $\leftarrow a_{j+1}$ in $Tr(\mathcal{P}, k)$, where $j = k * c$.
3. A FASP rule of the form: $a \leftarrow b$ can be easily translated into classical ASP as $\{a_i \leftarrow b_i \mid 1 \leq i \leq k\}$
4. A FASP rule of the form: $a \leftarrow b \otimes c$ is equivalent to saying that $I(a) \geq \max(I(b) + I(c) - 1, 0)$ for every answer set I of $\mathcal{P}$. This means that to obtain $I(a) \geq \frac{i}{k}$ for some $1 \leq i \leq k$, we can

choose $I(b) = \frac{j}{k}$ for any $i \leq j \leq k$ and then $I(c) = 1 - \frac{j-i}{k}$. This corresponds to the following set of ASP rules:

$$\{a_i \leftarrow b_j \wedge c_{k-j+i} \mid 1 \leq i \leq k, i \leq j \leq k\}$$

5. A FASP rule of the form: $a \leftarrow b \oplus c$ is equivalent to saying that $I(a) \geq \min(I(b) + I(c), 1)$ for every answer set I of $\mathcal{P}$. This means that to obtain $I(a) \geq \frac{i}{k}$ for some $1 \leq i \leq k$, we can choose $I(b) \geq \frac{j}{k}$ for some $0 \leq j \leq i$ and then $I(c) \geq \frac{i-j}{k}$. This can be translated as the following set of ASP rules:

$$\{a_i \leftarrow b_i, a_i \leftarrow c_i, a_i \leftarrow b_j \wedge c_{i-j} \mid 1 \leq i \leq k, 0 < j < i\}$$

6. A FASP rule of the form $a \leftarrow b \veebar c$ implies that $I(a) \geq \max(I(b), I(c))$ in every answer set I. This can be translated as the following set of ASP rules

$$\{a_i \leftarrow b_i, a_i \leftarrow c_i \mid 1 \leq i \leq k\}$$

7. A FASP rule of the form $a \leftarrow b \barwedge c$ can be translated into

$$\{a_i \leftarrow b_i \wedge c_i \mid 1 \leq i \leq k\}$$

8. For the FASP rule $a \oplus b \leftarrow c$ we first create fresh atom symbols $p_{s,t}$, where $0 \leq s, t \leq k$ such that $1 \leq s + t \leq k$. Each $p_{s,t}$ encodes the situation where a and b have truth values $\frac{s}{k}$ and $\frac{t}{k}$, respectively. We then encode the semantics saying that when c has a truth value of $\frac{i}{k}$, then the sum of the truth values of a and b should be at least $\frac{i}{k}$. We must also ensure that only "minimal choices" are generated in the answer sets. For example, if $I(c) = \frac{i}{k}$ and $I(a) = \frac{j}{k}$, we must eliminate the choices which generate $I(b) > \frac{i-j}{k}$. We use the following set of ASP rules for this translation.

$$\begin{aligned}
\{p_{0,i} \vee p_{1,i-1} \vee \ldots \vee p_{i-1,1} \vee p_{i,0} \leftarrow c_i \mid 1 \leq i \leq k\} \quad \cup \\
\{a_i \leftarrow p_{i,j}, b_j \leftarrow p_{i,j} \mid 0 \leq i, j \leq k\} \quad \cup \\
\{p_{i+1,j-1} \leftarrow p_{i,j} \wedge a_{i+1} \mid 0 \leq i \leq k-1, 1 \leq i+j \leq k\} \quad \cup \\
\{p_{i-1,j+1} \leftarrow p_{i,j} \wedge b_{j+1} \mid 0 \leq j \leq k-1, 1 \leq i+j \leq k\}
\end{aligned}$$

The first two sets of rules "distribute" the truth value of c to a and b, while the last two sets of rules ensure that only minimal models are generated by eliminating the non-minimal ones. For example, if we also have the fact $a \leftarrow \frac{1}{k}$ in $\mathcal{P}$, then the rule

$$p_{k-1,1} \leftarrow p_{k,0} \wedge a_1$$

will eliminate the (otherwise generated) non-minimal answer set A of $Tr(\mathcal{P}, k)$ containing a_1 and b_k, which corresponds to a (non-minimal) fuzzy model I of $\mathcal{P}$ having $I(a) = \frac{i}{k}$ and $I(b) = 1$.

