Solving disjunctive fuzzy answer set programs

Mushthofa Mushthofa[1,3], Steven Schockaert[2], and Martine De Cock[1,4]

[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: SchockaertS1@cardiff.ac.uk
[3] Department of Computer Science, Bogor Agricultural University, Indonesia, email: mush@ipb.ac.id
[4] Center for Data Science, University of Washington Tacoma, USA, email: mdecock@uw.edu

Abstract. Fuzzy Answer Set Programming (FASP) is an extension of the popular Answer Set Programming (ASP) paradigm which is tailored for continuous domains. Despite the existence of several prototype implementations, none of the existing solvers can handle disjunctive rules in a sound and efficient manner. We first show that a large class of disjunctive FASP programs called the *self-reinforcing cycle-free* (SRCF) programs can be polynomially reduced to normal FASP programs. We then introduce a general method for solving disjunctive FASP programs, which combines the proposed reduction with the use of mixed integer programming for minimality checking. We also report the result of the experimental benchmark of this method.

1 Introduction

Answer Set Programming (ASP) is one of the most popular and well-studied declarative programming paradigms [2, 11]. Based on the *stable model semantics* [15], ASP allows an intuitive encoding of combinatorial search and optimization problems [20]. Due to the availability of fast and efficient solvers, such as clasp [14] and DLV [19], ASP found practical applications in many fields [11, 13]. However, because of the fact that it relies on Boolean logic, ASP is not directly suitable for encoding problems in continuous domains.

Fuzzy Answer Set Programming (FASP) [30] is a form of declarative programming that extends classical ASP by allowing graded truth values in atomic propositions and extending classical Boolean operators to fuzzy logic connectives. Although work has done on the theoretical aspects of FASP, e.g., [6, 22–24, 5, 28, 18], progress on the development of FASP solvers has not yet reached the maturity level of ASP solvers. Several notable results about FASP are: (1) the development of a reduction method for FASP into bilevel linear programming [5], (2) a FASP solver based on answer set approximation operators [1] and (3) a solver for FASP based on a reduction to classical ASP [25]. The prototype solver developed in [1] only deals with normal rules and does not allow disjunctions at all. The method proposed by [5] only allows disjunctions in the head, while the evaluation method described in [25] only allows disjunctions in the body. No evaluation method/solver for FASP currently allows disjunctions in the body and the head of the rules.

Allowing disjunctions both in the head and the body of FASP rules enables us to represent and solve a broader class of problems. As an example, consider the following fuzzy graph colorability problem: given a graph $G = \langle V, E \rangle$ with weighted edges, can we color each node with a fuzzy color (intuitively, a grey level value between fully black and fully white) such that connected nodes are colored with *sufficiently* differing colors? Formally, let V be the set of the nodes and $E : V \times V \rightarrow [0, 1]$ be the set of fuzzy edges. We want to find functions $f_b : V \rightarrow [0, 1]$ and $f_w : V \rightarrow [0, 1]$ such that: (i) $f_b(x) \oplus f_w(x) = 1, \forall x \in V$ and (ii) $f(x) \otimes f(y) \otimes e(x, y) = 0, \forall x, y \in V, f \in \{f_b, f_w\}$, where $\oplus$ and $\otimes$ represent disjunction and conjunction in Łukasiewicz logic, respectively (see Section 2). Given the input graph as facts of the form $node(x) \leftarrow \overline{1}$ and $edge(x, y) \leftarrow \overline{c}$ for $c \in [0, 1]$[5], the following FASP program solves the problem (see Section 2 for formal definitions):

$$\begin{aligned} b(X) \oplus w(X) &\leftarrow node(X) \\ edge(X,Y) &\leftarrow edge(Y,X) \\ \overline{0} &\leftarrow b(X) \otimes b(Y) \otimes edge(X,Y) \\ \overline{0} &\leftarrow w(X) \otimes w(Y) \otimes edge(X,Y) \end{aligned}$$

Suppose we further require that for some subset $S \subseteq V$ of the nodes, the color assigned must be either fully black or fully white (i.e., either $f_w(v) = 1$ or $f_b(v) = 1$ for $v \in S$). We can encode this requirement by adding the so-called *saturation* rules $\{b(v) \leftarrow b(v) \oplus b(v), w(v) \leftarrow w(v) \oplus w(v) \mid v \in S\}$, which force the atoms $b(v)$ and $b(w)$ to be Boolean. In this example, we see how disjunctions in the head and in the body of the rules appear naturally in the problem's representation.

It is well known that the presence of disjunctions in (F)ASP programs can increase the computational complexity of various reasoning tasks. In [5], the complexity of answer set existence, set-membership and set-entailment in various classes of FASP under the Łukasiewicz semantics was studied. Interestingly, it was shown that for the class of *strict* FASP programs, where only the conjunction, maximum and negation operators are allowed in the body, disjunctions in the head do not increase the complexity of the reasoning tasks, which are still within the first level of the Polynomial Hierarchy (PH), even for disjunctive strict FASP. However, when disjunctions are allowed in the head as well as in the body, the complexity increases to the second level of PH.

