Towards end-to-end ASP computation

Taisuke Sato [a], Akihiro Takemura [b] and Katsumi Inoue [c]
[a] *National Institute of Informatics,Japan*
E-mail: taisukest@gmail.com
[b] *National Institute of Informatics, Japan*
E-mail: atakemura@nii.ac.jp
[c] *National Institute of Informatics, Japan*
E-mail: inoue@nii.ac.jp

Abstract. We propose an end-to-end approach for answer set programming (ASP) and linear algebraically compute stable models satisfying given constraints. The idea is to implement Lin-Zhao's theorem [1] together with constraints directly in vector spaces as numerical minimization of a cost function constructed from a matricized normal logic program, loop formulas in Lin-Zhao's theorem and constraints, thereby no use of symbolic ASP or SAT solvers involved in our approach. We also propose precomputation that shrinks the program size and heuristics for loop formulas to reduce computational difficulty. We empirically test our approach with programming examples including the 3-coloring and Hamiltonian cycle problems. As our approach is purely numerical and only contains vector/matrix operations, acceleration by parallel technologies such as many-cores and GPUs is expected.

Keywords: Answer Set Programming, end-to-end ASP, vector space, cost minimization, loop formula, supported model, stable model

1. Introduction

Computing stable model semantics [2] lies at the heart of answer set programming (ASP) [3–5] and there have been a variety of approaches proposed so far. Early approaches such as Smodels [6] used backtracking. Then the concept of loop formula was introduced and approaches that use a SAT solver to compute stable models based on Lin-Zhao's theorem [1] were proposed. They include ASSAT [1] and Cmodels [7] for example. Later more elaborated approaches such as Clasp [8, 9] based on conflict-driven no good learning have been developed. While these symbolic approaches continue to predominate in ASP, there has been another trend towards differentiable methods. For example Differentiable ASP/SAT [10] computes stable models by an ASP solver that utilizes derivatives of a cost function. More recently NeurASP [11] and SLASH [12] combined deep learning and ASP. In their approaches, deep learning is not used in an end-to-end way to compute stable models but used as a component to compute and learn probabilities represented by special atoms interfacing to ASP. A step towards end-to-end computation was taken by Aspis [13] and Takemura [14]. They formulated the computation of supported models, a super class of stable models[1], entirely as fixedpoint computation in vector spaces and obtain supported models represented by bi-

[1] Supported models and stable models of a propositional normal logic program coincide when the program is tight (no infinite call chain through positive goals) [15, 16]. Also in the context of probabilistic modeling, an end-to-end sampling of supported models was proposed by Sato and Kojima in [17, 18].

nary vectors [19, 20]. However there still remains a gap between computing stable models and computing supported ones[2].

In this paper, we propose an end-to-end approach for ASP and compute stable models satisfying given constraints in vector spaces. The idea is simple; we implement Lin-Zhao's theorem [1] together with constraints directly in vector spaces as a cost minimization problem, thereby no use of symbolic ASP or SAT solvers involved. Also as our approach is purely numerical and only contains vector/matrix operations, acceleration by parallel technologies such as many-cores and GPUs is expected.

Technically, Lin-Zhao's theorem [1] states that a stable model of a ground normal logic program coincides with a supported model which satisfies "loop formulas" associated with the program. They are propositional formulas indicating how to get out of infinite loops of top-down rule invocation. We formulate finding such a model as root finding in a vector space of a non-negative cost function represented in terms of the matricized program and loop formulas. The problem is that in whatever approach we may take, symbolic or non-symbolic, computing supported models is NP-hard (for example graph coloring is solved by computing supported models) and there can be exponentially many loop formulas to be satisfied. We reduce this computational difficulty in two ways. One is precomputation that removes atoms from the search space which are known to be false in any stable model and yields a smaller program. The other is to heuristically choose loop formulas to be satisfied. In a symbolic approach, the latter would mean allowing non-stable model computation but in our continuous approach, it means gracefully degraded gradient information for the continuous search process.

Our end-to-end computing framework differs from those by [13] and [14] in that they basically compute supported models and the computing process itself has no mechanism such as loop formulas to exclude non-stable models. Also we impose no restriction on the syntax of programs like the MD condition [13] or the SD condition [14], any propositional normal logic program is acceptable. More importantly we incorporate the use of constraints, i.e. rules like $\leftarrow a \wedge \neg b$, which make ASP programming smooth and practical.

Hence our contributions include

- a proposal of end-to-end computing of stable models in vector spaces for unrestricted programs
- augmentation of the above by constraints
- introduction of precomputation and heuristics to reduce computational difficulty of stable model computation.

We add that since our primary purpose in this paper is to establish theoretical feasibility of end-to-end ASP computing in vector spaces, programming examples are small and implementation is of preliminary nature.

In what follows, after preliminaries in Section 2, we formulate the computation of supported models in vector spaces in Section 3 and that of stable models in Section 4. We then show programming examples in Section 5 including ASP programs for the 3-coloring problem and the HC problem. We there compare performance of precomputation and loop formula heuristics. Section 6 contains related work and Section 7 is conclusion.

2. Preliminaries

In this paper, we mean by a program a propositional normal logic program P which is a finite set of *rule*s of the form $a \leftarrow G$ where a is an atom[3] called the head, G a conjunction of literals[4] called the body of the rule respectively. We suppose P is written in a given set of atoms $\mathcal{A}$ but usually assume $\mathcal{A}$ = atom(P), i.e. the set of atoms occurring in P. We use G^+ to denote the conjunction of positive literals in G. G may be empty. The empty conjunction is always true. We call $a \leftarrow G$ *rule* for a. Let $a \leftarrow G_1, \ldots, a \leftarrow G_m$ be rules for a in P. When $m > 0$, put $\mathrm{iff}(a) = a \Leftrightarrow G_1 \vee \cdots \vee G_m$. When $m = 0$, i.e. there is no rule for a, put $\mathrm{iff}(a) = a \Leftrightarrow \bot$ where $\bot$ is a special symbol representing the empty disjunction which is always false. We call $\mathrm{iff}(a)$ the *completed rule* for a. The *completion*

[2] Independently of differentiable approaches mentioned here, Tuan et al. adopted matrix encoding for propositional normal logic programs based on [21] and proposed to compute stable models in vector spaces by a generate-and-test approach using sparse representation [22].