9. For the FASP rule $a \otimes b \leftarrow c$ a similar construct as the translation scheme for $a \oplus b \leftarrow c$ can be used, as follows: create atom symbols $p_{s,t}$, where $1 \leq s, t \leq k$ such that $s + t > k$, with a similar meaning as before. The rule $a \otimes b \leftarrow c$ can then be translated as:

$$\begin{aligned}
\{p_{k,i} \vee p_{k-1,i+1} \vee \ldots \vee p_{i,k} \leftarrow c_i \mid 1 \leq i \leq k\} \quad \cup \\
\{a_i \leftarrow p_{i,j}, b_j \leftarrow p_{i,j} \mid 1 \leq i, j \leq k, i+j > k\} \quad \cup \\
\{p_{i+1,j-1} \leftarrow p_{i,j} \wedge a_{i+1} \mid 1 \leq i < k, 1 \leq j \leq k, i+j > k\} \quad \cup \\
\{p_{i-1,j+1} \leftarrow p_{i,j} \wedge b_{j+1} \mid 1 \leq i \leq k, 1 \leq j < k, i+j > k\}
\end{aligned}$$

10. A FASP rule of the form $a \veebar b \leftarrow c$ states that $\max(I(a), I(b)) \geq I(c)$ in every answer set I of $\mathcal{P}$. Hence, we can translate it into the following set of ASP rules: $\{a_i \vee b_i \leftarrow c_i \mid 1 \leq i \leq k\}$
11. Similar to the previous translation, we can translate the FASP rule $a \barwedge b \leftarrow c$ into the following set of ASP rules: $\{a_i \leftarrow c_i, b_i \leftarrow c_i \mid 1 \leq i \leq k\}$
12. For a rule $a \leftarrow$ **not** b which states that $I(a) \geq 1 - I(b)$ for every answer set I, we use the following set of ASP rules:

$$\{a_i \leftarrow \textbf{not } b_{k-i+1} \mid 1 \leq i \leq k\}$$

which enforces the constraint $I(a) \geq 1 - I(b)$ while at the same time preserving the NAF-semantics.

Finally, we must add the set of rules

$$\{a_i \leftarrow a_{i+1} \mid a \in \mathcal{B}_\mathcal{P}, 1 \leq i \leq k-1\}$$

into $Tr(\mathcal{P}, k)$ to ensure that the atoms a_i are consistent with the interpretation that the truth value of a is at least $\frac{i}{k}$. We can show the following result.

Proposition 2. The number of rules in $Tr(\mathcal{P}, k)$ is $O(n \cdot k^2)$, where n is the number of rules in $\mathcal{P}$.

Now, consider a function $\mathcal{M}_k$, mapping a classical interpretation A to a fuzzy interpretation I, defined as follows:

$$I(a) = \mathcal{M}_k(A)(a) = \max\{\frac{i}{k} | a_i \in A\}$$

We can show that the following proposition holds:

Proposition 3. A is an answer set of $Tr(\mathcal{P}, k)$ iff $I = \mathcal{M}_k(A)$ is an answer set of $\mathcal{P}$ under the truth-lattice $\mathcal{L}_k$.

For the case where the truth-lattice $\mathcal{L}_\infty$ is assumed, one must perform the translation and find k-answer sets for (possibly exponentially) many values of k. If no constant symbols appear in $\mathcal{P}$, we can start looking for k-answer sets for $k = 1, 2, \ldots$ and so on. However, if $\mathcal{P}$ contains a constant symbol $\overline{c}$, where $c = \frac{a}{b}$ for some integers a and b with $\gcd(a, b) = 1$, then translating a rule such as $a \leftarrow \overline{\frac{a}{b}}$ into an ASP rule $a_j \leftarrow$ where $j = a/b * k$ requires k to be a multiple of b. Therefore, in the search for k-answer sets using the translation above, one must always choose a value of k which is a multiple of every denominator of the constants appearing in the program. In other words, if there are n constant symbols $\{\overline{a_1/b_1}, \ldots, \overline{a_n/b_n}\}$ in $\mathcal{P}$, then we choose values of k which are divisible by the least common multiple of $b_1, \ldots, b_n$. The following proposition provides the result for the infinite-valued truth lattice.