In classical ASP, allowing disjunctions in the head has been shown to increase the complexity of the reasoning tasks, from NP-complete and coNP-complete for *brave* and *cautious* reasoning, respectively, to Σ_2^P and Π_2^P (see e.g., [7]). However, [3] has shown that the semantics of a large class of disjunctive ASP programs, called the *head-cycle free* (HCF) programs, can be efficiently expressed in non-disjunctive propositional logic, effectively reducing their complexity to the first level of PH. Furthermore, by a sequence of applications of the so-called *shift operators* described in [8], one can reduce any HCF disjunctive ASP program into an equivalent normal program.

In this paper, we show that a large class of disjunctive FASP programs can be similarly reduced to normal programs. Unlike in classical ASP, however, the reduction is not

[5] The symbol $\overline{c}$ for a numeric value c represents a truth-value constant in a program

based on head-cycle freeness, but rather on the concept we define as *self-reinforcing cycle freeness*, which covers a much larger subclass of disjunctive FASP programs. More specifically, we will show that the shifting method for HCF disjunctive programs as being used in classical ASP also applies for FASP programs. However, the class of disjunctive FASP programs that can be shifted is strictly larger than the class of HCF programs. In fact, we can show that every disjunctive strict FASP program can be reduced to a normal FASP program, even if head cycles occur. Subsequently, we introduce a general method for finding answer sets of disjunctive FASP programs (allowing disjunctions in the body, as well), which combines the proposed reduction with an additional minimality check based on mixed integer programming.

2 Preliminaries

2.1 Fuzzy answer set programming

We follow the definition of FASP syntax as described in [5] and consider only the Łukasiewicz operators and semantics. As is the case in classical ASP, the syntax of a FASP program $\mathcal{P}$ is based on atoms drawn from either a propositional or a first-order Herbrand base $\mathcal{B}_{\mathcal{P}}$. For simplicity, in this paper we consider only propositional atoms. A (classical) literal is either 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.

We denote by $Lit(E)$ the set of classical literals appearing in an expression E. A FASP program is a finite set of rules, where a rule r is of the form $\alpha \leftarrow \beta$. Here, α is a head expression (called the *head* of r) and β is a body expression (called the *body* of r). We also write *Head(r)* and *Body(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 only has one literal in the head is called a *normal* rule. A rule which contains the application of the operator $\oplus$ in the head is called a *disjunctive* rule. A FASP program is called [*positive*, *normal*] iff it contains only [positive, normal] rules, respectively. A FASP program which contains disjunctive rules is called a disjunctive program. A FASP program whose only connectives in the body are **not**, $\otimes$ and $\barwedge$, and has only disjunctions in the head is called a *strict* FASP program. Intuitively, the class of strict FASP programs contains those programs whose syntax corresponds to that of classical ASP programs. In this paper, we restrict the discussion to FASP syntax that allows only disjunction in the head (but no restrictions for the connectives in the body). Furthermore, following the rule rewriting technique described in [25], we may assume w.l.o.g. that the rules in a FASP program only contain at most one application of the Łukasiewicz connectives, either in the body or in the head of the rules.

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 $\mathcal{L}_k = \langle \mathbb{Q}_k, \leq \rangle$, where $\mathbb{Q}_k = \{\frac{i}{k} \mid 0 \leq i \leq k\}$, and $k \geq 1$ is a positive integer such that $c \in \mathbb{Q}_k$ for every constant c in the program. An interpretation of a FASP program $\mathcal{P}$ is a function $I : \mathcal{B}_\mathcal{P} \to \mathcal{L}$[6], which can be extended to to expressions and rules as follows:

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

for appropriate expressions α and β. Here, the operators **not**, $\otimes, \oplus, \veebar, \barwedge$ and $\leftarrow$ denote the Łukasiewicz negation, t-norm (conjunction), t-conorm (disjunction), 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 $I(Head(r)) \geq I(Body(r))$. An interpretation is a model of a program $\mathcal{P}$ iff it satisfies every rule in $\mathcal{P}$. 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 model I of $\mathcal{P}$ a *minimal* model if there is no other model J of $\mathcal{P}$ such that $J < I$.

For a positive FASP program $\mathcal{P}$, a model I of $\mathcal{P}$ is called an *answer set* of $\mathcal{P}$ iff it is a minimal model of $\mathcal{P}$. For a non-positive FASP program $\mathcal{P}$, a generalization of the so-called Gelfond-Lifschitz reduct is defined in [23] 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. an interpretation I is then defined as the positive program $\mathcal{P}^I = \{r^I \mid r \in \mathcal{P}\}$. A model I of $\mathcal{P}$ is called an answer set of $\mathcal{P}$ iff I is an answer set of $\mathcal{P}^I$. We say that an answer set I is k-valued if $I(a) \in \mathcal{L}_k$ for every literal a; we say that an answer set is finite valued if it is k-valued for some $k \in \mathbb{N}$.

Example 1 *Consider the disjunctive FASP program $\mathcal{P}_1$ which has the following rules:*

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

One can check that under both the truth-lattices $\mathcal{L}_3$ and $\mathcal{L}_\infty$, the interpretation $I_x = \{(a, \frac{1}{3}), (b, \frac{1}{3}), (c, \frac{2}{3}), (d, \frac{2}{3} - x), (e, x)\}$ is a minimal model of $\mathcal{P}_1^{I_x}$ for any $0 \leq x \leq \frac{2}{3}$, and hence it is an answer set of $\mathcal{P}_1$. However, $\mathcal{P}_1$ has no answer sets under any $\mathcal{L}_k$ where k is not a multiple of 3.

