Gradient-Based Supported Model Computation in Vector Spaces

Akihiro Takemura[1,2], Katsumi Inoue[2,1]

[1]The Graduate University for Advanced Studies, SOKENDAI, Japan
[2]National Institute of Informatics, Japan.

Abstract

We propose an alternative method for computing supported models of normal logic programs in vector spaces using gradient information. To this end, we first translate the program into a definite program and embed the program into a matrix. We then define a loss function based on the implementation of the immediate consequence operator T_P by matrix-vector multiplication with a suitable thresholding function. We propose an almost everywhere differentiable alternative to the non-linear thresholding operation, and incorporate penalty terms into the loss function to avoid undesirable results. We report the results of several experiments where our method shows improvements over an existing method in terms of success rates of finding correct supported models of normal logic programs.

Keywords

Logic Programming, Supported Model Computation, Differentiable Logic Programming

1. Introduction

Performing logical inference with linear algebraic methods has been studied as an attractive alternative to traditional symbolic inference methods [1, 2, 3]. Linear algebraic approaches to logical inference offer promising characteristics compared to the traditional methods. For example, efficiency and scalability: since linear algebra is at the heart of many scientific applications, there are existing highly-efficient optimized algorithms for performing basic operations in linear algebra, and these new methods can benefit from high computational power offered in modern systems in the form of multi-core CPUs and GPUs. In this work, we present a gradient-based method for computing supported models in vector spaces.

Sakama et al. introduced the matrix representations of definite, disjunctive and normal logic programs, and a linear algebraic method for computing the least models and stable models [2]. More recently Nguyen et al. proposed an improved version of the algorithm by taking advantage of sparsity [4]. Sato et al. proposed a method for computing supported models in vector spaces via 3-valued completion models of a normal logic program [5]. The matrix encoding method used in this work is influenced by the aforementioned works, and we extend the previous works by proposing an encoding that is more suitable towards our goal of computing supported models as fixed points in vector spaces. Another difference is that while our method uses gradient

14th Workshop on Answer Set Programming and Other Computing Paradigms
atakemura@nii.ac.jp (A. Takemura); inoue@nii.ac.jp (K. Inoue)
0000-0003-4130-8311 (A. Takemura); 0000-0002-2717-9122 (K. Inoue)

information to find supported models, previous works use non-differentiable operations to find models.

Gradient-based methods have been considered, for example, for abduction [6], SAT and probabilistic inference [7]. Our design of the loss function follows the direction of previous works by Sato [8, 7], and we introduce additional terms for computing supported models. Although one should be able to compute supported models by obtaining a completion of the program and solving it with a SAT solver e.g. using the matrix-based SAT solver in [7], our method does not require completion nor SAT solver. Blair et al. introduced an extension of 2-valued semantics in vector spaces using a polynomial representation of normal programs, and used gradient descent to solve the dynamical system where the solutions correspond to the supported models [9]. In contrast we employ a matrix representation of the program, thresholding function and gradient descent for minimizing the loss function.

The closest to our work is Aspis et al.'s method [10], as both our algorithm and theirs use gradient-based method to find supported models of normal logic programs in vector spaces. Their method uses a matrix representation of the program reduct and the Newton's method for root finding to find fixed points. Some differences are: (i) While our method uses a normal to definite program translation and a matrix representation of the program, their method incorporates the notion of reduct into the program matrix. (ii) Our loss function includes a thresholding term and additionally regularization terms for dealing with fractional assignments and facts, while in their method the convergence depends solely on the fixed point criterion. The use of regularization to penalize fractional-valued interpretations is critical to avoid converging to unwanted middle points. Our goal in this paper is to improve upon Aspis et al.'s method, by extending Sato et al.'s earlier work on using loss functions for logical inference, and incorporating new terms for supported model computation.

The structure of this paper is as follows: Section 2 covers the necessary background and definitions. Section 3 presents a method for representing logic programs with matrices. Section 4 introduces the thresholding function and loss function, as well as the gradient-based search algorithm for finding supported models. Section 5 presents the results of experiments designed to test the ability of the algorithm and compare against known results in the literature. Finally, Section 6 presents the conclusion.

2. Background

We consider a language $\mathcal{L}$ that contains a finite set of propositional variables defined over a finite alphabet and the logical connectives $\neg$, $\wedge$, $\vee$ and $\leftarrow$. The *Herbrand base*, B_P, is the set of all propositional variables that appear in a logic program P.

A *definite program* is a set of *rules* of the form (1) or (2), where h and b_i are propositional variables (*atoms*) in $\mathcal{L}$. We refer to (2) as an *OR-rule*, which is a shorthand for m rules: $h \leftarrow b_1, h \leftarrow b_2, \ldots, h \leftarrow b_m$. For each rule r we define $head(r) = h$ and $body(r) = \{b_1, \ldots, b_m\}$. A rule r is referred to as a *fact* if $body(r) = \emptyset$.

$$h \leftarrow b_1 \wedge b_2 \wedge \cdots \wedge b_m (m \geq 0) \tag{1}$$

$$h \leftarrow b_1 \vee b_2 \vee \cdots \vee b_m (m \geq 0) \tag{2}$$

A *normal program* is a set of rules of the form (3) where h and b_i are propositional variables in $\mathcal{L}$.

$$h \leftarrow b_1 \wedge b_2 \wedge \cdots \wedge b_l \wedge \neg b_{l+1} \wedge \neg b_{l+2} \wedge \cdots \wedge \neg b_m (m \geq l \geq 0) \tag{3}$$

We refer to the positive and negative occurrences of atoms in the body as $body^+(r) = \{b_1, \ldots, b_l\}$ and $body^-(r) = \{b_{l+1}, \ldots, b_m\}$, respectively. A normal program is a definite program if $body^-(r) = \emptyset$ for every rule $r \in P$.

An *Herbrand interpretation* I, of a normal program P is a subset of B_P. A *model* M of P is an interpretation of P where for every rule $r \in P$ of the form (3), $body^+(r) \subseteq M$ and $body^-(r) \cap M = \emptyset$ imply $h \in M$. A program is called *consistent* if it has a model. A *supported model* M is a model of P where for every $p \in M$ there exists a rule $r \in P$ such that $p = h$, $body^+(r) \subseteq M$ and $body^-(r) \cap M = \emptyset$ [11, 12].

As we shall show later, in this paper we transform normal logic programs into definite programs for searching supported models. Thus we use the following definition of the immediate consequence operator T_P. $T_P : 2^{B_P} \rightarrow 2^{B_P}$ is a function on Herbrand interpretations. For a definite program P, we have: $T_P(I) = \{h | h \leftarrow b_1 \wedge \cdots \wedge b_m \in P \,\text{and}\, \{b_1, \ldots, b_m\} \subseteq I\} \cup \{h | h \leftarrow b_1 \vee \cdots \vee b_m \in P \,\text{and}\, \{b_1, \ldots, b_m\} \cap I \neq \emptyset\}$. It is known that a supported model M of a program P is a fixed point of T_P, i.e., $T_P(M) = M$[11].