[3] We equate propositional variables with atoms.

[4] A literal is an atom (positive literal) or its negation (negative literal).

of P, comp(P), is defined as comp(P) = $\{\text{iff}(a) \mid \text{atom } a \text{ occurs in } P\}$. For a finite set S, we denote the number of elements in S by $|S|$. So $|P|$ is the number of rules in the program P.

A *model* (assignment) I over a set of atoms $\mathcal{A}$ is a mapping which determines the truth value of each atom $a \in \mathcal{A}$. Then the truth value of a formula F is inductively defined by I and if F becomes true evaluated by I, we say I satisfies F, F is true in I or I is a model of F and write $I \models F$. This notation is extended to a set $F = \{F_1, \ldots, F_k\}$ by considering F as a conjunction $F_1 \wedge \cdots \wedge F_k$. For convenience, we always equate I with $\{a \in \mathcal{A} \mid I \models a\}$, i.e. the set of atoms true in I. When I satisfies all rules in the program P, i.e. $I \models P$, I is said to be a model of P. If no rule body contains negative literals, P is said to be a definite program. In that case, P always has the least model (in the sense of set inclusion) $\{a \in \mathcal{A} \mid P \vdash a\}$, i.e. the set of atoms provable from P. A model I of comp(P) is called a *supported model* of P [20]. When P is a definite program, its least model is also a supported model. In general, for non-definite P, there can be zero or multiple supported models. *Stable model*s are a subclass of supported models. They are defined as follows. Given a program P and a model I, remove all rules from P whose body contains a negative literal false in I, then remove all negative literals from the remaining rules. The resulting program, P^I, is called the Gelfond-Lifschitz (GL) reduction of P by I or just the reduct of P by I. It is a definite program and has the least model. If this least model is identical to I, I is called a *stable model* of P [2]. P may have zero or multiple stable models like supported models. Since the existence of a stable model is NP-complete [4] and so is a supported model, their computation is expected to be hard.

Let $F = d_1 \vee \cdots \vee d_h$ be a Boolean formula in n variables (atoms) in disjunctive normal form (DNF) where each d_i ($1 \leqslant i \leqslant h$) is a conjunction of literals and called a disjunct of F. When F has no disjunct, F is false. F is called *full* when every d_i is a conjunction of n distinct literals.

A walk in a directed graph is a sequence $v_1 \to v_2 \to \cdots \to v_k$ ($k \geqslant 1$) of vertices representing the corresponding non-zero sequence of edges $(v_1, v_2), \ldots, (v_{k-1}, v_k)$. When $v_k = v_1$, it is said to be closed. A cycle is a closed walk $v_1 \to v_2 \to \cdots \to v_k \to v_1$ where $\{v_1, \ldots, v_k\}$ are all distinct. A Hamiltonian cycle is a cycle which visits every vertex once. A path is a walk with no vertex repeated. A directed subgraph is called *strongly connected* if there are paths from v_1 to v_2 and from v_2 to v_1 for any pair of distinct vertices v_1 and v_2. This "strongly connected" relation induces an equivalence relation over the set of vertices and an induced equivalence class is called a strongly connected component (SCC).

The *positive dependency graph* pdg(P) for a program P is a directed graph where vertices are atoms occurring in P and there is an edge (a, b) from atom a to atom b if-and-only-if (iff) there is a rule $a \leftarrow G$ in P such that b is a positive literal in G. P is said to be *tight* [15][5] when pdg(P) is acyclic, i.e. has no cycle. A *loop* $L = \{a_1, \ldots, a_k\}$ ($k > 0$) in P is a set of atoms where for any pair of atoms a_1 and a_2 in L ($a_1 = a_2$ allowed), there is a path in pdg(P) from a_1 to a_2 and also from a_2 to a_1.

We denote vectors by bold italic lower case letters such as $\mathbf{a}$ where $\mathbf{a}(i)$ represents the i-th element of $\mathbf{a}$. Vectors are column vectors by default. We use $(\mathbf{a} \bullet \mathbf{b})$ to stand for the inner product (dot product) of vectors $\mathbf{a}$ and $\mathbf{b}$ of the same dimension. $\|\mathbf{a}\|_1$ and $\|\mathbf{a}\|_2$ respectively denote the 1-norm and 2-norm of $\mathbf{a}$ where $\|\mathbf{a}\|_1 = \sum |\mathbf{a}(i)|$ and $\|\mathbf{a}\|_2 = \sqrt{\sum \mathbf{a}(i)^2}$. We use $\mathbf{1}$ to denote an all-ones vector of appropriate dimension. A model I over a set $\mathcal{A} = \{a_1, \ldots, a_n\}$ of n ordered atoms is equated with an n-dimensional binary vector $\mathbf{u}_I \in \{1, 0\}^n$ such that $\mathbf{u}_I(i) = 1$ if a_i is true in I and $\mathbf{u}_I(i) = 0$ otherwise ($1 \leqslant i \leqslant n$). $\mathbf{u}_I$ is called the vectorized I.