2.2 Finite-valued evaluation of FASP programs

We briefly recall the method from [25] to evaluate FASP programs based on a reduction to classical ASP. The method relies on a procedure $Tr(\cdot, k)$ that translates a FASP

[6] Note that by $\mathcal{L}$ here, we mean the "set" part of the lattice $\mathcal{L}$

program into a classical ASP program whose answer sets correspond to the k-valued answer sets of the original FASP program. For normal programs, every k-valued answer set of the program can be found by using the translation. Furthermore, any answer set obtained from the translation is guaranteed to be a k-valued answer set of the original FASP program.

For disjunctive FASP programs, however, the result does not necessarily hold, as illustrated in the next example.

Example 2 *Program $\mathcal{P}_2$ has the following rules: $\{a \oplus b \leftarrow \overline{1}, a \leftarrow b, b \leftarrow a\}$. The finite-valued answer set obtained by applying the translation method to $\mathcal{P}_2$ using $k = 1$ is $A_1 = \{(a, 1), (b, 1)\}$. However, A_1 is not an answer set of $\mathcal{P}_2$ in $\mathcal{L}_\infty$. In fact, the only answer set of $\mathcal{P}_2$ in $\mathcal{L}_\infty$ is $A_2 = \{(a, 0.5), (b, 0.5)\}$, which is obtained using the translation method when $k = 2$.*

To ensure that each k-valued answer set obtained is indeed an answer set of the FASP program, [25] suggested conducting an extra minimality check, which can however be costly. In this paper, we show how to reduce a large class of disjunctive FASP programs into normal programs, allowing us to avoid the minimality check.

3 Evaluating disjunctive rules

In this section we will identify a large fragment of the class of disjunctive FASP programs which can be reduced in polynomial time to a normal FASP program. Subsequently, we will show how this reduction can be used to develop a sound method for finding answer sets of general disjunctive FASP programs.

Following [3], the head-cycle free (HCF) ASP programs are programs whose positive dependency graphs (see Section 3.3) do not contain cycles that go through two literals occurring in the head of a rule. In [8], it was shown that any HCF program can be reduced to an equivalent program using the *shift* operator. Briefly, the shift operator replaces any rule $a_1 \vee \ldots \vee a_n \leftarrow B$ with the set of rules $R = \{a_i \leftarrow B \wedge NB_i \mid 1 \leq i \leq n\}$, where $NB_i = \bigwedge_{1 \leq j \leq n, j \neq i} \textbf{not } a_j$. For example, the program $\{a \vee b \leftarrow\}$ can be reduced to the equivalent program $\{a \leftarrow \textbf{not } b, b \leftarrow \textbf{not } a\}$. However, when we introduce head cycles, such as in the program $\mathcal{P}_3 = \{a \vee b \leftarrow, a \leftarrow b, b \leftarrow a\}$, shifting is no longer guaranteed to produce an equivalent normal program. Interestingly, in the case of FASP programs, the syntactically similar program $\mathcal{P}_4 = \{a \oplus b \leftarrow \overline{1}, a \leftarrow b, b \leftarrow a\}$ is equivalent to the shifted version: $\mathcal{P}_4' = \{a \leftarrow \textbf{not } b, b \leftarrow \textbf{not } a, a \leftarrow b, b \leftarrow a\}$. In fact, we will show that any strict disjunctive FASP program can be reduced to an equivalent normal FASP program in this way. This explains the observation in [5] that allowing disjunction in the head does not affect the computational complexity of strict FASP programs. For programs with disjunction in the body, shifting does not always yield an equivalent FASP program, for e.g., $\mathcal{P}_4 \cup \{a \leftarrow a \oplus a\}$ is not equivalent to $\mathcal{P}_4' \cup \{a \leftarrow a \oplus a\}$. Intuitively, we can safely shift disjunctive rules if there is no interaction between disjunctions in the body and a head cycle. We will now formalize this idea based on the notion of a self-reinforcing cycle.

3.1 SRCF programs

We first extend the notion of *proof* for classical disjunctive programs as defined in [3]. Let $\mathbf{0}$ denotes the interpretation that assigns zeros to all atoms. Let I be an interpretation of a program $\mathcal{P}$, and let a be any atom such that $I(a) > 0$. Then, a support of a in $\mathcal{P}$ w.r.t. I is defined as a sequence of rules $r_1, r_2, \ldots, r_n \in \mathcal{P}^I$ such that:

1. $\mathbf{0}(Body(r_1)) > 0$
2. $a \in Head(r_n)$
3. $\sum_{a \in Lit(Head(r_i))} I(a) = I(Body(r_i))$ for all $1 \leq i \leq n$
4. For every $x \in Lit(Body(r_i))$ there exists a $j < i$ such that $x \in Lit(Head(r_j))$

We characterize the non-existence of self-reinforcing cyclic rules in a program using the following definition: for a FASP program $\mathcal{P}$, we say that $\mathcal{P}$ is self-reinforcing cycle free (SRCF) w.r.t. an atom a, iff we can find a stratification function $f : \mathcal{B}_{\mathcal{P}} \to \mathbb{N}$, such that for every rule $r \in \mathcal{P}$ which contains a, it holds that:

1. $f(x) \geq f(y)$ for every $x \in Lit(Head(r))$ and $y \in Lit(Body(r))$
2. If $r \equiv x \leftarrow y \oplus z$, then $f(x) > f(y)$ and $f(x) > f(z)$.