Proposition 4. For every answer set I of a FASP program $\mathcal{P}$ under the truth-lattice $\mathcal{L}_\infty$, there exists a positive integer k such that $I = \mathcal{M}_k(A)$ for some answer set A of $Tr(\mathcal{P}, k)$.

For normal programs, we additionally have the following proposition.

Proposition 5. If $\mathcal{P}$ is a normal FASP program and A is answer set of $Tr(\mathcal{P}, k)$, then $I = \mathcal{M}_k(A)$ is an answer set of $\mathcal{P}$ under the $\mathcal{L}_\infty$.

Example 4.1. Consider again the program $\mathcal{P}_1$ from Example 3.1. Obviously, $Rw(\mathcal{P}_1) = \mathcal{P}_1$. Furthermore, one can check that $Tr(\mathcal{P}_1, 1)$ and $Tr(\mathcal{P}_1, 2)$ have no answer sets. However, the ASP program $Tr(\mathcal{P}_1, 3)$ containing the following rules:

$$\begin{aligned}
\{a_i \leftarrow \textbf{not } c_{3-i+1} \mid 1 \leq i \leq 3\} \cup \\
\{b_i \leftarrow \textbf{not } c_{3-i+1} \mid 1 \leq i \leq 3\} \cup \\
\{c_i \leftarrow a_i, c_i \leftarrow b_i, c_i \leftarrow a_j \wedge b_{i-j} \mid 1 \leq i \leq 3, 1 \leq j < i\} \cup \\
\{p_i \leftarrow p_{i+1} \mid i = 1, 2, p \in \{a, b, c\}\}
\end{aligned}$$

does have an answer set, namely $A_1 = \{a_1, b_1, c_1, c_2\}$ which corresponds to the only answer set I_1 of $\mathcal{P}_1$ under $\mathcal{L}_\infty$ (i.e. $\mathcal{M}_3(A_1) = I_1$).

For disjunctive programs, the result from Proposition 5 does not necessarily follow. Consider the following example.

Example 4.2. Program $\mathcal{P}_2$ has the following rules: $\{a \oplus b \leftarrow \overline{1}, a \leftarrow b, b \leftarrow a\}$. $Tr(\mathcal{P}_2, 1)$ has one answer set, namely $A_1 = \{a_1, b_1\}$. However, $I_1 = \mathcal{M}_1(A_1) = \{(a, 1), (b, 1)\}$ is not an answer set of $\mathcal{P}_2$, whose only answer set is $I_2 = \{(a, 0.5), (b, 0.5)\}$.

For disjunctive programs, we only have the following weaker result.

Proposition 6. If $\mathcal{P}$ is a disjunctive FASP program and A is an answer set of $Tr(\mathcal{P}, k)$, then $I = \mathcal{M}_k(A)$ is an answer set of $\mathcal{P}$ under the infinitely-valued truth lattice $\mathcal{L}_\infty$ iff there is no other model J of $\mathcal{P}^I$ such that $J < I$.

This means that for disjunctive programs, when our method has found a k-answer set, we still need to verify whether it is an answer set under $\mathcal{L}_\infty$. This can easily be checked using a mixed integer programming solver, or any other method for entailment checking in Łukasiewicz logic. We omit the details.

5 Implementation and experiments

We have developed a prototype FASP solver (named ffasp) implementing the method described in this paper. The solver reads a FASP program, performs the rewriting and translation, submits the translation result to a back-end ASP solver, retrieves the answer sets back (if any) and converts them to a fuzzy answer set format. We use clingo [10] as the back-end ASP solver for ffasp. The source code for the implementation is available at `http://code.google.com/p/ffasp`.