Definition 1 (Singly-Defined (SD) program)**.** *A normal program P is called an SD program if* $head(r_1) \neq head(r_2)$ *for any two rules* r_1 *and* r_2 *in P where* $r_1 \neq r_2$.

Any normal program P can be converted into an SD-program P' in the following manner. If there are more than one rule with the same head ($h \leftarrow body(r_1), \ldots, h \leftarrow body(r_k)$, where $k > 1$), then replace them with a set of new rules $\{h \leftarrow b_1 \vee ... \vee b_k, b_1 \leftarrow body(r_1), ..., b_k \leftarrow body(r_k)\}$ containing new atoms $\{b_1, ..., b_k\}$. This is a stricter condition than the *multiple definitions condition* (*MD-condition*) [2]: for any two rules r_1 and r_2 in the program, $head(r_1) = head(r_2)$ implies $|body(r_1)| \leq 1$ and $|body(r_2)| \leq 1$. Consequently all SD-programs satisfy the MD condition. Unless noted otherwise, we assume all programs in this paper are SD-programs.

Given a normal program P, it is transformed into a definite program by replacing the negated literals in rules of the form (3) and rewriting:

$$h \leftarrow b_1 \wedge b_2 \wedge \cdots \wedge b_l \wedge \overline{b}_{l+1} \wedge \overline{b}_{l+2} \wedge \cdots \wedge \overline{b}_m (m \geq l \geq 0) \tag{4}$$

where $\overline{b}_i$ are new atoms associated with the negated b_i. A collection of rules of the form (4) is referred to as the *positive form* P^+ where $B_{P^+} = B_P \cup \{\overline{a} \,|\, a \in B_P\}$. For transformed rules of the form (4), we refer to $\{b_1, \ldots, b_l\}$ as the *positive part* and $\{\overline{b}_{l+1}, \ldots, \overline{b}_m\}$ as the *negative part*. After transformation, the program should contain rules of the forms (1), (2), or (4).

By an interpretation I^+ of P^+, we mean any set of atoms $I^+ \subseteq B_{P^+}$ that satisfies the condition for any atom $a \in B_{P^+}$, precisely one of either a or $\overline{a}$ belongs to I^+.

3. Representing Logic Programs with Matrices

In this section, we first describe the relationship between the positive forms of normal logic programs and its supported models. Then we describe the matrix encoding of the positive forms, which in turn represents the original normal logic programs in vector spaces.

3.1. Relationship between Positive Forms and Supported Models

Consider a (inconsistent) program $p \leftarrow \neg p$, and its positive form $p \leftarrow \overline{p}$. P^+ is a definite program but it has no models in this case due to the restriction we place on the interpretation: if $p \in I^+$ then $\overline{p} \notin I^+$ and vice versa. Then in this case, the implication is that there are no fixed points of T_{P^+} for P^+ that satisfy the condition $p \in I^+$ iff $\overline{p} \notin I^+$. On the other hand, when a model M of P exists, we can show that the corresponding M^+ is a model of P^+.

Proposition 1. *Let P be a normal program, and let P^+ be its positive form. If M is a model of P, then $M' = M \cup \{\overline{a} \mid a \in B_{P^+} \setminus M\}$ is a model of P^+. Conversely, if M^+ is a model of P^+, then $M^+ \cap B_P$ is a model of P.*

Proof. Follows from the definition of M' and M^+. Consider M'. Since $a \notin M'$ if $\overline{a} \in M'$ and vice versa, for each rule $r \in P^+$, $body(r) \subseteq M'$ implies $head(r) = a \in M'$. Thus, M' is a model of $P+$. Now consider M^+. Let $K = M^+ \cap B_P$. Given that M^+ is a model of P^+ and $a \in K$ if $a \in M^+$, for each rule $r \in P$, $body^+(r) \subseteq K$ and $body^-(r) \cap K = \emptyset$ implies $head(r) = a \in K$. Thus, K is a model of P. □

Proposition 2. *Let M be a supported model of P, and put $M' = M \cup \{\overline{a} \mid a \in B_{P^+} \setminus M\}$. Then, $T_{P^+}(M') = M$.*

Proof. Suppose $a \in M$. Since M is a supported model, there exists a rule $r \in P$ such that $head(r) = a$, $body^+(r) \subseteq M$ and $body^-(r) \cap M = \emptyset$. Correspondingly, there exists a rule $r' \in P^+$ such that $head(r') = a$, $body^+(r') \subseteq M'$ and $body^-(r') \subseteq M'$. That is, $a \in T_{P^+}(M')$. Hence, $M \subseteq T_{P^+}(M')$.

Conversely, suppose $a \in T_{P^+}(M')$. Then, there exists a rule $r' \in P^+$ such that $head(r') = a$ and $body(r') \subseteq M'$. Since M' is a model of P^+ by Proposition 1, $body(r') \subseteq M'$ implies $head(r') = a \in M'$. Because a is a positive literal, $a \in M$ holds. Hence, $T_{P^+}(M') \subseteq M$. Therefore, $M = T_{P^+}(M')$. □

Proposition 3. *Let M^+ be an interpretation of $P+$. If $T_{P^+}(M^+) = M^+ \cap B_P$ holds, then $M = M^+ \cap B_P$ is a supported model of P.*

Proof. Suppose $T_{P^+}(M^+) = M^+ \cap B_P$. Because $M^+ \cap B_P$ recovers the positive literals from M^+, for each $a \in (M^+ \cap B_P)$, there exists a rule $r \in P$ such that $head(r) = a$, $body^+(r) \subseteq (M^+ \cap B_P)$ and $body^-(r) \cap (M^+ \cap B_P) = \emptyset$. Thus, $M = M^+ \cap B_P$ is a supported model of P. □

3.2. Matrix Encoding of Logic Programs

We first introduce the matrix and vector notations used in this paper, before showing our method for encoding normal programs into matrices. Matrices and vectors are represented as bold upper-case e.g. $\mathbf{M}$ and lower-case letters e.g. $\mathbf{v}$, respectively. A 1-vector with length N is represented by $\mathbf{1}_N$. The indices of the entries of matrices and vectors appear in the subscript, for example, $\mathbf{M}_{ij}$ refers to the element at i-th row and j-th column of a matrix $\mathbf{M}$ and $\mathbf{v}_i$ refers to the i-th element of a column vector $\mathbf{v}$. Let $\mathbf{M}_{i:}$ and $\mathbf{M}_{:j}$ denote the i-th row slice and j-th column slice of $\mathbf{M}$, respectively. We denote the horizontal concatenation of matrices $\mathbf{M}_1$ and

$\mathbf{M}_2$ as $[\mathbf{M}_1\ \mathbf{M}_2]$, and denote the vertical concatenation of column vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ as $[\mathbf{v}_1; \mathbf{v}_2]$. Some matrices and vectors in this paper have a superscript to distinguish them from others, e.g. $\mathbf{M}^p$, and it should not be confused with indices appearing in the subscript; for example, $\mathbf{M}^p_{ij}$ refers to the i, j-th entry of $\mathbf{M}^p$.