Intuitively, we can see that that a program $\mathcal{P}$ is SRCF w.r.t. atom a iff the dependency graph does not contain any cycle which goes through a and involves at least one rule with disjunction in the body. We say that a program $\mathcal{P}$ is SRCF iff it is SRCF w.r.t every atom $a \in \mathcal{B}_{\mathcal{P}}$. The following theorem characterizes the notion of support for SRCF programs.

Theorem 1 (Support). *Let $\mathcal{P}$ be an SRCF program and let I be a consistent interpretation. Then I is an answer set of $\mathcal{P}$ iff:*

1. *I is a model of $\mathcal{P}$.*
2. *Every $a \in \{x \mid I(x) > 0\}$ has a support in $\mathcal{P}$ w.r.t. I.*

The proof runs parallel to the proof of Theorem 2.3 in [3] by noting that support plays a similar role for the answer sets of SRCF FASP programs as proof does for HCF ASP programs. Due to space constraints, we omit the details.

Example 3 *Consider program $\mathcal{P}_5 = \{a \oplus b \leftarrow \overline{1}, c \leftarrow b \otimes$ **not** $a, c \leftarrow a\}$. It is clear that $I = \{(a, 0.3), (b, 0.7), (c, 0.4)\}$ is an answer set of $\mathcal{P}_5$. In accordance with Theorem 1, for each of a, b and c, we can take $r_1 = a \oplus b \leftarrow \overline{1}, r_2 = c \leftarrow b \otimes \overline{0.7}$ as the support of these atoms in $\mathcal{P}_5$ w.r.t. I. Furthermore, any $J > I$ obtained by increasing the truth value of a, b or c will not have a support for that atom. On the other hand, the non-SRCF program $\mathcal{P}_6 = \{a \oplus b \leftarrow \overline{1}, a \leftarrow b, b \leftarrow a, a \leftarrow a \oplus a\}$ has only one answer set, namely $I = \{(a, 1), (b, 1)\}$. One can check that there is no support for each of a and b in $\mathcal{P}_6$ w.r.t. I, since in this case, $I(a) + I(b) > 1$.*

The following lemma holds in both classical and fuzzy ASP.

Lemma 1 (Locality). *Let $\mathcal{P}'$ be any subset of a program $\mathcal{P}$. If I is an answer set of $\mathcal{P}'$ and it satisfies all the rules in $\mathcal{P} - \mathcal{P}'$, then I is also an answer set of $\mathcal{P}$.*

Proof. Since I is an answer set of $\mathcal{P}'$ and I satisfies every rule of $\mathcal{P}-\mathcal{P}'$, I also satisfies every rule in $\mathcal{P}$. Then clearly, I satisfies $\mathcal{P}^I$ as well. Suppose that I is not the minimal model of $\mathcal{P}^I$, i.e., that there is another model $J < I$ of $\mathcal{P}^I$. Since $\mathcal{P}' \subseteq \mathcal{P}$ (and hence $\mathcal{P}'^I \subseteq \mathcal{P}^I$), it must also be the case that J satisfies $\mathcal{P}'^I$. But this means that I is not the minimal model of $\mathcal{P}'^I$, contradicting the assumption that I is an answer set of $\mathcal{P}'$.

We now present the main result for this section.

Theorem 2. *Let $\mathcal{P}_1 = \mathcal{P} \cup \{a \oplus b \leftarrow c\}$ be any SRCF program w.r.t. a, b and c. Then, an interpretation I is an answer set of $\mathcal{P}_1$ iff it is also an answer set of $\mathcal{P}_2 = \mathcal{P} \cup \{a \leftarrow c \otimes$ **not** $b, b \leftarrow c \otimes$ **not** $a\}$.*

Proof. (a) "If"-part: Let I be an answer set of $\mathcal{P}_2$. Then I is a minimal model of $\mathcal{P}^I \cup \{a \leftarrow c \otimes (\overline{1 - I(b)}), b \leftarrow c \otimes (\overline{1 - I(a)})\}$. Clearly, we have $I(a) + I(b) \geq I(c)$. We consider two cases:
   (i) $I(a) + I(b) = I(c)$. Let $p \in \{x \mid I(x) > 0\}$. By Theorem 1, there is a support R_p of p in $\mathcal{P}_2$ w.r.t I. If $\{a \leftarrow c \otimes (\overline{1 - I(b)}), b \leftarrow c \otimes (\overline{1 - I(a)})\} \cap R_p = \emptyset$, then we must have $R_p \subseteq \mathcal{P}^I$. This means that R_p is a support for p in $\mathcal{P}$ w.r.t I. On the other hand, if any (or both) of $\{a \leftarrow c \otimes (\overline{1 - I(b)}), b \leftarrow c \otimes (\overline{1 - I(a)})\}$ occurs in R_p, we can replace it (them) with the rule $a \oplus b \leftarrow c$, to obtain the set R'_p which can serve as a support for p in $\mathcal{P}_1$ w.r.t. I. In any case, each support R_p in $\mathcal{P}_2$ can be replaced with a support for p in $\mathcal{P}_1$. By Theorem 1, this means that every answer set of $\mathcal{P}_2$ is also an answer set of $\mathcal{P}_1$.
   (ii) $I(a) + I(b) > I(c)$. In this case, we have that $\{a \leftarrow c \otimes (\overline{1 - I(b)}), b \leftarrow c \otimes (\overline{1 - I(a)})\} \not\subseteq R_p$ for any support R_p of any $p \in \{x \mid I(x) > 0\}$, since it does not satisfy the first condition in the definition of support. Therefore, we have that $R_p \subseteq \mathcal{P}$ for any p, which, by Theorem 1 means that I is also an answer set of $\mathcal{P}$. Since I definitely satisfies $a \oplus b \leftarrow c$, by Lemma 1, I is also an answer set of $\mathcal{P}_1$.