The syntax of the input language of ffasp is similar to the classical ASP language syntax, with the addition of the following: (1) constant atoms, which are numbers between 0 and 1 in either decimal or rational format prefixed with a "#" symbol, (2) the connective "*", denoting the Łukasiewicz t-norm, (3) the connective "+", denoting the Łukasiewicz t-conorm (4) the connective "v", denoting the max operator and 5) the connective "^", denoting the min operator. Currently, the solver only allows one type of connective in the head or the body of the same rule. This does not reduce the expressivity of the language of the FASP program that can be evaluated using ffasp, since every FASP rule can be rewritten into this form. The following is an example of rules accepted by ffasp:

```
a :- #0.2 v b.
p(X,Y) + s(X) :- q(X) * r(Y) * X<Y.
b ^ #0.1 :- c * not a * #1/2.
```

Given an input program $\mathcal{P}$, ffasp first computes the value $l = \mathrm{lcm}\{b_1, \ldots, b_n\}$, where a_i/b_i, $1 \leq i \leq n$ are the constant atoms appearing in $\mathcal{P}$. If no constant atom appears in $\mathcal{P}$, then $l = 1$. The ffasp solver iteratively computes $Tr(Rw(\mathcal{P}, k))$ for each positive integer k which is a multiple of l and calls clingo to find a k-answer set. If an answer set is found, ffasp reports it and the computation stops. Otherwise, the computation is continued until a certain limit on k is reached, or until a predetermined time limit is reached. For example, given the program:

```
a :- not p.  b :- not p. c :- not p.
p :- a+b+c.  q :- a*b*c.
```

ffasp checks for k-answer set for $k = 1, 2, 3, 4$, after which the following output is produced:

```
{a[0.25], b[0.25], c[0.25], p[0.75]}
```

To asses the efficiency of the proposed method, we have compared the running time of ffasp with a recent implementation of a FASP solver called fasp, described in [2]. To the best of our knowledge, it is the only currently available FASP solver. First, we note that the language accepted by ffasp is strictly more expressive than the language accepted by fasp, e.g., disjunctions in the body are not allowed in fasp. Hence, this experiment is intended to measure the efficiency of ffasp for the fragment of the input language corresponding to normal FASP programs, as accepted by fasp.

For this experiment, we use the same instances of the problems *Graph Coloring*, *Hamiltonian Path*, *Stratified* and *Odd Cycle* as in [2], with the following modification. Since ffasp starts the search using an initial k computed from the least common multiple of all the denominators of the constants appearing in the program, its performance is affected by the number of different constants allowed in the program. We therefore make the (in practice reasonable) assumption that the number of different truth values that the constants in the program can take is bounded by a certain parameter d (i.e. they are taken from the set $\mathbb{Q}_d$). To this effect, we perform a rounding of the constant atoms appearing in instances of *Graph Coloring* and *Hamiltonian Path* used in [2] to the nearest value in $\mathbb{Q}_d$. For example, a constant atom `#0.872` in the original instances is rounded into `#0.9` in our test instances when $d = 10$, and to `#0.87` when $d = 100$. We selected 6 instances from the original instances of *Graph Coloring*, and for each of them, we generated 5 instances according to the rounding method described previously (with $d = 20, 40, 80$ and 100), yielding a total of 30 instances. For *Hamiltonian Path*, we selected 10 of the original instances and use $d = 20, 40, \ldots, 160$, yielding a total of 80 instances.

The experiment was conducted on a Macbook with OS X version 10.8.5 running on Intel Core i5 2.4 GHz with 4 GB of memory. Execution time for each instance is limited to 10 minutes while the maximum value for k is set to 100 for all instances of *Stratified* and *Odd Cycle*, as well as the instances of Graph Coloring and *Hamiltonian Path* having $d \leq 100$, and set to 200 for the instances having $d > 100$. Tabel 1 presents the results of the experiment.