Italic capital letters such as A stand for a matrix. We use $A(i, j)$ to denote the i, j-th element of A, $A(i, :)$ the i-th row of A and $A(:, j)$ the j-th column of A, respectively. We often consider one dimensional matrices as (row or column) vectors. $\|A\|_F$ denotes the Frobenius norm of A. Let $A, B \in \mathbb{R}^{m \times n}$ be $m \times n$ matrices. $A \odot B$ denotes their Hadamard product, i.e., $(A \odot B)(i, j) = A(i, j)B(i, j)$ for $i, j (1 \leqslant i \leqslant m, 1 \leqslant j \leqslant n)$. $[A; B]$ designates the $2m \times n$ matrix of A stacked onto B. We implicitly assume that all dimensions of vectors and matrices in various expressions are compatible. We introduce a piece-wise linear function $\min_1(x) = \min(x, 1)$ that returns the lesser of 1 and x as an activation function which is related to the popular activation function $\text{ReLU}(x) = \max(x, 0)$ by $1 - \min_1(x) = \text{ReLU}(1 - x)$. $\min_1(A)$ denotes the result of component-wise application of $\min_1(x)$ to matrix A. We also introduce thresholding notation. Suppose θ is a real number and $\mathbf{a}$ an n-dimensional vector. Then $[\mathbf{a} \leqslant \theta]$ denotes a binary vector obtained by thresholding $\mathbf{a}$ at θ where for $i (1 \leqslant i \leqslant n)$, $[\mathbf{a} \leqslant \theta](i) = 1$ if $\mathbf{a}(i) \leqslant \theta$ and $[\mathbf{a} \leqslant \theta](i) = 0$

[5] In [15], it is called "positive-order-consistent".

otherwise. $[\mathbf{a} \geqslant \theta]$ is treated similarly. We extend thresholding to matrices. Thus $[A \leqslant 1]$ means a matrix such that $[A \leqslant 1](i,j) = 1$ if $A(i,j) \leqslant 1$ and $[A \leqslant 1](i,j) = 0$ otherwise. For convenience, we generalize bit inversion to an n-dimensional vector $\mathbf{a}$ and use an expression $1-\mathbf{a}$ to denote the n-dimensional vector such that $(1-\mathbf{a})(i) = 1-\mathbf{a}(i)$ for $i\,(1 \leqslant i \leqslant n)$. $1-A$ is treated similarly.

3. Computing supported models in vector spaces

In this section, we formulate the semantics of supported models in vector spaces and show how to compute it by cost minimization.

3.1. Matricized programs

First we encode programs as matrices. For concreteness, we explain by an example (generalization is easy). Suppose we are given a program P_0 below containing three rules $\{r_1, r_2, r_3\}$ in a set of atoms $\mathcal{A}_0 = \{p, q, r\}$.

$$P_0 = \begin{cases} p \leftarrow q \wedge \neg r & : \text{rule } r_1 \text{ for } p \\ p \leftarrow \neg q & : \text{rule } r_2 \text{ for } p \\ q & : \text{rule } r_3 \text{ for } q \end{cases} \tag{1}$$

Assuming atoms are ordered as p, q, r and correspondingly so are the rules $\{r_1, r_2, r_3\}$ as in (1), we encode P_0 as a pair of matrices (D_0, Q_0). Here Q_0 represents conjunctions (the bodies of $\{r_1, r_2, r_3\}$) and D_0 their disjunctions so that they jointly represent P_0.

$$Q_0 = \begin{array}{c} \begin{array}{cccccc} p & q & r & \neg p & \neg q & \neg r \end{array} \\ \left[\begin{array}{cccccc} 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \end{array} \begin{array}{l} : r_1 \text{ has the body } q \wedge \neg r \\ : r_2 \text{ has the body } \neg q \\ : r_3 \text{ has the empty body} \end{array} \tag{2}$$

$$D_0 = \begin{array}{c} \begin{array}{ccc} r_1 & r_2 & r_3 \end{array} \\ \left[\begin{array}{ccc} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{array}\right] \end{array} \begin{array}{l} : p \text{ has two rules } r_1 \text{ and } r_2 \\ : q \text{ has one rule } r_3 \\ : r \text{ has no rule} \end{array} \tag{3}$$

As can be seen, Q_0 represents conjunctions in P_0 in such a way that $Q_0(1,:)$ for example represents the conjunction $q \wedge \neg r$ of the first rule in P_0 by setting $Q_0(1,2) = Q_0(1,6) = 1$ and so on. D_0 represents disjunctions of rule bodies. So $D_0(1,1) = D_0(1,2) = 1$ means the first atom p in $\{p, q, r\}$ has two rules, the first rule r_1 and the second rule r_2, representing a disjunction $(q \wedge \neg r) \vee \neg q$ for p.

In general, a program P that has m rules in n atoms is numerically encoded as a pair $P = (D, Q)$ of binary matrices $D \in \{0,1\}^{n \times m}$ and $Q \in \{0,1\}^{m \times 2n}$, which we call a matricized P. Q represents rule bodies in P. Suppose atoms are ordered like $\mathcal{A} = \{a_1, \ldots, a_n\}$ and similarly rules are ordered like $\{r_1 : a_{i_1} \leftarrow G_1, \ldots, r_m : a_{i_m} \leftarrow G_m\}$. Then the j-th row $Q(j,:)\,(1 \leqslant j \leqslant m)$ encodes the j-th conjunction G_j of the j-th rule $a_{i_j} \leftarrow G_j$. Write $G_j = a_{i_1} \wedge \cdots \wedge a_{i_p} \wedge \neg a_{i_{p+1}} \wedge \cdots \wedge \neg a_{i_{p+q}}$ $(1 \leqslant p, q \leqslant n)$. Then an element of $Q(j,:)$ is zero except for $Q(j,i_1) = \cdots = Q(j,i_p) = Q(j,n+i_{p+1}) = \cdots = Q(j,n+i_{p+q}) = 1$. D combines these conjunctions as a disjucntion (DNF) for each atom in $\mathcal{A}$. If the i-th atom $a_i \in \mathcal{A}\,(1 \leqslant i \leqslant n)$ has rules $\{a_i \leftarrow G_{j_1}, \ldots, a_i \leftarrow G_{j_s}\}$ in P, we put $D(i,j_1) = \cdots = D(i,j_s) = 1$ to represent a disjunction $G_{j_1} \vee \cdots \vee G_{j_s}$ which is the right hand side of the completed rule for a_i: $\text{iff}(a_i) = a_i \Leftrightarrow G_{j_1} \vee \cdots \vee G_{j_s}$. If a_i has no rule, we put $D(i,j) = 0$ for all $j\,(1 \leqslant j \leqslant m)$. Thus the matricized $P = (D, Q)$ can represent the completed program $\text{comp}(P)$.