(b) "Only if"-part: Similar to the previous part, let I be an answer set of $\mathcal{P}_1$. Then I is a minimal model of $\mathcal{P}^I \cup \{a \oplus b \leftarrow c\}$, and also $I(a) + I(b) \geq I(c)$. As before, we consider two cases:
   (i) $I(a) + I(b) = I(c)$. Let $p \in \{x \mid I(x) > 0\}$. By Theorem 1, there is a support R_p of p in $\mathcal{P}_2$ w.r.t I. If $a \oplus b \leftarrow c \notin R_p$, then we must have $R_p \subseteq \mathcal{P}^I$, which means that R_p is a support for p w.r.t. $\mathcal{P}$. On the other hand, if $a \oplus b \leftarrow c \in R_p$, we can replace it with the two rules $\{a \leftarrow c \otimes \overline{1 - I(b)}, b \leftarrow c \otimes \overline{1 - I(a)}\}$ to obtain the set R'_p which can serve as a support for p in $\mathcal{P}_2$ w.r.t I. In any case, each support R_p in $\mathcal{P}_1$ w.r.t. I can be replaced with a support for p in $\mathcal{P}_2$ w.r.t I. By Theorem 1, this means that every answer set of $\mathcal{P}_1$ is also an answer set of $\mathcal{P}_2$.
   (ii) $I(a) + I(b) > I(c)$. Similar to case (a)(ii), here we have that $a \oplus b \leftarrow c \notin R_p$ for any support R_p of any $p \in \{x \mid I(x) > 0\}$. Again, using Theorem 1, we get that every answer set of $\mathcal{P}_1$ is also an answer set of $\mathcal{P}_2$.

This result allows us to reduce an SRCF disjunctive FASP program to an equivalent normal program by performing the shifting operations, thus allowing the use of evaluation methods geared towards normal programs.

Example 4 *The program $\mathcal{P}_5$ with the following rules:*

$$\{a \oplus b \leftarrow \overline{1}, c \leftarrow b, c \leftarrow d \oplus e\}$$

is SRCF, since we can assign the stratification function $f(a) = f(b) = f(d) = f(e) = 1$ and $f(c) = 2$. Hence, by Theorem 2, it is equivalent to the normal program:

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

However, program $\mathcal{P}_5 \cup \{d \leftarrow c\}$ is not SRCF, and the shifting method does not work.

As a corollary of Theorem 2, any strict disjunctive FASP program can be reduced to a normal FASP program by shifting.

Example 5 *Consider program $\mathcal{P}_2$ from Example 2. It is a strict disjunctive FASP program, and hence it can be reduced to the equivalent normal program $\{a \leftarrow$ **not** $b, b \leftarrow$ **not** $a, a \leftarrow b, b \leftarrow a\}$.*

3.2 Non-SRCF programs

For non-SRCF programs, finding an answer set in $\mathcal{L}_\infty$ requires finding an answer set I in $\mathcal{L}_k$ for some $k \geq 1$, and checking whether I is also an answer set for $\mathcal{L}_\infty$. We show in this section how the last step can be implemented using Mixed Integer Programming (MIP). For some background on MIP, one can consult, e.g., [17] and [29].

In [16], a representation of infinitely-valued Łukasiewicz logic using MIP was proposed by defining a translation of each of the Łukasiewicz expressions $x \oplus y$, $x \otimes y$ and $\neg x$ into a set of MIP inequality constraints characterizing the value of each of the expressions. Given a FASP program $\mathcal{P}$ and an interpretation I, we can use the MIP representation of $\mathcal{P}$ (denoted as $MIP(\mathcal{P})$) based on the representations proposed by [16] to check whether I is the minimal model of $\mathcal{P}^I$, as follows:

1. For each atom a in $\mathcal{P}^I$, we use a MIP variable $v_a \in [0, 1]$ in $MIP(\mathcal{P})$ to express the truth value that a can take.
2. For any expression $e \in \{a \oplus b, a \otimes b, a \veebar b, a \barwedge b\}$ in any rule of $\mathcal{P}^I$, we create the appropriate set of constraints in MIP to represent the value of the expression, as suggested in [16]. For example, for $a \oplus b$, we have the following MIP constraints:

$$\begin{aligned} v_a + v_b + z_{a\oplus b} &\geq v_{a\oplus b} \\ v_a + v_b - z_{a\oplus b} &\leq v_{a\oplus b} \\ v_a + v_b - z_{a\oplus b} &\geq \ \ 0 \\ v_a + v_b - z_{a\oplus b} &\leq \ \ 1 \\ v_{a\oplus b} &\geq z_{a\oplus b} \end{aligned}$$

   In each case, z_e is a 0-1 variable and v_e is a variable representing the value of the expression e.
3. For each rule $\alpha \leftarrow \beta \in \mathcal{P}^I$, we add the constraint $v_\alpha \geq v_\beta$, where v_α and v_β are the variables corresponding to the values of the atoms/expressions α and β, respectively.