Let P be a normal program with size $|B_P| = N$, P^+ its positive form and B_{P^+} the Herbrand base of P^+. Then we have $|B_{P^+}| = 2N$ since for every $b \in B_P$ we add its negated version $\overline{b}$. To encode P^+ into a program matrix $\mathbf{Q}$, we encode atoms appearing in the bodies of the rules with one-hot encoding. After performing this encoding for all rules in P^+, we obtain an $(N \times 2N)$ binary program matrix $\mathbf{Q}$, which is equivalent to a vertical concatenation of row vectors each encoding a rule $r \in P^+$ with one-hot encoding.

Definition 2 (Program Matrix)**.** *Let P be a normal program with $|B_P| = N$ and P^+ its positive form with $|B_{P^+}| = 2N$. Then P^+ is represented by a matrix $\mathbf{Q} \in \mathbb{Z}^{N \times 2N}$ such that for each element $\mathbf{Q}_{ij} (1 \leq i \leq N, 1 \leq j \leq 2N)$ in $\mathbf{Q}$,*

- *$\mathbf{Q}_{ij} = 1$ if atom $a_j \in B_{P^+} (1 \leq j \leq 2N)$ appears in the body of the rule $r_i (1 \leq i \leq N)$;*
- *$\mathbf{Q}_{ij} = 0$, otherwise.*

The i-th row of $\mathbf{Q}$ corresponds to the atom a_i appearing in the head of the rule r_i, and the j-th column corresponds to the atom $a_j (1 \leq j \leq 2N)$ appearing in the body of the rules $r_i (1 \leq i \leq N)$. Atoms that do not appear in the head of any of the rules in P^+ are encoded as zero-only row vectors in $\mathbf{Q}$.

This definition is different from [2] or [10], in that we do not explicitly include $\top$ and $\bot$ in the program matrix, and we do not use fractional values to encode long rules. In fact, our encoding method is similar to that of [5], except that we do not use $(2N \times 2N)$ space for the program matrix since we do not encode rules with $\overline{b} \in B_{P^+}$ in the head.[1]

Definition 3 (Interpretation Vector)**.** *Let P be a definite program and $B_P = \{a_1, \ldots, a_N\}$. Then an interpretation $I \subseteq B_P$ is represented by a vector $\mathbf{v} = (\mathbf{v}_1, \ldots, \mathbf{v}_N)^\intercal \in \mathbb{Z}^N$ where each element $\mathbf{v}_i\ (1 \leq i \leq N)$ represents the truth value of the proposition a_i such that $\mathbf{v}_i = 1$ if $a_i \in I$, otherwise $\mathbf{v}_i = 0$. We assume propositional variables share the common index such that $\mathbf{v}_i$ corresponds to a_i, and we write $\mathrm{var}(\mathbf{v}_i) = a_i$.*

Recall that the positive form P^+ of a normal program is a definite program, and all negated literals in the body are replaced by new atoms, e.g. in (4) $\neg b_{l+1}$ is replaced by $\overline{b}_{l+1}$. We now extend the definition of interpretation vectors to include relations between the positive and negative occurrences of atoms, in order to maintain whenever we have $b_1 \in I$, $\overline{b}_{l+1} \notin I$ and vice versa.

Definition 4 (Companion Vector)**.** *Let $B^P_{P^+} \subseteq B_{P^+}$ denote the positive part of P, $B^N_{P^+} \subseteq B_{P^+}$ denote the negative part of P, with size $|B^P_{P^+}| = |B^N_{P^+}| = N$. Let $\mathbf{v}^P \in \mathbb{Z}^N$ be a vector representing truth assignments for $a_i \in B^P_{P^+}$ such that $\mathbf{v}^P_i = 1$ if $a_i \in I$ and $\mathbf{v}^P_i = 0$ otherwise. Define a companion vector $\mathbf{w} \in \mathbb{Z}^{2N}$ representing an interpretation $I^+ \subseteq B_{P^+}$ as follows: $\mathbf{w} = [\mathbf{v}^P; \mathbf{1}_N - \mathbf{v}^P]$.*

[1] In [5], the program matrix encodes the completion of P^+.

4. Gradient Descent For Computing Supported Models

In this section, we first show how the evaluation of the T_P operator can be carried out in vector spaces. Then we introduce our loss function and its gradient, and finally we introduce an algorithm for finding supported models in vector spaces.

4.1. Computing the T_P operator in Vector Spaces

Sakama et al. [2] showed that the T_P operator can be computed in vector spaces using a thresholding function, which they called *θ-thresholding*. Here we introduce a variant of θ-thresholding to accommodate the new program encoding method as well as the differentiability requirement. First we initialize the parameter vector $\mathbf{t}$.

Definition 5 (Parameter Vector $\mathbf{t}$)**.** *A set of parameters to the θ-thresholding is represented by a column vector $\mathbf{t} \in \mathbb{Z}^N$ such that for each element $\mathbf{t}_i (1 \leq i \leq N)$ in $\mathbf{t}$,*

- $\mathbf{t}_i = |body(r_i)|$ *if the rule $r_i \in P^+$ is a conjunctive rule, e.g. (1), (4);*
- $\mathbf{t}_i = 1$ *if the rule $r_i \in P^+$ is a disjunctive rule e.g. (2);*
- $\mathbf{t}_i = 0$*, otherwise.*

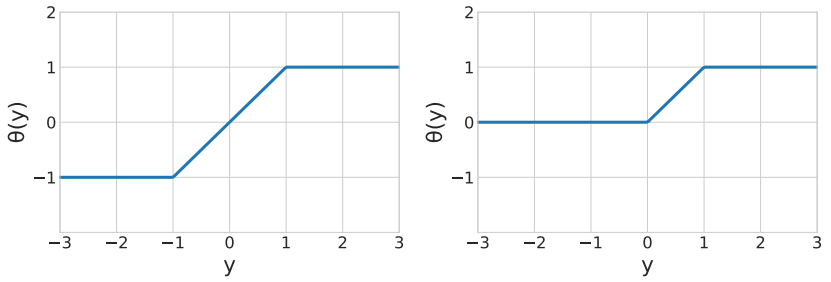
In some previous works [2, 13], the information about the nature of the rules was also stored in the program matrix $\mathbf{Q}$ alongside the atom occurrences; conjunctive rules with $|body(r_i)| > 1$ had fractional values $\mathbf{Q}_{ij} = 1/|body(r_i)|$ and disjunctive bodies had integer values $\mathbf{Q}_{ij} = 1$. Here we follow the approach taken by Sato et al. [5], and only store information about the occurrences of atoms in P^+ and keep supplementary information in the parameter vector $\mathbf{t}$ to recover the original program.