3.2. Evaluation of formulas and the reduct of a program in vector spaces

Here we explain how the propositional formulas and the reduct of a program are evaluated by a model in vector spaces. Let I be a model over a set $\mathcal{A}$ of atoms. Recall that I is equated with a subset of $\mathcal{A}$. We inductively define the relation "a formula F is true in I"[6], $I \models F$ in notation, as follows. For an atom a, $I \models a$ iff $a \in I$. For a compound formula F, $I \models \neg F$ iff $I \not\models F$. When F is a disjunction $F_1 \vee \cdots \vee F_k$ ($k \geqslant 0$), $I \models F$ iff there is some i ($1 \leqslant i \leqslant k$) s.t. $I \models F_i$. So the empty disjunction ($k = 0$) is always false. We consider a conjunction $F_1 \wedge \cdots \wedge F_k$ as a syntax sugar for $\neg(\neg F_1 \vee \cdots \vee \neg F_k)$ using De Morgan's law. Consequently the empty conjunction is always true. Let P be a program having m ordered rules in n ordered atoms as before and $G = a_{i_1} \wedge \cdots \wedge a_{i_p} \wedge \neg a_{i_{p+1}} \wedge \cdots \wedge \neg a_{i_{p+q}}$ the body of a rule $a \leftarrow G$ in P. By definition, $I \models G$ (G is true in I) iff $\{a_{i_1}, \ldots, a_{i_p}\} \subseteq I$ and $\{a_{i_{p+1}}, \ldots, a_{i_{p+q}}\} \cap I = \emptyset$. Also let $\mathrm{iff}(a_i) = a_i \Leftrightarrow G_{j_1} \vee \cdots \vee G_{j_s}$ be the completed rule for an atom a_i in P. We see $I \models \mathrm{iff}(a_i)$ iff $(a_i \in I \text{ iff } I \models G_{j_1} \vee \cdots \vee G_{j_s})$.

Now we isomorphically embed the above symbolic evaluation to the one in vector spaces. Let I be a model over ordered atoms $\mathcal{A} = \{a_1, \ldots, a_n\}$. We first vectorize I as a binary column vector $\mathbf{u}_I$ such that $\mathbf{u}_I(i) = 1$ if $a_i \in I$ and $\mathbf{u}_I(i) = 0$ ($1 \leqslant i \leqslant n$) otherwise, and introduce the dualized $\mathbf{u}_I$ written as $\mathbf{u}_I^\delta$ by $\mathbf{u}_I^\delta = [\mathbf{u}_I; (1 - \mathbf{u}_I)]$. $\mathbf{u}_I^\delta$ is a vertical concatenation of $\mathbf{u}_I$ and the bit inversion of $\mathbf{u}_I$.

Consider a matricized program $P = (D, Q)$ ($D \in \{0,1\}^{n \times m}$, $Q \in \{0,1\}^{m \times 2n}$) and its j-th rule r_j having a body G_j represented by $Q(j,:)$. Compute $Q(j,:)\mathbf{u}_I^\delta$ which is the number of true literals in I in G_j and compare it with the number of literals $|Q(j,:)|_1$[7] in G_j. When $|Q(j,:)|_1 = Q(j,:)\mathbf{u}_I^\delta$ holds, all literals in G_j are true in I and hence the body G_j is true in I. In this way, we can algebraically compute the truth value of each rule body, but since we consider a conjunction as a negated disjunction, we instead compute $Q(j,:)(1 - \mathbf{u}_I^\delta)$ which is the number of false literals in G_j. If this number is non-zero, G_j have at least one literal false in I, and hence G_j is false in I. The converse is also true. The existence of a false literal in G_j is thus computed by $\min_1(Q(j,:)(1 - \mathbf{u}_I^\delta))$ which is 1 if there is a false literal, and 0 otherwise. Consequently $1 - \min_1(Q(j,:)(1 - \mathbf{u}_I^\delta)) = 1$ if there is no false literal in G_j and vice versa. In other words, $1 - \min_1(Q(j,:)(1 - \mathbf{u}_I^\delta))$ computes $I \models G_j$.

Now let $\{a_i \leftarrow G_{j_1}, \ldots, a_i \leftarrow G_{j_s}\}$ be an enumeration of rules for $a_i \in \mathcal{A}$ and $G_{j_1} \vee \cdots \vee G_{j_s}$ the disjunction of the rule bodies. $d_i = \sum_{t=1}^{s}(1 - \min_1(Q(j_t,:)(1 - \mathbf{u}_I^\delta)))$ is the number of rule bodies in $\{G_{j_1}, \ldots, G_{j_s}\}$ that are true in I. Noting $D(i,j) = 1$ if $j \in \{j_1, \ldots, j_s\}$ and $D(i,j) = 0$ otherwise by construction of D in $P = (D, Q)$, we replace the summation $\sum_{t=1}^{s}$ by matrix multiplication and obtain $d_i = D(i,:)(1 - \min_1(Q(1 - \mathbf{u}_I^\delta)))$. Introduce a column vector $\mathbf{d}_I = D(1 - \min_1(Q(1 - \mathbf{u}_I^\delta)))$. We have $\mathbf{d}_I(i) = d_i$ = the number of rules for a_i whose bodies are true in I ($1 \leqslant i \leqslant n$).