All instances of *Stratified*, *Odd Cycle* and *Graph Coloring* are satisfiable, and both solvers find an answer set for each instance . Out of 80 instances of the problem *Hamiltonian Path*, both solvers agree in finding that 46 of them are satisfiable (as reported in the column "Num. of Instances.") and both produce an answer set for each instance. The fasp solver was furthermore able to prove that the remaining 34 instances are unsatisfiable, whereas our solver cannot in practice prove unsatisfiability (as this would require checking an exponential number of values for k). No time-outs were observed during this experiment. However, as d increases, computation time for ffasp increases significantly as well (which is to be expected). On the other hand, fasp does not suffer from the same disadvantage, and for larger d will eventually take over ffasp in terms of efficiency.

Although we have not provided a benchmark result to test the performance of ffasp for programs with disjunction in the body and for disjunctive programs (due to time and space constraints), we believe that the ability of ffasp to handle such classes is a significant advantage. Indeed, it has been observed in [4] that most applications of FASP require rules with disjunction in the body. Allowing applications of Łukasiewicz operators in the head and body of the rules can increase the expressivity of the language. In classical ASP, allowing

Table 1. Experiment results

Problem	d	Num. of instances	Avg. execution time (s)	
			fasp (from [2])	ffasp (our solver)
Stratified	-	90	2.136	0.527
Odd Cycle	-	90	2.130	0.111
Graph Coloring	20	6	39.224	1.883
Graph Coloring	40	6	38.858	6.247
Graph Coloring	60	6	39.035	13.903
Graph Coloring	80	6	42.563	25.425
Graph Coloring	100	6	42.969	40.177
Hamiltonian Path	20	7	21.382	0.266
Hamiltonian Path	40	6	17.503	1.021
Hamiltonian Path	60	5	22.898	2.451
Hamiltonian Path	80	5	22.806	4.615
Hamiltonian Path	100	6	27.710	7.600
Hamiltonian Path	120	6	33.386	11.684
Hamiltonian Path	140	5	24.416	16.730
Hamiltonian Path	160	6	30.565	22.019

disjunction in the body is redundant, since a rule containing disjunction in the body, such as: $a \leftarrow b \vee c$, is equivalent to the two rules: $a \leftarrow b$ and $a \leftarrow c$. In contrast, disjunction in the body of a FASP rule, such as $a \leftarrow b \oplus c$ cannot be replaced by using two normal rules, and instead increase the expressiveness of the language. For example, by allowing Łukasiewicz t-conorm in the body, one can perform the so-called "saturation technique" to force an atom a to take only Boolean values by adding the rule $a \leftarrow a \oplus a$. This allows for the mixing of fuzzy and Boolean predicates in a FASP program. Similarly, having operators in the head of FASP rules also allow for a more concise and intuitive encoding of a problem, as it is the case with having disjunction in the head of classical ASP rules.

6 Conclusion

We have proposed a new method to solve the satisfiability problem in FASP by using finite methods, and showed how the reasoning tasks in FASP can be reduced to reasoning tasks in classical ASP. A key advantage of our approach over other recent proposals, such as the ones proposed in [13], [5], [21], and more recently, [2], is that our solver is not restricted to normal programs without disjunction in the body. Indeed, most interesting problems in FASP require the use of Łukasiewicz disjunction in the body of rules [5]. Apart from the bi-level mixed integer programming (biMIP) approach proposed in [18], which is difficult to use in practice given the lack of scalable biMIP solvers, our method is the first approach that can handle such programs.

We have also developed a prototype implementation of this method, and assessed its efficiency by comparing it with a previous FASP solver implementation. The benchmark result shows that the method we propose is efficient for computing answer sets in many practical instances (given the reasonable assumption that the number of truth values that constants can take is bounded). In addition, our solver is also the first implemented FASP solver (to the best of our knowledge) to offer the ability to solve disjunctive programs, and programs with disjunctions in the body.

REFERENCES

[1] Stefano Aguzzoli and Agata Ciabattoni, 'Finiteness in infinite-valued Łukasiewicz logic', *Journal of Logic, Language and Information*, **9**(1), 5–29, (2000).