Definition 6 (Parameterized θ-thresholding)**.** *Let $\mathbf{w} \in \mathbb{Z}^{2N}$ be a companion vector representing $I^+ \subseteq B_{P^+}$. Given a parameter vector $\mathbf{t} \in \mathbb{Z}^N$, a program matrix $\mathbf{Q} \in \mathbb{Z}^{N \times 2N}$, and a vector $\mathbf{y} = \mathbf{Q}\mathbf{w}$ where $\mathbf{y} \in \mathbb{Z}^N$, we apply the thresholding function element-wise as follows:*

$$\theta_{\mathbf{t}}(\mathbf{y}_i) = \begin{cases} \min(\max(0, \mathbf{y}_i - (\mathbf{t}_i - 1)), 1) & (\mathbf{t}_i \geq 1) \\ 0 & (\mathbf{t}_i < 1) \end{cases} \tag{5}$$

This thresholding function resembles $hardtanh$ which is an activation function developed for use in natural language processing [14]. In the original $hardtanh$ function, the range of the linear region is $[-1, 1]$ (Figure 1(a)), but here we define the linear region between $[\mathbf{t}_i - 1, \mathbf{t}_i]$ (Figure 1(b)). This function is almost everywhere differentiable except at $\mathbf{y}_i = \mathbf{t}_i - 1$ and $\mathbf{y}_i = \mathbf{t}_i$. The special case $\mathbf{t}_i < 1$ in Equation (5) corresponds to the case $\mathbf{t}_i = 0$ where the head does not appear in the program P^+ and is assumed to be $false$.

Intuitively, for the head of a conjunctive rule to be $true$ it is sufficient to check whether all literals in the body hold, otherwise the rule evaluates to $false$. Similarly for a disjunctive rule it is sufficient to check whether at least one of the literals in the body holds for the head to hold. This is formalized in Proposition 4.



(a) $hardtanh$ (b) Parameterized thresholding with $\mathbf{t} = 1$

Figure 1: (a) $\theta(y)$ where $\theta = hardtanh$ (b) $\theta(y)$ where $\theta =$ parameterized θ-thresholding

Proposition 4 (Thresholded T_P Operator)**.** *Let P^+ be the positive form of a normal program P and $\mathbf{Q} \in \mathbb{Z}^{N \times 2N}$ its matrix representation. Let $\mathbf{v} \in \mathbb{Z}^N$ be a subset of an interpretation vector representing $a_i \in I^P \subseteq B^P_{P+}$ if $v_i = 1$ for $\{a_1, \ldots, a_N\}$. Let $\mathbf{w} \in \mathbb{Z}^{2N}$ be a companion vector to $\mathbf{v}$. Then $\mathbf{z} = [\mathbf{u}; 1 - \mathbf{u}] \in \mathbb{Z}^{2N}$ is a vector representing $J = T_P(I)$ satisfying the condition ($a \in J$ iff $\overline{a} \notin J$), iff $\mathbf{u} = \theta_{\mathbf{t}}(\mathbf{Qw})$.*

Proof. Consider $\mathbf{u} = \theta_{\mathbf{t}}(\mathbf{Qw})$. For $\mathbf{u} = (\mathbf{u}_1, \ldots, \mathbf{u}_N)^\intercal$, by the definition of the thresholding function, $\mathbf{u}_k = 1\,(1 \leq k \leq N)$ iff $\mathbf{u}'_k \geq \mathbf{t}_k$ in $\mathbf{u}' = \mathbf{Qw}$. Take a row slice $\mathbf{Q}_{k:}$, then $\mathbf{u}'_k = \mathbf{Q}_{k:}\mathbf{w} = \mathbf{Q}_{k1}\mathbf{w}_1 + \cdots + \mathbf{Q}_{k2N}\mathbf{w}_{2N}$, and $\mathbf{u}_k = 1$ iff $\mathbf{u}'_k \geq \mathbf{t}_k$. Both $\mathbf{Q}_{k:}$ and $\mathbf{w}$ are 0-1 vectors, then it follows that there are at least $\mathbf{t}_k$ elements where $\mathbf{Q}_{kj} = \mathbf{w}_j = 1\,(1 \leq j \leq 2N)$. The first N elements of $\mathbf{w}$ represent $a_i \in I^P \subseteq B^P_{P+}$ if $\mathbf{w}_i = 1$, and if $a_i \in I^P$ then $\overline{a}_i \notin I^N \subseteq B^N_{P+}$ which is maintained through the definition of the companion vector $\mathbf{w}$. 1) For a conjunctive rule $a_k \leftarrow a_1 \wedge \cdots \wedge a_m\,(1 \leq m \leq 2N)$, $\{a_1, \ldots, a_{2N}\} \in I$ implies $a_k \in T_P(I)$. 2) For an OR-rule $a_k \leftarrow a_1 \vee \cdots \vee a_m\,(1 \leq m \leq 2N)$, $\{a_1, \ldots, a_{2N}\} \subseteq I$ implies $a_k \in T_P(I)$. $a_m \in I$ is represented by $\mathbf{z}_m = 1\,(1 \leq m \leq 2N)$. Then by putting $J = \{\mathrm{var}(\mathbf{z}_m) | \mathbf{z}_m = 1\}$, $J = T_P(I)$ holds.

Consider $J = T_P(I)$. For $\mathbf{v} = (\mathbf{v}_1, \ldots, \mathbf{v}_N)^\intercal$ representing $I^P \subseteq B^P_{P+}$, $\mathbf{w} = (\mathbf{v}_1, \ldots, \mathbf{v}_N, 1 - \mathbf{v}_1, \ldots, 1 - \mathbf{v}_N)^\intercal$ is a vector representing $I \subseteq B_{P+}$ if we set $I = \{\mathrm{var}(\mathbf{w}_i) | \mathbf{w}_i = 1\}$. $\mathbf{u}' = \mathbf{Qw}$ is a vector such that $\mathbf{u}'_k \geq \mathbf{t}_k\,(1 \leq k \leq N)$ iff $\mathrm{var}(\mathbf{u}'_k) \in T_P(I)$. Define $\mathbf{u} = (\mathbf{u}_1, \ldots, \mathbf{u}_N)^\intercal$ such that $\mathbf{u}_k = 1\,(1 \leq k \leq N)$ iff $\mathbf{u}'_k \geq \mathbf{t}_k$ in $\mathbf{Qw}$, and $\mathbf{u}_k = 0$ otherwise. Define an interpretation $J \subseteq B_{P+}$ such that it can be partitioned into subsets of positive and negative occurrences of atoms $(J^P \cup J^N) = J \subseteq B_{P+}$. Since only positive atoms occur in the head, $\mathbf{u}$ represents a positive subset of interpretation $J^P \subseteq J \subseteq B_{P+}$ by setting $J^P = \{\,\mathrm{var}(\mathbf{u}_i) | \mathbf{u}_i = 1\}\,(1 \leq i \leq N)$. If $a_i \in T_P(I)$ then $\mathbf{u}_i = 1$, and $\overline{a}_i \notin T_P(I)$ is represented by $1 - \mathbf{u}_i = 0$. Conversely, if $a_i \notin T_P(I)$ then $\mathbf{u}_i = 0$, and $1 - \mathbf{u}_i = 1$ represents $\overline{a}_i \in T_P(I)$. Thus $1 - \mathbf{u}$ represents $J^N \subseteq J \subseteq B_{P+}$. $\mathbf{z} = [\mathbf{u}; 1 - \mathbf{u}]$ is then a vector representing $J^P \cup J^N = J$ if we set $J = \{\mathrm{var}(\mathbf{z}_m) | \mathbf{z}_m = 1\}\,(1 \leq m \leq 2N)$. Thus $\mathbf{z} = [\mathbf{u}; 1 - \mathbf{u}]$ represents $J = T_P(I)$ if $\mathbf{u} = \theta_{\mathbf{t}}(\mathbf{Qw})$.