Proposition 1. *Let $P = (D, Q)$ be a matricized program P in a set of atoms $\mathcal{A}$ and $\mathbf{u}_I$ a vectorized model I over $\mathcal{A}$. Put $\mathbf{d}_I = D(1 - min_1(Q(1 - \mathbf{u}_I^\delta)))$. It holds that*

$$I \models comp(P) \quad \textit{if-and-only-if} \quad \|\mathbf{u}_I - \min_1(\mathbf{d}_I)\|_2 = 0. \tag{4}$$

(Proof) Put $n = |\mathcal{A}|$. Suppose $I \models \mathrm{comp}(P)$ and write $\mathrm{iff}(a_i)$, the completed rule for an atom $a_i \in \mathcal{A}$ ($1 \leqslant i \leqslant n$), as $\mathrm{iff}(a_i) = a_i \Leftrightarrow G_{j_1} \vee \cdots \vee G_{j_s}$ ($s \geqslant 0$). We have $I \models \mathrm{iff}(a_i)$. So if $\mathbf{u}_I(i) = 1$, $I \models a_i$, and hence $I \models G_{j_1} \vee \cdots \vee G_{j_s}$, giving $d_i \geqslant 1$ because d_i is the number of rule bodies in $\{G_{j_1}, \ldots, G_{j_s}\}$ that are true in I. So $\min_1(d_i) = 1$ holds. Otherwise if $\mathbf{u}_I(i) = 0$, we have $I \not\models a_i$ and $I \not\models G_{j_1} \vee \cdots \vee G_{j_s}$. Consequently none of the rule bodies are true in I and we have $d_i = \min_1(d_i) = 0$. Putting the two together, we have $\mathbf{u}_I(i) = d_i$. Since i is arbitrary, we conclude $\mathbf{u}_I = \min_1(\mathbf{d}_I)$, or equivalently $\|\mathbf{u}_I - \min_1(\mathbf{d}_I)\|_2 = 0$. The converse is similarly proved. Q.E.D.

Proposition 1 says that whether I is a supported model of the program P or not is determined by computing $\mathbf{u}_I - \min_1(\mathbf{d}_I)$ in vector spaces whose complexity is $O(mn)$ where m is the number of rules in P, n that of atoms occurring in P.

In the case of $P_0 = (Q_0, D_0)$ in (1) having three rules $\{r_1, r_2, r_3\}$, take a model $I_0 = \{p, q\}$ over the ordered atom set $\mathcal{A}_0 = \{p, q, r\}$ where p and q are true in I_0 but r is false in I_0. Then we have $\mathbf{u}_{I_0} = [1\ 1\ 0]^T$, $\mathbf{u}_{I_0}^\delta = [1\ 1\ 0\ 0\ 0\ 1]^T$, $1 - \mathbf{u}_{I_0}^\delta = [0\ 0\ 1\ 1\ 1\ 0]^T$ and finally $Q_0(1 - \mathbf{u}_{I_0}^\delta) = [0\ 1\ 0]^T$. The last equation says that the rule bodies of r_1, r_2 and

[6] Equivalently, I satisfies F.

[7] $|\mathbf{v}|_1 = \sum_i |\mathbf{v}(i)|$ is the 1-norm of a vector $\mathbf{v}$.

r_3 have respectively zero, one and zero literal false in I_0. Hence $\min_1(Q_0(1-\mathbf{u}_{I_0}^{\delta})) = [0\ 1\ 0]^T$ indicates that only the second rule body is false and the other two bodies are true in I_0. So its bit inversion $1-\min_1(Q_0(1-\mathbf{u}_{I_0}^{\delta})) = [1\ 0\ 1]^T$ indicates that the second rule body is false in I_0 while others are true in I_0. Thus by combining these truth values in terms of disjunctions D_0, we obtain $\mathbf{d}_{I_0} = D_0(1-\min_1(Q_0(1-\mathbf{u}_{I_0}^{\delta}))) = [1\ 1\ 0]^T$.

$\mathbf{d}_{I_0} = [1\ 1\ 0]^T$ denotes for each atom $a \in \mathcal{A}_0$ the number of rules for a whose body is true in I_0. For example $\mathbf{d}_{I_0}(1) = 1$ means that the first atom p in $\mathcal{A}_0$ has one rule $(p \leftarrow q \wedge \neg r)$ whose body $(q \wedge \neg r)$ is true in I_0. Likewise $\mathbf{d}_{I_0}(2) = 1$ means that the second atom q has one rule $(q \leftarrow)$ whose body (empty) is true in I_0. $\mathbf{d}_{I_0}(3) = 0$ indicates that the third atom r has no such rule. Therefore $\min_1(\mathbf{d}_{I_0}) = [1\ 1\ 0]^T$ denotes the truth values of the right hand sides of the completed rules $\{\mathrm{iff}(p), \mathrm{iff}(q), \mathrm{iff}(r)\}$ evaluated by I_0. Since $\mathbf{u}_{I_0} = \min_1(\mathbf{d}_{I_0})$ holds, it follows from **Proposition** 1 that I_0 is a supported model of P_0.

We next show how P^I, the reduct of P by I, is dealt with in vector spaces. We assume P has m rules $\{r_1, \ldots, r_m\}$ in a set $\mathcal{A} = \{a_1, \ldots, a_n\}$ of n ordered atoms as before. We first show the evaluation of the reduct of the matricized program $P = (D, Q)$ by a vectorized model $\mathbf{u}_I$. Write $Q \in \{0,1\}^{m \times 2n}$ as $Q = [Q^{(1)}\ Q^{(2)}]$ where $Q^{(1)} \in \{0,1\}^{m \times n}$ (resp. $Q^{(2)} \in \{0,1\}^{m \times n}$) is the left half (resp. the right half) of Q representing the positive literals (resp. negative literals) of each rule body in Q. Compute $M^{(2)} = 1 - \min_1(Q^{(2)}\mathbf{u}_I)$. It is an $m \times 1$ matrix (treated as a column vector here) such that $M^{(2)}(j) = 0$ if the body of r_j contains a negative literal false in I and $M^{(2)}(j) = 1$ otherwise $(1 \leqslant j \leqslant m)$. Let r_j^+ be a rule r_j with negative literals in the body deleted. We see that $P^I = \{r_j^+ \mid M^{(2)}(j) = 1, 1 \leqslant j \leqslant m\}$ and P^I is syntactically represented by $(D^I, Q^{(1)})$ where $D^I = D$ with columns $D(:, j)$ replaced by the zero column vector if $M^{(2)}(j) = 0$ $(1 \leqslant j \leqslant m)$. $D^I(i,:)$ denotes a rule set $\{r_j^+ \mid D^I(i,j) = 1, 1 \leqslant j \leqslant m\}$ in P^I for $a_i \in \mathcal{A}$. We call $P^I = (D^I, Q^{(1)})$ the matricized reduct of P by I.