4. For each atom a, we add the constraint $v_a \leq I(a)$.
5. We set the objective function of the MIP program to minimise the value $\sum_{a \in \mathcal{B}_\mathcal{P}} v_a$.

Theorem 3. *The interpretation I is the minimal model of $\mathcal{P}^I$ iff the solution returned in $MIP(\mathcal{P})$ is equal to I.*

3.3 Incorporating program decomposition

While in practical applications FASP programs will not always be SRCF, often it will be possible to decompose programs such that many of the resulting components are SRCF. In this section, we show how we can apply the reduction from Section 3.1 to these individual components, and thus efficiently solve the overall program.

Program modularity and decomposition using dependency analysis have been extensively studied and implemented in classical ASP. In [21], the concept of splitting sets for decomposing an ASP program was introduced. Dependency analysis and program decomposition using strongly connected components (SCC) was described in [27] and [10], and has been used as a framework for efficient evaluation of logic programs, such as in [26, 12] and [9]. In this section, we build on this idea to develop a more efficient evaluation framework for FASP programs by exploiting the program's modularity/decomposability.

For a (ground) FASP program $\mathcal{P}$, consider a directed graph $G_\mathcal{P} = \langle V, E \rangle$, called the *dependency graph* of $\mathcal{P}$, defined as follows: (i) $V = \mathcal{B}_\mathcal{P}$ and (ii) $(a, b) \in E$ iff there exists a rule $r \in \mathcal{P}$ s.t. $a \in Lit(Body(r))$ and $b \in Lit(Head(r))$. By SCC analysis, we can decompose $G_\mathcal{P}$ into SCCs $C_1, \ldots, C_n$. With each SCC C_i, we associate a *program component* $PC_i \subseteq \mathcal{P}$, defined as the maximal set of rules such that for every $r \in PC_i$, the literals in PC_i are contained in C_i. The set of all program components of $\mathcal{P}$ is denoted as $PC(\mathcal{P})$. We define dependency between program components PC_i and PC_j as follows: PC_i depends on PC_j iff there is an atom a in PC_i and atom b in PC_j such that a depends on b in $G_\mathcal{P}$. The program component graph $C = \langle PC(\mathcal{P}), E_C \rangle$ is defined according to the dependency relation between the program components.

Similar to the case in classical ASP, the program component graph of a FASP program allows us to decompose the program into "modular components" that can be separately evaluated. For non-disjunctive components, the evaluation method described in [25] can be directly used. For SRCF disjunctive components, we can perform the shifting method as described in Section 3.1 to reduce the component into a normal program, and again use the evaluation method for normal programs. For non-SRCF disjunctive components, an extra minimality check as defined in Section 3.2 is needed after finding a k-answer set. Evaluation proceeds along the program components according to the topological sorting of the components in the program component graph, feeding the "partial answer sets" obtained from one component into the next. If a "complete answer set" is found, we stop. Otherwise, we backtrack to the previous component(s), obtaining k-answer sets for the next values of k until the stopping criterion is met.

Proposition 1. *Label an edge in $G_\mathcal{P}$ with the symbol $\oplus$ if the edge corresponds to a rule containing a disjunction in the body. A component is non-SRCF w.r.t. the atoms in that component iff there is a cycle in the component containing a labelled edge.*

Example 6 *Consider the program $\mathcal{P}_6$ containing the rules:*

$$\{a \leftarrow b \oplus c, b \leftarrow a \otimes \overline{0.5}, c \leftarrow \overline{0.7}, d \oplus e \leftarrow a\}$$

*Program $\mathcal{P}_6$ is not SRCF, hence we cannot directly use Theorem 2 to perform shifting. However, using SCC program decomposition, we obtain three components $PC_1 = \{a \leftarrow b \oplus c, b \leftarrow a \otimes \overline{0.5}\}$, $PC_2 = \{c \leftarrow \overline{0.7}\}$ and $PC_3 = \{d \oplus e \leftarrow a\}$. PC_1 only depends on PC_2, PC_2 has no dependencies, while PC_3 depends only on PC_1. Proceeding according to the topological order of the program components, we start by evaluating PC_2 and obtain the partial answer set $\{(c, 0.7)\}$. We feed this partial answer set into the next component PC_1. This program is normal and hence requires no minimality check associated to disjunctive programs. We obtain the partial answer set $\{(a, 1), (b, 0.5), (c, 0.7)\}$. The last component PC_3 is disjunctive, but it is also SRCF w.r.t. its atoms, and hence we can perform shifting to obtain the normal program $\{d \leftarrow a \otimes$ **not** $e, e \leftarrow a \otimes$ **not** $d\}$, which again can be evaluated without using minimality checks.*

4 Experimental benchmark

In this section, we experimentally evaluate the effectiveness of the proposed method. We implemented our method on top of the solver developed in [25][7]. We used clingo from the Potassco project [14] as the underlying ASP solver for finding k-answer sets, and the Coin-OR Cbc[8] solver as the MIP program solver for minimality checking.