□

Example 1. *Consider the following program:*

$$p \leftarrow q$$

$$q \leftarrow p \wedge r$$
$$r \leftarrow \neg p \tag{6}$$

This program has one supported (stable) model: $\{r\}$. *We have* $B_P = \{p, q, r\}$ *and* $B_{P^+} = \{p, q, r, \overline{p}, \overline{q}, \overline{r}\}$ *and the matrix representation* $\mathbf{Q}$ *and parameter vector* $\mathbf{t}$ *are:*

$$\mathbf{Q} = \begin{array}{c} \\ p \\ q \\ r \end{array} \begin{array}{c} \begin{array}{cccccc} p & q & r & \overline{p} & \overline{q} & \overline{r} \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \end{array} \quad \mathbf{t} = \begin{array}{c} \\ p \\ q \\ r \end{array} \begin{array}{c} \\ \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix} \end{array} \tag{7}$$

Suppose an assignment $\mathbf{v}^{\{r\}} = (0\ 0\ 1)^\intercal$ *is given. Then the companion vector* $\mathbf{w}$ *is:*

$$\mathbf{w} = [\mathbf{v}^{\{r\}}; \mathbf{1}_3 - \mathbf{v}^{\{r\}}] = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}^\intercal \tag{8}$$

Compute the matrix multiplication product $\mathbf{Qw}$ *and apply the parameterized thresholding:*

$$\mathbf{u} = \theta_{\mathbf{t}}(\mathbf{Qw}) = \theta_{\mathbf{t}}(\begin{pmatrix} 0 & 1 & 1 \end{pmatrix}^\intercal) = \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}^\intercal = \mathbf{v}^{\{r\}} \tag{9}$$

Let $\mathbf{z}$ *be a companion vector to* $\mathbf{u}$*, i.e.* $\mathbf{z} = [\mathbf{u}; 1 - \mathbf{u}]$*, then we have*

$$\mathbf{z} = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}^\intercal \tag{10}$$

Define $J = \{\mathrm{var}(\mathbf{z}_m) | \mathbf{z}_m = 1\}$*, then we have* $J = \{r, \overline{p}, \overline{q}\}$*, and* $J \cap B_P = \{r\}$.

Proposition 5 (Supported Model Computation with Thresholded T_P)**.** *Let* $\mathbf{v} \in \mathbb{Z}^N$ *be a 0-1 vector representing a subset of interpretation* $I^P \subseteq I \subseteq B_{P^+}$*, and* $\mathbf{z} = [\mathbf{v}; \mathbf{1}_N - \mathbf{v}]$ *be its companion vector representing* $I \subseteq B_{P^+}$ *satisfying* ($a \in I$ *iff* $\overline{a} \notin I$)*. Given a program matrix* $\mathbf{Q}$ *representing a program* P^+ *and a thresholding function* $\theta_{\mathbf{t}}$ *parameterized by a vector* $\mathbf{t}$*, the fixed points of* P^+ *are represented by 0-1 binary vectors* $\mathbf{z}^{FP} = [\mathbf{v}^{FP}; \mathbf{1}_N - \mathbf{v}^{FP}] \in \mathbb{Z}^{2N}$ *where* $\mathbf{v}^{FP} = \theta_{\mathbf{t}}(\mathbf{Qz}^{FP})$*. Then* $\mathbf{z}^{FP}$ *are vectors representing models* M^+ *of* P^+ *satisfying* ($a \in M^+$ *iff* $\overline{a} \notin M^+$) *iff* $\theta_{\mathbf{t}}(\mathbf{Qz}^{FP}) = \mathbf{v}^{FP}$*. When such 0-1 binary vector* $\mathbf{z}^{FP}$ *exists,* $M^+ \cap B_P = M$ *is a supported model of* P.

Proof. Let $I \subseteq B_{P^+}$ be a model of P^+, represented by $\mathbf{z}^{FP}$. Consider two cases (i) $T_P(I) = I$ and (ii) $\mathbf{v}^{FP} = \theta_{\mathbf{t}}(\mathbf{Qz}^{FP})$. In both cases, by Propositions 2, 3 and 4, if a supported model of P exists, the results hold.

□

Example 2. *Consider Program (6) in Example 1, and interpretation vector* $\mathbf{z}$ *(10) representing an interpretation* J*. We show that* $J = \{r, \overline{p}, \overline{q}\}$ *is indeed a model of the positive form, and corresponds to the supported model of* P*. Construct a vector* $\mathbf{v}^{\{r\}}$ *and companion vector* $\mathbf{w}$ *as in (8). By comparing (8) and (10), we see that* $\mathbf{w} = \mathbf{z}$ *and this is consistent with the fixed point definition* $J = T_P(J)$*.* $J \cap B_P$ *results in* $\{r\}$ *which is the supported model of* P.

4.2. Loss Function for Computing Supported Models

By the fixed point definition of supported models, a supported model M satisfies $\mathbf{v}^{M^P} = \theta_{\mathbf{t}}(\mathbf{Q}[\mathbf{v}^{M^P}; \mathbf{1}_N - \mathbf{v}^{M^P}])$. We now use this definition to design a loss function which can be minimized using gradient descent. Gradient descent is a method for minimizing an objective function (loss function), by updating the parameters in the opposite direction of the gradient of the objective function with respect to the parameters. The size of the update is determined by the gradient and the step size α. Intuitively this algorithm follows the direction of the slope of the loss surface until it reaches the bottom of a valley (local minimum).

Let $\mathbf{u}$ be an $(N \times 1)$ vector representing an assignment to the heads of the rules in P^+. Define $\mathbf{w} = [\mathbf{u}; \mathbf{1}_N - \mathbf{u}]$ as the $(2N \times 1)$ interpretation vector that stores assignments to each atom $a \in M \subseteq B_{P^+}$. We define an $(N \times 1)$ vector $\mathbf{f}$ which stores information about occurrences of facts in the program P^+. This vector will be used later during the minimization step to ensure that facts are not forgotten.