[2] Mario Alviano and Rafael Penaloza, 'Fuzzy answer sets approximations', in *Proceedings of the 29th International Conference on Logic Programming*, (2013).

[3] Chitta Baral, *Knowledge representation, reasoning and declarative problem solving*, Cambridge university press, 2003.

[4] Marjon Blondeel, Steven Schockaert, Dirk Vermeir, and Martine Cock, 'Fuzzy answer set programming: An introduction', in *Soft Computing: State of the Art Theory and Novel Applications*, eds., Ronald R. Yager, Ali M. Abbasov, Marek Z. Reformat, and Shahnaz N. Shahbazova, volume 291 of *Studies in Fuzziness and Soft Computing*, 209–222, Springer Berlin Heidelberg, (2013).

[5] Marjon Blondeel, Steven Schockaert, Dirk Vermeir, and Martine De Cock, 'Complexity of fuzzy answer set programming under Łukasiewicz semantics', *International Journal of Approximate Reasoning*, (2013).

[6] Thomas Eiter and Georg Gottlob, 'Complexity results for disjunctive logic programming and application to nonmonotonic logics.', in *ILPS*, pp. 266–278. Citeseer, (1993).

[7] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner, 'Answer set programming: A primer', in *Reasoning Web. Semantic Technologies for Information Systems*, 40–110, Springer, (2009).

[8] Esra Erdem, *Theory and Applications of Answer Set Programming*, Ph.D. dissertation, 2002. AAI3101204.

[9] Timur Fayruzov, Martine De Cock, Chris Cornelis, and Dirk Vermeir, 'Modeling protein interaction networks with answer set programming', in *Bioinformatics and Biomedicine, 2009. IEEE International Conference on*, pp. 99–104. IEEE, (2009).

[10] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Schneider, 'Potassco: The Potsdam answer set solving collection', *AI Communications*, **24**(2), 107–124, (2011).

[11] Michael Gelfond and Vladimir Lifschitz, 'The stable model semantics for logic programming.', in *ICLP/SLP*, volume 88, pp. 1070–1080, (1988).

[12] Jeroen Janssen, Steven Schockaert, Dirk Vermeir, and Martine De Cock, 'General fuzzy answer set programs', in *Fuzzy Logic and Applications*, 352–359, Springer, (2009).

[13] Jeroen Janssen, Dirk Vermeir, Steven Schockaert, and Martine De Cock, 'Reducing fuzzy answer set programming to model finding in fuzzy logics', *Theory and Practice of Logic Programming*, **12**(6), 811–842, (2012).

[14] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello, 'The DLV system for knowledge representation and reasoning', *ACM Transactions on Computational Logic*, **7**(3), 499–562, (2006).

[15] Vladimir Lifschitz, 'What is answer set programming?.', in *AAAI*, volume 8, pp. 1594–1597, (2008).

[16] Fangzhen Lin and Yuting Zhao, 'ASSAT: Computing answer sets of a logic program by SAT solvers', *Artificial Intelligence*, **157**(1), 115–137, (2004).

[17] Daniele Mundici, 'Satisfiability in many-valued sentential logic is NP-complete', *Theoretical Computer Science*, **52**(1), 145–153, (1987).

[18] Steven Schockaert, Jeroen Janssen, and Dirk Vermeir, 'Fuzzy equilibrium logic: Declarative problem solving in continuous domains', *ACM Transactions on Computational Logic*, **13**(4), 33, (2012).

[19] Steven Schockaert, Jeroen Janssen, and Dirk Vermeir, 'Satisfiability checking in Łukasiewicz logic as finite constraint satisfaction', *Journal of Automated Reasoning*, **49**(4), 493–550, (2012).

[20] Davy Van Nieuwenborgh, Martine De Cock, and Dirk Vermeir, 'Fuzzy answer set programming', in *Logics in Artificial Intelligence*, 359–372, Springer, (2006).

[21] Davy Van Nieuwenborgh, Martine De Cock, and Dirk Vermeir, 'Computing fuzzy answer sets using DLVHEX', in *Logic Programming*, 449–450, Springer, (2007).