For this benchmark, we used two problems: (1) the fuzzy graph colorability as given in the introduction, and (2) the fuzzy set covering problem, which is a generalization of the classical set covering problem, defined as follows: A fuzzy set F is defined as a function $F : U \to [0, 1]$, where U is the universe of discourse, and $F(u)$ for $u \in U$ is the degree of membership of u in F. A fuzzy subset S of F is a fuzzy set such that $S(u) \leq F(u), \forall u \in U$. Given a fuzzy set F and a collection of subsets $C = \{S_1, \ldots, S_n\}$ of F, the problem asks whether we can find a fuzzy sub-collection of C, such that every member of F is covered *sufficiently* by the subsets selected from C, and that the degree to which a subset S_i is selected is below a given threshold. We encode the problem in FASP as follows: the fuzzy set F is given by a set of facts of the form $f(x) \leftarrow \overline{a}$, the subsets S_i given by facts of the form $subset(s_i)$ and their membership degrees by $member(s_i, x) \leftarrow \overline{b}$. The maximum degree to which a subset S_i can be selected is denoted by a constraint $\overline{c} \leftarrow in(s_i)$. The following FASP program encodes the problem goal and constraints:

$$\begin{aligned} in(S) \oplus out(S) &\leftarrow subset(S) \\ covered(X) &\leftarrow (in(s_1) \otimes member(s_1, X)) \oplus \ldots \oplus (in(s_n) \otimes member(s_n, X)) \\ \overline{0} &\leftarrow f(X) \otimes \textbf{not } covered(X) \end{aligned}$$

For both benchmark problems, instances are generated randomly with no attempt to produce "hard" instances. Constant truth values for fuzzy facts (e.g., for edge weights)

[7] https://github.com/mushthofa/ffasp

[8] https://projects.coin-or.org/Cbc

are drawn randomly from the set $\mathbb{Q}_{10}$. Two types of instances are considered: (1) where no saturation rules are present, which means that the program is an SRCF program, and (2) where the saturation rules are added randomly with a 0.1 probability for each $b(x)$ and $w(x)$ atoms (in the graph coloring problem instances) and each $in(x)$ atoms (in the set covering problem instances). Since fuzzy answer set evaluation using finite-valued translation such as the one used in [25] cannot, in principle, be used to prove inconsistency, we opted to generate only instances that are known to be satisfiable.

To be able to see the advantage of applying our approach, we run the solver on all instances both with and without employing SRCF detection and shifting to reduce to normal programs. When SRCF detection is not employed, a minimality check has to be performed to verify that the answer sets obtained in any disjunctive component of the program are indeed minimal. Thus, our experiment will be useful to see the effectiveness of the proposed reduction over the baseline method of computing answer sets and checking for minimality.

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 2 minutes, while memory usage is limited to 1 GB. Table 1 presents the results of the experiment. Each value is an average over ten repeats.

Table 1. Values in the cells indicate average execution times (over ten instances) in seconds for the non-timed-out executions. Cells labeled with '(TO)' indicates that all executions of corresponding instances exceeded time/memory limit

problem	fuzzy graph coloring					fuzzy set cover				
saturation		no		yes			no		yes	
method		δ	σ	δ	σ		δ	σ	δ	σ
1	$n = 20$	2.9	1.7	2.7	1.8	$n = 10$	7.9	5.2	8.2	7.9
2	$n = 30$	6.6	3.8	6.1	3.8	$n = 15$	13.4	6.5	17.3	17.1
3	$n = 40$	10.6	5.7	10.7	6.1	$n = 20$	18.2	9.6	17.4	17.3
4	$n = 50$	19.8	11.5	23.0	11.0	$n = 25$	29.8	13.4	30.1	29.9
5	$n = 60$	34.8	17.7	36.0	20.4	$n = 30$	71.4	17.6	71.4	70.6
6	$n = 70$	53.4	25.4	55.3	28.2	$n = 35$	(TO)	22.3	(TO)	(TO)
7	$n = 80$	74.8	33.9	76.1	41.1	$n = 40$	(TO)	27.8	(TO)	(TO)

δ = no shifting, σ = with shifting applied

From the result, we can see that when SRCF detection and shifting are used, execution times are generally lower than when only minimality checks are used, even when saturation rules are present. This is especially true for the instances of the fuzzy graph coloring problem. The use of program decomposition/modularity analysis to separate program components that are SRCF from those that are non-SRCF can be beneficial since this means we can isolate the need for minimality checks to only those non-SRCF components, while the rest can be evaluated as normal programs after performing the shifting operation. For the set covering problem, we see no significant improvement in the running time when using the shifting operator for instances with saturation, one of the reasons being that the instances are such that minimality checks are needed regardless of what method is used (due to the fact that most components are non-SRCF). However, for the non-saturated instances, again we still see a clear advantage of using SRCF detection and shifting.

5 Conclusion

In this paper, we have identified a large class of disjunctive FASP programs, called SRCF programs, which can be efficiently evaluated by reducing them to equivalent normal FASP programs. We also proposed a method to perform a minimality check to determine the answer sets of non-SRCF programs based on a MIP formulation, and we showed how we can decompose FASP programs to further increase the applicability of our approach. We have implemented our approach on top of a current FASP solver, and integrated a MIP solver into the system to allow efficient minimality checking. To the best of our knowledge, our implementation represents the first FASP solver to allow evaluation rules with disjunctions in the body and/or in the head. We have also performed a benchmark testing of the proposed methods to measure their computational efficiency. Our result indicates that identifying SRCF components of a FASP program allows us to evaluate the program more efficiently.