Definition 7 (Fact Vector $\mathbf{f}$)**.** *The set of facts in the program P^+ is represented by a column vector $\mathbf{f} \in \mathbb{Z}^N$ such that for each element $\mathbf{f}_i (1 \leq i \leq N)$ in $\mathbf{f}$,*

- $\mathbf{f}_i = 1$ *if the rule r_i is a fact: $a \leftarrow$*
- $\mathbf{f}_i = 0$ *otherwise.*

Definition 8 (Loss Function)**.** *Given a program matrix $\mathbf{Q}$, a candidate vector $\mathbf{x}$, thresholding function $\theta_{\mathbf{t}}$, and constants λ_1 and λ_2, define the loss function as follows:*

$$L(\mathbf{x}) = \frac{1}{2}\Big(\|\theta_{\mathbf{t}}(\mathbf{Q}[\mathbf{x}; \mathbf{1}_N - \mathbf{x}]) - \mathbf{x}\|_{\mathrm{F}}^2 + \lambda_1 \|\mathbf{x} \odot (\mathbf{x} - \mathbf{1}_N)\|_{\mathrm{F}}^2 + \lambda_2 \|\mathbf{f} - (\mathbf{x} \odot \mathbf{f})\|_{\mathrm{F}}^2 \Big) \tag{11}$$

where $\|\mathbf{x}\|_{\mathrm{F}}$ denotes the Frobenius norm and $\odot$ element-wise product.

The first term is derived directly from the fixed point definition of supported models, and should be 0 if $\mathbf{x}$ is a supported model of P^+. The second term, which resembles a regularization term often used in the machine learning literature, is added to penalize candidate vectors $\mathbf{x}$ that contain fractional values, and is 0 if and only if $\mathbf{x}$ is a 0-1 vector. The third term will be 0 if and only if the facts are preserved, and will be positive non-zero if any part of the assignment is lost, i.e. by assigning 0 ($false$) to a fact where $\mathbf{f}_i = 1$.

We introduce submatrices of $\mathbf{Q}$, $\mathbf{Q}_p$ and $\mathbf{Q}_n$ that correspond to the positive bodies and negative bodies of the matrix, respectively, such that $\mathbf{Q} = [\mathbf{Q}_p\, \mathbf{Q}_n]$ (horizontal concatenation of submatrices). Both $\mathbf{Q}_p$ and $\mathbf{Q}_n$ have the shape $(N \times N)$.

Definition 9 (Gradient of the Loss Function)**.** *The gradient of the loss function with respect to $\mathbf{x}$ is given by:*

$$\begin{aligned} \frac{\partial L(\mathbf{x})}{\partial \mathbf{x}} = & \left((\mathbf{Q}_p - \mathbf{Q}_n)^T \cdot \theta_{\mathbf{t}}(\mathbf{Q}\mathbf{z}_{\mathbf{x}}) \odot \frac{\partial \theta_{\mathbf{t}}(\mathbf{Q}\mathbf{z}_{\mathbf{x}})}{\partial \mathbf{x}} \right) - \theta_{\mathbf{t}}(\mathbf{Q}\mathbf{z}_{\mathbf{x}} - \mathbf{x}) \\ & + \lambda_1(\mathbf{x} \odot (\mathbf{1}_N - \mathbf{x}) \odot (\mathbf{1}_N - 2\mathbf{x})) + \lambda_2(\mathbf{x} \odot \mathbf{f} - \mathbf{f}) \end{aligned} \tag{12}$$

where $\mathbf{z}_{\mathbf{x}} \in \mathbb{R}^{2N} = [\mathbf{x}; \mathbf{1}_N - \mathbf{x}]$ and

$$\frac{\partial \theta_{\mathbf{t}}(\mathbf{w}_i)}{\partial \mathbf{x}_i} = \begin{cases} 1 & \text{if } (\mathbf{t}_i \geq 1) \text{ and } (\mathbf{t}_i - 1) \leq \mathbf{w}_i \leq \mathbf{t}_i \\ 0 & \text{otherwise} \end{cases} \tag{13}$$

We can update $\mathbf{x}$ iteratively using, for example, gradient descent or quasi-Newton's method, to reduce the loss to 0. Here we shown an example of update rule for gradient descent. Let α be the step size, then the gradient descent update rule is given by:

$$\mathbf{x}_{\text{new}} \leftarrow \mathbf{x} - \alpha \frac{\partial L(\mathbf{x})}{\partial \mathbf{x}} \tag{14}$$

Using this update rule we can design an algorithm to find supported models, as shown in Algorithm 1.

The convergence characteristics of the gradient descent algorithm are well-known [15]. Assuming at least one 0-1 vector representing a supported model exists for $\mathbf{Q}$, all we require for Algorithm 1 to converge to the supported model is that the initial vector $\mathbf{x}_{\text{init}}$ to be in the region surrounding the supported model where the slope points towards the model. When there are multiple supported models (correspondingly multiple local minima), we expect the algorithm to converge to different models depending on the choice of initial vector $\mathbf{x}_{\text{init}}$. However, it is often not known *apriori* which particular values or regions of $\mathbf{x}_{\text{init}}$ lead to which models. Therefore we implement a uniform sampling strategy, where we sample uniformly from $[0, 1]$, and make no assumptions with regard to the existence of supported models for $\mathbf{Q}$.

Depending on the program, an optimal 0-1 vector interpretation may not exist, so we limit the number of iterations to $\mathrm{max\_iter}$ before assuming non-convergence. With gradient descent, it is often time consuming to reduce the loss function completely to zero. We therefore implement a "peeking at a solution" heuristic, similar to the one presented in [7], where while updating $\mathbf{x}$ we round $\mathbf{x}$ a to 0-1 vector to see whether the resulting $\mathbf{x}_r$ is a solution (Lines 7-11). The output is sensitive to the choice of initial vector $\mathbf{x}_{\text{init}}$, and a poor choice of $\mathbf{x}_{\text{init}}$ may result in non-convergence to optimal solutions. We alleviate this dependency on the initial vector by introducing the $\mathrm{max\_retry}$ parameter and changing the initial vector on each try. This algorithm declares failure to find any supported models (returns FALSE) when both $\mathrm{max\_retry}$ and $\mathrm{max\_iter}$ are exhausted.

An initial implementation of the proposed method is made using Python 3.7. Our experimental environment is a Linux container limited to 8 virtual CPU cores and 16GB RAM hosted on a desktop with Ubuntu 18.04, Intel Core i9-9900K 3.6GHz and 64GB RAM.

5. Experiments

In this section, we first compare the performance of our method with the one presented by Aspis et al. [10] in terms of success rate of finding correct supported models. Then we qualitatively show the effect of the rounding heuristic by plotting the trajectories of the candidate interpretation vector.