The matricized reduct $P^I = (D^I, Q^{(1)})$ is evaluated in vector spaces as follows. Compute $M^{(1)} = M^{(2)} \odot (1 - \min_1(Q^{(1)}(1-\mathbf{u}_I)))$[8]. $M^{(1)}$ denotes the truth values of rule bodies in P^I evaluated by I. Thus $M^{(1)}(j) = 1$ $(1 \leqslant j \leqslant m)$ if r_j^+ is contained in P^I and its body is true in I. Otherwise $M^{(1)}(j) = 0$ and r_j^+ is not contained in P^I or the body of r_j^+ is false in I. Introduce $\mathbf{d}_I^+ = DM^{(1)}$. $\mathbf{d}_I^+(i)$ $(1 \leqslant i \leqslant n)$ is the number of rules in P^I for the i-th atom a_i in $\mathcal{A}$ whose bodies are true in I.

Proposition 2. *Let $P = (D, Q)$ be a matricized program P in a set $\mathcal{A} = \{a_1, \ldots, a_n\}$ of n ordered atoms and I a model over $\mathcal{A}$. Write $Q = [Q^{(1)}\ Q^{(2)}]$ as above. Let $\mathbf{u}_I$ be the vectorized model I. Compute $M^{(2)} = 1 - \min_1(Q^{(2)}\mathbf{u}_I)$, $M^{(1)} = M^{(2)} \odot (1 - \min_1(Q^{(1)}(1-\mathbf{u}_I)))$ and $\mathbf{d}_I^+ = DM^{(1)}$. Also compute $\mathbf{d}_I = D(1 - \min_1(Q(1-\mathbf{u}_I^{\delta})))$. Then, $I \models \mathrm{comp}(P)$, $\|\mathbf{u}_I - \min_1(\mathbf{d}_I)\|_2 = 0$, $\|\mathbf{u}_I - \min_1(\mathbf{d}_I^+)\|_2 = 0$ and $I \models \mathrm{comp}(P^I)$ are all equivalent.*

(Proof) We prove $\mathbf{d}_I = \mathbf{d}_I^+$ first. Recall that a rule r_j^+ in P^I is created by removing negative literals true in I from the body of r_j in P. So for any $a_i \in \mathcal{A}$, it is immediate that a_i has a rule $r_j \in P$ whose body is true in I iff a_i has the rule $r_j^+ \in P^I$ whose body is true in I. Thus $\mathbf{d}_I(i) = \mathbf{d}_I^+(i)$ for every i $(1 \leqslant i \leqslant n)$, and hence $\mathbf{d}_I = \mathbf{d}_I^+$. Consequently, we have $\|\mathbf{u}_I - \min_1(\mathbf{d}_I)\|_2 = 0$ iff $\|\mathbf{u}_I - \min_1(\mathbf{d}_I^+)\|_2 = 0$. Also $I \models \mathrm{comp}(P^I)$ iff $\|\mathbf{u}_I - \min_1(\mathbf{d}_I^+)\|_2 = 0$ is proved similarly to **Proposition** 1 (details omitted). The rest follows from **Proposition** 1. Q.E.D.

From the viewpoint of end-to-end ASP, **Proposition** 2 means that we can obtain a supported model I as a binary solution $\mathbf{u}_I$ of the equation $\mathbf{u}_I - \min_1(\mathbf{d}_I) = 0$ derived from P or $\mathbf{u}_I - \min_1(\mathbf{d}_I^+) = 0$ derived from the reduct P^I. Either equation is possible and gives the same result but their computation will be different. This is because the former equation $\mathbf{u}_I - \min_1(\mathbf{d}_I)$ is piecewise linear w.r.t. $\mathbf{u}_I$ whereas the latter $\mathbf{u}_I - \min_1(\mathbf{d}_I^+)$ is piecewise quadratic w.r.t. $\mathbf{u}_I$.

Now look at $P_0 = \{r_1, r_2, r_3\}$ in (1) and a model $I_0 = \{p, q\}$ again. $P_0^{I_0} = \left\{ \begin{array}{l} p \leftarrow q \\ q \leftarrow \end{array} \right.$ is the reduct of P_0 by I_0. $P_0^{I_0}$ has the least model $\{p, q\}$ that coincides with I_0. So I_0 is a stable model of P_0. To simulate the reduction process of P_0 in vector spaces, let $P_0 = (D_0, Q_0)$ be the matricized P_0. We first decompose Q_0 in (2) as $Q_0 = [Q_0^{(1)}\ Q_0^{(2)}]$ where $Q_0^{(1)}$ is the positive part and $Q_0^{(2)}$ the negative part of Q_0. They are

[8] $\odot$ is component-wise product.

$$Q_0^{(1)} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \text{and} \quad Q_0^{(2)} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

Let $\mathbf{u}_{I_0} = [1\ 1\ 0]^T$ be the vectorized I_0. We first compute $M_0^{(2)} = 1 - \min_1(Q_0^{(2)}\mathbf{u}_{I_0})$ to determine rules to be removed. Since $M_0^{(2)} = [1\ 0\ 1]^T$, the second rule r_2, indicated by $M_0^{(2)}(2) = 0$, is removed from P_0, giving $P_0^{I_0} = \{r_1^+, r_3^+\}$. Using $M_0^{(2)}$ and D_0 shown in (3), we then compute $M_0^{(1)} = M_0^{(2)} \odot (1 - \min_1(Q_0^{(1)}(1 - \mathbf{u}_{I_0}))) = [1\ 0\ 1]^T$ and $\mathbf{d}_{I_0}^+ = D_0 M_0^{(1)} = [1\ 1\ 0]^T$. $\mathbf{d}_{I_0}^+$ denotes the number rule bodies in $P_0^{I_0}$ true in I_0 for each atom. Thus, since $\mathbf{u}_{I_0} = \min_1(\mathbf{d}_{I_0}^+)(= [1\ 1\ 0]^T)$ holds, I_0 is a supported model of P_0 by **Proposition** 2.