References

1. Alviano, M., Peñaloza, R.: Fuzzy answer sets approximations. Theory and Practice of Logic Programming 13(4-5), 753–767 (2013)
2. Baral, C.: Knowledge representation, reasoning and declarative problem solving. Cambridge University Press (2003)
3. Ben-Eliyahu, R., Dechter, R.: Propositional semantics for disjunctive logic programs. Annals of Mathematics and Artificial intelligence 12(1-2), 53–87 (1994)
4. Blondeel, M., Schockaert, S., Vermeir, D., De Cock, M.: Fuzzy answer set programming: An introduction. In: Yager, R.R., Abbasov, A.M., Reformat, M.Z., Shahbazova, S.N. (eds.) Soft Computing: State of the Art Theory and Novel Applications, Studies in Fuzziness and Soft Computing, vol. 291, pp. 209–222. Springer Berlin Heidelberg (2013)
5. Blondeel, M., Schockaert, S., Vermeir, D., De Cock, M.: Complexity of fuzzy answer set programming under Łukasiewicz semantics. International Journal of Approximate Reasoning 55(9), 1971–2003 (2014)
6. Damásio, C.V., Pereira, L.M.: Antitonic logic programs. In: Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning, pp. 379–392 (2001)
7. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and expressive power of logic programming. ACM Computing Surveys 33(3), 374–425 (2001)
8. Dix, J., Gottlob, G., Marek, W.: Reducing disjunctive to non-disjunctive semantics by shift-operations. Fundamenta Informaticae 28(1), 87–100 (1996)
9. Eiter, T., Faber, W., Mushthofa, M.: Space efficient evaluation of ASP programs with bounded predicate arities. In: 24th AAAI Conference on Artificial Intelligence. pp. 303–308 (2010)
10. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive datalog. ACM Transactions on Database Systems 22(3), 364–418 (1997)

11. Eiter, T., Ianni, G., Krennwallner, T.: Answer set programming: A primer. In: Tessaris, S., Franconi, E., Eiter, T., Gutierrez, C., Handschuh, S., Rousset, M.C., Schmidt, R. (eds.) Reasoning Web. Semantic Technologies for Information Systems, Lecture Notes in Computer Science, vol. 5689, pp. 40–110. Springer Berlin Heidelberg (2009)
12. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: DLVHEX: A prover for semantic-web reasoning under the answer-set semantics. In: Proceedings of IEEE/WIC/ACM International Conference on Web Intelligence, 2006. pp. 1073–1074 (2006)
13. Erdem, E.: Theory and Applications of Answer Set Programming. Ph.D. thesis (2002), the University of Texas at Austin
14. Gebser, M., Kaufmann, B., Kaminski, R., Ostrowski, M., Schaub, T., Schneider, M.: Potassco: The Potsdam answer set solving collection. AI Communications 24(2), 107–124 (2011)
15. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proceedings of the Fifth International Conference and Symposium on Logic Programming. vol. 88, pp. 1070–1080 (1988)
16. Hähnle, R.: Proof theory of many-valued logic-linear optimization-logic design: connections and interactions. Soft Computing 1(3), 107–119 (1997)
17. Jeroslow, R.G.: Logic-based decision support: mixed integer model formulation. Elsevier (1989)
18. Lee, J., Wang, Y.: Stable models of fuzzy propositional formulas. In: Proceedings of the 14th European Conference on Logics in Artificial Intelligence, JELIA 2014, p. 326 (2014)
19. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Transactions on Computational Logic 7(3), 499–562 (2006)
20. Lifschitz, V.: What is answer set programming?. In: Proceedings of the 23rd AAAI Conference in Artificial Intelligence. vol. 8, pp. 1594–1597 (2008)
21. Lifschitz, V., Turner, H.: Splitting a logic program. In: Proceedings of the 11th International Conference on Logic Programming. pp. 23–37 (1994)
22. Loyer, Y., Straccia, U.: Epistemic foundation of stable model semantics. Theory and Practice of Logic Programming 6(4), 355–393 (2006)
23. Lukasiewicz, T., Straccia, U.: Tightly integrated fuzzy description logic programs under the answer set semantics for the semantic web. In: Proceedings of the 1st International Conference on Web Reasoning and Rule Systems, pp. 289–298 (2007)
24. Madrid, N., Ojeda-Aciego, M.: Measuring inconsistency in fuzzy answer set semantics. IEEE Transactions on Fuzzy Systems 19(4), 605–622 (2011)
25. Mushthofa, M., Schockaert, S., De Cock, M.: A finite-valued solver for disjunctive fuzzy answer set programs. In: Proceedings of European Conference in Artificial Intelligence 2014. pp. 645–650 (2014)
26. Oikarinen, E., Janhunen, T.: Achieving compositionality of the stable model semantics for smodels programs. Theory and Practice of Logic Programming 8(5-6), 717–761 (2008)
27. Ross, K.A.: Modular stratification and magic sets for datalog programs with negation. In: Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems. pp. 161–171 (1990)
28. Schockaert, S., Janssen, J., Vermeir, D.: Fuzzy equilibrium logic: Declarative problem solving in continuous domains. ACM Transactions on Computational Logic 13(4), 33:1–33:39 (2012)
29. Schrijver, A.: Theory of linear and integer programming. John Wiley & Sons (1998)
30. Van Nieuwenborgh, D., De Cock, M., Vermeir, D.: Fuzzy answer set programming. In: Proceedings of the 10th European Conference on Logics in Artificial Intelligence, pp. 359–372 (2006)