5.1. Comparing Success Rates with Aspis et al.

Aspis et al. [10] encode the program reduct into a matrix and employ the Netwon's method for root finding to find fixed points of the program. The gradient is calculated by a Jacobian matrix, and the thresholding operation is carried out with a parameterized sigmoid. They present two types of sampling methods for setting the initial vector; *uniform sampling*, similarly to our

Algorithm 1 Gradient descent search of supported models

Input: Program matrix $\mathbf{Q}$, Thresholding parameter $\mathbf{t}$, Initial vector $\mathbf{x}_{\text{init}}$, $\text{max\_retry} \geq 1$, $\text{max\_iter} \geq 1$, $\epsilon > 0$, step size $\alpha > 0$, $\lambda_1 > 0$, $\lambda_2 > 0$
Output: supported model $\mathbf{x}$ or FALSE

$\mathbf{x} \leftarrow \mathbf{x}_{\text{init}}$
for n_try $\leftarrow 1$ **to** max_retry **do**
  if n_try > 1 **then**
    $\mathbf{x} \leftarrow$ random vector
  end if
  for n_iter $\leftarrow 1$ **to** max_iter **do**
    $\mathbf{x}_r \leftarrow \text{round}(\mathbf{x})$ ▷ Rounding heuristic
    loss $\leftarrow L(\mathbf{x}_r)$ ▷ Loss function, see Def. (8)
    if (loss $\leq \epsilon$) **then**
      $\mathbf{x} \leftarrow \mathbf{x}_r$
      return $\mathbf{x}$
    else
      gradient $\leftarrow \frac{\partial L(\mathbf{x})}{\partial \mathbf{x}}$ ▷ Gradient, see Def. (9)
      $\mathbf{x} \leftarrow \mathbf{x} - \alpha \cdot$ gradient ▷ Gradient update
    end if
  end for
end for
return FALSE

method, where the values are sampled uniformly from $[0, 1]$, and *semantic sampling*[2], where the values are sampled uniformly from $\left[0, \gamma^{\perp}\right] \cup \left[\gamma^{\top}, 1\right]$. Semantic sampling results in an initial vector that is semantically equivalent to an interpretation.

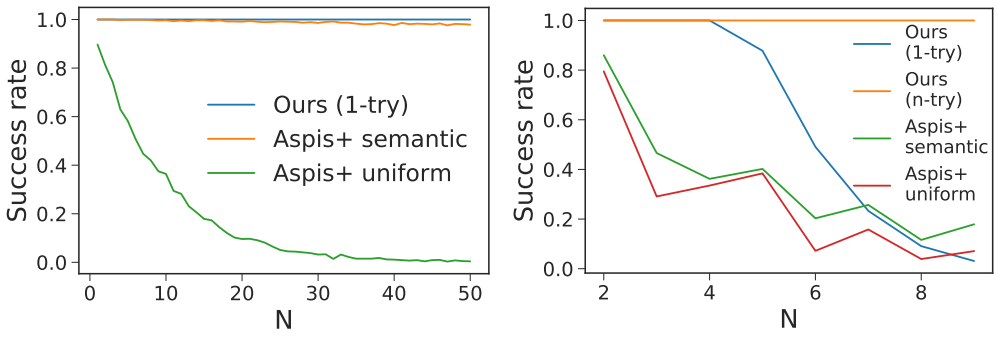
Firstly we consider the "N-negative loops" task, which involves programs in the following form: for $1 \leq i \leq N$,

$$\begin{aligned} p_i &\leftarrow \text{not } q_i \\ q_i &\leftarrow \text{not } p_i \end{aligned} \tag{15}$$

For our algorithm, we disabled automatic retry with $\text{max\_retry} = 1$, so that the result depends solely on a single application of the algorithm. Other parameters were set as follows: $\text{max\_iter} = 10^3$, $\epsilon = 10^{-4}$, $\lambda_1 = \lambda_2 = 1$, $\alpha = 10^{-1}$. For Aspis et al.'s algorithm, we used the following settings: $\text{max\_iter} = 10^3$, $\epsilon = 10^{-4}$, $\gamma = 0.5$, $\tau = 0.087$.[3] We generated programs of the form Program (15) with N up to 50, and then applied the algorithms exactly once for 1000 times. We measured the rate of success of converging to supported models. Figure 2(a) shows our method with uniform initialization outperforms their uniform sampling method, although improvement over their semantic sampling method is marginal.

Secondly we consider the "choose 1 out of N" task, where the task is choose exactly 1 out of

[2] $\gamma^{\perp}$ is an upper bound on false values that variables can take, and $\gamma^{\top}$ is a lower bound on true values.
[3] We used $\tau = 0.087$ which was their best performing setting resulting in 89% success and near 100% success for uniform and semantic sampling, respectively, for $N = 1$ loop.



(a) N-negative loops (b) N-choices

Figure 2: (a) Success rate of converging to a model at the N-negative loops task. (b) Success rate of converging to a model at the N-choices task.

N options. The programs in this class have the following form:

$$\begin{aligned} p_1 &\leftarrow \text{not}\, p_2,\ \text{not}\, p_3,\ ,..,\ \text{not}\, p_N \\ p_2 &\leftarrow \text{not}\, p_1,\ \text{not}\, p_3,\ ,..,\ \text{not}\, p_N \\ &\vdots \\ p_N &\leftarrow \text{not}\, p_1,\ \text{not}\, p_2,\ ,..,\ \text{not}\, p_{N-1} \end{aligned} \tag{16}$$

We generated programs for N between 2 and 9, and applied the algorithms using the same parameters as the "N-negative loops" task, and repeated the process for 1000 times for each N.[4] For our algorithm, we consider the following scenarios: when the algorithm is allowed (i) only 1 try ($\text{max\_retry} = 1$, **1-try** in Fig. 2(b)), and (ii) retry up to 1000 times ($\text{max\_retry} = 1000$, **n-try** in Fig. 2(b)).

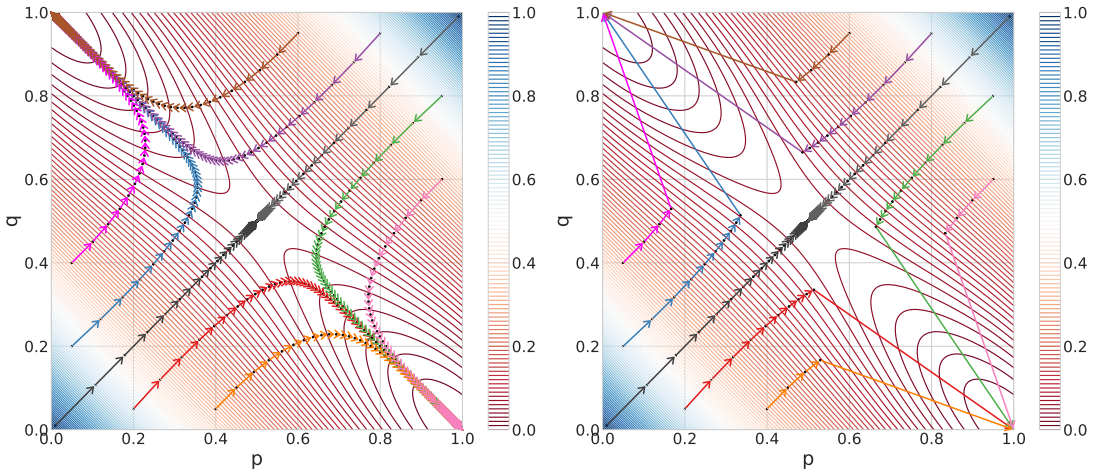
From Figure 2(b), we observe that the success rate of our algorithm (1-try) starts to drop rapidly past $N = 5$, and approaches the same level of success rate (10-20% range) as Aspis et al. by $N = 7$. We can allow multiple retries to help the algorithm find the supported models, as indicated by the high success rate of the n-try case, however this means that potentially the run time will be significantly longer compared to $\text{max\_retry} = 1$. In this example, there is a global constraint that exactly one of $\{p_1, ..., p_N\}$ is chosen, but neither our method nor Aspis et al.'s method can capitalize on this information. Handling of global constraints is left for future work.