3.3. Cost minimization for supported models

Having linear algebraically reformulated several concepts in logic programming, we tackle the problem computing supported models in vector spaces. Although there already exist approaches for this problem, we tackle it without assuming any condition on programs while allowing constraints. Aspis et al. formulated the problem as solving a non-linear equation containing a sigmoid function [13]. They encode normal logic programs differently from ours based on Sakama's encoding [21] and impose the MD condition on programs which is rather restrictive. No support is provided for constraints in their approach. Later Takemura et al. proposed another approach [14] which encodes a program in terms of a single matrix and evaluates conjunctions by the number of true literals. They compute supported models by minimizing a non-negative function, not solving an equation like [13]. Their programs are however restricted to those satisfying the SD condition and constraints are not considered.

Here we introduce an end-to-end way of computing supported models in vector spaces through cost minimization of a new cost function based on the evaluation of disjunction. We impose no syntactic restriction on programs and allow constraints. We believe that these two features make our end-to-end ASP approach more feasible.

We can base our supported model computation either on **Proposition** 1 or on **Proposition** 2. In the latter case, we have to compute GL reduction which requires complicated computation compared to the former case. So for the sake of simplicity, we explain the former. Then our task in vector spaces is to find a binary vector $\mathbf{u}_I$ representing a supported model I of a matricized program $P = (D, Q)$ that satisfies $\|\mathbf{u}_I - \min_1(\mathbf{d}_I)\|_2 = 0$ where $\mathbf{d}_I = D(1 - \min_1(Q(1 - \mathbf{u}_I^\delta)))$. For this task, we relax $\mathbf{u}_I \in \{1,0\}^n$ to $\mathbf{u} \in \mathbb{R}^n$ and introduce a non-negative cost function J_{SU}:

$$J_{SU} = 0.5 \cdot \left(\|\mathbf{u} - \min_1(\mathbf{d})\|_2^2 + \ell_2 \cdot \|\mathbf{u} \odot (1 - \mathbf{u})\|_2^2\right) \quad \text{where} \quad \ell_2 > 0 \text{ and } \mathbf{d} = D(1 - \min_1(Q(1 - \mathbf{u}^\delta))). \tag{5}$$

Proposition 3. *Let J_{SU} be defined from a program $P = (D, Q)$ as above.*
$J_{SU} = 0$ if-and-only-if $\mathbf{u}$ is a binary vector representing a supported model of P.

(Proof) Apparently if $J_{SU} = 0$, we have $\|\mathbf{u} - \min_1(\mathbf{d})\|_2^2 = 0$ and $\|\mathbf{u} \odot (1 - \mathbf{u})\|_2^2 = 0$. The second equation means $\mathbf{u}$ is binary ($x(1-x) = 0 \Leftrightarrow x \in \{1,0\}$), and the first equation means this binary $\mathbf{u}$ is a vector representing a supported model of P by **Proposition** 1. The converse is obvious. Q.E.D.

J_{SU} is piecewise differentiable and we can obtain a supported model of P as a root $\mathbf{u}$ of J_{SU} by minimizing J_{SU} to zero using Newton's method. The Jacobian $J_{a_{SU}}$ required for Newton's method is derived as follows. We assume P is written in n ordered atoms $\{a_1, \ldots, a_n\}$ and $\mathbf{u} = [u_1, \ldots, u_n]^T$ represents their continuous truth values where $\mathbf{u}(p) = u_p \in \mathbb{R}$ is the continuous truth value for atom a_p ($1 \leqslant p \leqslant n$). For the convenience of derivation, we introduce the dot product $(A \bullet B) = \sum_{i,j} A(i,j)B(i,j)$ of matrices A and B and a one-hot vector $\mathbf{I}_p$ which is a zero vector except for the p-th element and $\mathbf{I}_p(p) = 1$. We note $(A \bullet (B \odot C)) = ((B \odot A) \bullet C)$ and $(A \bullet (BC)) = ((B^T A) \bullet C) = ((AC^T) \bullet B)$ hold.

Let $P = (D, Q)$ be the matricized program and write $Q = [Q^{(1)}\, Q^{(2)}]$. Introduce N, M, $\mathbf{d}$, E, F and compute J_{SU} by

$$
\begin{array}{lcll}
N & = & Q(1-\mathbf{u}^{\delta}) = Q^{(1)}(1-\mathbf{u}) + Q^{(2)}\mathbf{u} & \text{: (continuous) counts of false literals in the rule bodies} \\
M & = & 1-\min_1(N) & \text{: (continuous) truth values of the rule bodies} \\
\mathbf{d} & = & DM & \text{: (continuous) counts of true disjuncts for each atom} \\
E & = & \min_1(\mathbf{d}) - \mathbf{u} & \text{: error between the estimated truth values of atoms and } \mathbf{u} \\
F & = & \mathbf{u} \odot (1-\mathbf{u}) & \\
J_{sq} & = & (E \bullet E) & \\
J_{nrm} & = & (F \bullet F) & \\
J_{SU} & = & J_{sq} + \ell_2 \cdot J_{nrm}. &
\end{array}
\tag{6}
$$

We then compute the Jacobian $J_{a_{SU}}$ of J_{SU}. We first compute $\dfrac{\partial J_{sq}}{\partial u_p}$ where $u_p = \mathbf{u}(p)$ $(1 \leqslant p \leqslant n)$.

$$
\begin{aligned}
\frac{\partial M}{\partial u_p} &= -[N \leqslant 1] \odot \big((Q^{(2)} - Q^{(1)})\mathbf{I}_p\big) = [N \leqslant 1] \odot \big((Q^{(1)} - Q^{(2)})\mathbf{I}_p\big) \\
\frac{\partial J_{sq}}{\partial u_p} &= \left(E \bullet [DM \leqslant 1] \odot \left(D\left(\frac{\partial M}{\partial u_p}\right)\right) - \mathbf{I}_p\right) \\
&= \left(E \bullet [DM \leqslant 1] \odot (D([N \leqslant 1] \odot ((Q^{(1)} - Q^{(2)})\mathbf{I}_p))) - \mathbf{I}_p\right) \\
&= \left(D^T([DM \leqslant 1] \odot E) \bullet [N \leqslant 1] \odot (((Q^{(1)} - Q^{(2)})\mathbf{I}_p)\right) - (E \bullet \mathbf{I}_p) \\
&= \left((Q^{(1)} - Q^{(2)})^T([N \leqslant 1] \odot (D^T([DM \leqslant 1] \odot E))) - E \bullet \mathbf{I}_p\right)
\end{aligned}
$$

Since p is arbitrary, we have $\dfrac{\partial J_{sq}}{\partial \mathbf{u}} = (Q^{(1)} - Q^{(2)})^T([N \leqslant 1] \odot (D^T([DM \leqslant 1] \odot E))) - E$.