5.2. Effect of the Rounding Heuristic on the Trajectories

Here we qualitatively demonstrate the effect of the rounding heuristic (Lines 7-11 in Algorithm 1) by plotting the trajectories of the interpretation vector. We consider a simple negative loop program, i.e., program (15) with $i = 1$, where we have exactly two supported models $\{p\}$ and $\{q\}$. The contours in the following figures represent the value of the loss function at each

[4]For each program whose maximum rule length is $N - 1$, we estimate the upper bound of τ according to their method, and subtract 10^{-3} from this value and use it as τ. This ensures that τ is smaller than the upper bound.

(p, q)-coordinate. The dots and arrows represent the trajectories of the algorithm starting from various initial vectors. Note that the lowest points on this landscape are at $(p, q) = (1, 0)$ ($\{p\}$) and $(p, q) = (0, 1)$ ($\{q\}$).



(a) Trajectories of various attempts without rounding. (b) Trajectories of various attempts with rounding.

Figure 3: (a) Trajectories without rounding (b) Trajectories with rounding heuristic.

From Figure 3(a) we observe that once the algorithm reaches the bottom of the valley, the update becomes smaller and the convergence is slow. From Figure 3(b) we see that the rounding heuristic (peeking at solution) avoids the bottom of the valley entirely and after several iterations the algorithm jumps straight to the 0-1 interpretation, which makes it converge faster.

In both cases we see the algorithm converging to the middle point $\{p = 0.5, q = 0.5\}$ when starting from initial vectors with $(p = q)$ Currently we do not break this symmetry in the algorithm, for example, by adding a very small noise to the vector during update, and instead use a retry function which resets the initial vectors to a random value on restart.

6. Conclusion

In this work we presented a new method for computing supported models of a normal logic program, using a matrix representation of the program and a gradient-based search algorithm. We defined a loss function based on the implementation of the T_P-operator by matrix-vector multiplication with a suitable thresholding function. We incorporated penalty terms into the loss function to avoid undesirable results. We showed the results of several experiments where our method showed improvements over an existing method in terms of success rates of finding correct supported models of normal logic programs.

Raw computational performance is not the focus of this paper and we leave thorough benchmarking for future work. Our method does not compute stable models except cases where supported and stable models coincide. For computing stable models, we may consider using our method as a preprocessor to the method by [4], where our method can be used to find supported models as promising candidates, which can then be refined for better performance.

References

[1] T. Sato, Embedding tarskian semantics in vector spaces, in: The Workshops of the the Thirty-First AAAI Conference on Artificial Intelligence, Saturday, February 4-9, 2017, San Francisco, California, USA, volume WS-17 of *AAAI Workshops*, AAAI Press, 2017.

[2] C. Sakama, K. Inoue, T. Sato, Linear Algebraic Characterization of Logic Programs, in: G. Li, Y. Ge, Z. Zhang, Z. Jin, M. Blumenstein (Eds.), Knowledge Science, Engineering and Management, Lecture Notes in Computer Science, Springer International Publishing, Cham, 2017, pp. 520–533. doi:10.1007/978-3-319-63558-3_44.

[3] Y. Aspis, K. Broda, A. Russo, Tensor-based abduction in horn propositional programs, in: ILP 2018 - 28th International Conference on Inductive Logic Programming, CEUR Workshop Proceedings, 2018, pp. 68–75.

[4] T. Q. Nguyen, K. Inoue, C. Sakama, Enhancing linear algebraic computation of logic programs using sparse representation, in: F. Ricca, A. Russo, S. Greco, N. Leone, A. Artikis, G. Friedrich, P. Fodor, A. Kimmig, F. A. Lisi, M. Maratea, A. Mileo, F. Riguzzi (Eds.), Proceedings 36th International Conference on Logic Programming (Technical Communications), ICLP Technical Communications 2020, (Technical Communications) UNICAL, Rende (CS), Italy, 18-24th September 2020, volume 325 of *EPTCS*, 2020, pp. 192–205. doi:10.4204/EPTCS.325.24. arXiv:2009.10247.

[5] T. Sato, C. Sakama, K. Inoue, From 3-valued Semantics to Supported Model Computation for Logic Programs in Vector Spaces, in: 12th International Conference on Agents and Artificial Intelligence, 2020, pp. 758–765.

[6] T. Sato, K. Inoue, C. Sakama, Abducing Relations in Continuous Spaces, in: Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, International Joint Conferences on Artificial Intelligence Organization, Stockholm, Sweden, 2018, pp. 1956–1962. doi:10.24963/ijcai.2018/270.

[7] T. Sato, R. Kojima, Logical Inference as Cost Minimization in Vector Spaces, in: A. El Fallah Seghrouchni, D. Sarne (Eds.), Artificial Intelligence. IJCAI 2019 International Workshops, Lecture Notes in Computer Science, Springer International Publishing, Cham, 2020, pp. 239–255. doi:10.1007/978-3-030-56150-5_12.

[8] T. Sato, A linear algebraic approach to Datalog evaluation, Theory and Practice of Logic Programming 17 (2017) 244–265. doi:doi:10.1017/S1471068417000023.

[9] H. A. Blair, F. Dushin, D. W. Jakel, A. J. Rivera, M. Sezgin, Continuous Models of Computation for Logic Programs: Importing Continuous Mathematics into Logic Programming's Algorithmic Foundations, in: K. R. Apt, V. W. Marek, M. Truszczynski, D. S. Warren (Eds.), The Logic Programming Paradigm: A 25-Year Perspective, Artificial Intelligence, Springer, Berlin, Heidelberg, 1999, pp. 231–255. doi:10.1007/978-3-642-60085-2_10.

[10] Y. Aspis, K. Broda, A. Russo, J. Lobo, Stable and Supported Semantics in Continuous Vector Spaces, in: Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning, 2020, pp. 59–68. doi:10.24963/kr.2020/7.

[11] W. Marek, V. Subrahmanian, The relationship between stable, supported, default and autoepistemic semantics for general logic programs, Theoretical Computer Science 103 (1992) 365–386. doi:10.1016/0304-3975(92)90019-C.

[12] K. R. Apt, H. A. Blair, A. Walker, Towards a theory of declarative knowledge, in: Founda-

tions of Deductive Databases and Logic Programming, Elsevier, 1988, pp. 89–148.
[13] H. D. Nguyen, C. Sakama, T. Sato, K. Inoue, An efficient reasoning method on logic programming using partial evaluation in vector spaces, Journal of Logic and Computation 31 (2021) 1298–1316. doi:10.1093/logcom/exab010.
[14] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, P. Kuksa, Natural Language Processing (Almost) from Scratch, Journal of Machine Learning Research 12 (2011) 2493–2537.
[15] S. Boyd, L. Vandenberghe, Convex Optimization, Cambridge University Press, USA, 2004.