Next we compute $\dfrac{\partial J_{nrm}}{\partial u_p}$:

$$
\begin{aligned}
\frac{\partial F}{\partial u_p} &= \left(\frac{\partial \mathbf{u}}{\partial u_p}\right) \odot (1-\mathbf{u}) + \mathbf{u} \odot \left(\frac{\partial (1-\mathbf{u})}{\partial u_p}\right) \\
&= (\mathbf{I}_p \odot (1-\mathbf{u})) - (\mathbf{u} \odot \mathbf{I}_p) = (1-2\mathbf{u}) \odot \mathbf{I}_p \\
\frac{\partial J_{nrm}}{\partial u_p} &= \left(F \bullet \left(\frac{\partial F}{\partial u_p}\right)\right) \\
&= (F \bullet (1-2\mathbf{u}) \odot \mathbf{I}_p) = ((1-2\mathbf{u}) \odot F \bullet \mathbf{I}_p)
\end{aligned}
$$

Again since p is arbitrary, we have $\dfrac{\partial J_{nrm}}{\partial \mathbf{u}} = (1-2\mathbf{u}) \odot F$ and reach

$$
\begin{aligned}
J_{a_{SU}} &= \left(\frac{\partial J_{sq}}{\partial \mathbf{u}}\right) + \ell_2 \cdot \left(\frac{\partial J_{nrm}}{\partial \mathbf{u}}\right) \\
&= (Q^{(1)} - Q^{(2)})^T([N \leqslant 1] \odot (D^T([\mathbf{d} \leqslant 1] \odot E))) - E + \ell_2 \cdot ((1-2\mathbf{u}) \odot F) \\
&\quad \text{where } N = Q(1-\mathbf{u}^{\delta}), \mathbf{d} = D(1-\min_1(N)), E = \min_1(\mathbf{d}) - \mathbf{u}, \text{ and } F = \mathbf{u} \odot (1-\mathbf{u}).
\end{aligned}
\tag{7}
$$

3.4. Adding constraints

A rule which has no head like $\leftarrow a \wedge \neg b$ is called a constraint. We oftentimes need supported models which satisfy constraints. Since constraints are just rules without a head, we encode constraints as rule bodies in a program using a binary matrix $Q_c = [Q_c^{(1)} Q_c^{(2)}]$. We call Q_c *constraint matrix*. We introduce N_c, a non-negative function J_c of $\mathbf{u}$ and J_c's Jacobian J_{a_c} as follows.

$$N_c = Q_c(1-\mathbf{u}^\delta) = Q_c^{(1)}(1-\mathbf{u}) + Q_c^{(2)}\mathbf{u}$$

$$J_c = (\mathbf{1} \bullet (1 - \min_1(N_c))) \quad \text{where } \mathbf{1} \text{ is an all-ones vector} \tag{8}$$

$$J_{a_c} = (Q_c^{(1)} - Q_c^{(2)})^T [N_c \leqslant 1] \tag{9}$$

The meaning of N_c and J_c is clear when $\mathbf{u}$ is binary. Note that any binary $\mathbf{u}$ is considered as a model over a set $\mathcal{A} = \{a_1, \ldots, a_n\}$ of n ordered atoms in an obvious way. Suppose k constraints are given to be satisfied. Then Q_c is a $k \times 2n$ binary matrix and N_c is a $k \times 1$ matrix. $N_c(i)$ $(1 \leqslant i \leqslant k)$ is the number of literals falsified by $\mathbf{u}$ in a conjunction G_i of the i-th constraint $\leftarrow G_i$. So $N_c(i) = 0$, or equivalently $1 - \min_1(N_c(i)) = 1$ implies G_i has no false literal i.e. $\mathbf{u} \models G_i$, and vice versa. Hence $J_c = \sum_{i=1}^{k}(1 - \min_1(N_c(i))) = (\mathbf{1} \bullet (1 - \min_1(N_c)))$ equals the number of violated constraints. Consequently when $\mathbf{u}$ is binary. we can say that $J_c = 0$ iff all constraints are satisfied by $\mathbf{u}$.

When $\mathbf{u}$ is not binary but just a real vector $\in \mathbb{R}^n$, N_c and J_c are thought to be a continuous approximation to their binary counterparts. Since J_c is a piecewise differentiable non-negative function of $\mathbf{u}$, the approximation error can be minimized to zero by Newton's method using J_{a_c} in (9) (the derivation of J_{a_c} is straightforward and omitted).

3.5. An algorithm for computing supported models with constraints

Here we present a minimization algorithm for computing supported models of the matricized program $P = (D, Q)$ which satisfy constraints represented by a constraint matrix Q_c. We first combine J_{SU} and J_c into J_{SU+c} using $\ell_3 > 0$.

$$\begin{aligned} J_{SU+c} &= J_{SU} + \ell_3 \cdot J_c \qquad (10)\\ &= 0.5 \cdot \left(\|\mathbf{u} - \min_1(\mathbf{d})\|_2^2 + \ell_2 \cdot \|\mathbf{u} \odot (1-\mathbf{u})\|_2^2 \right) + \ell_3 \cdot (\mathbf{1} \bullet (1 - \min_1(Q_c(1-\mathbf{u}^\delta)))) \quad \ell_2 > 0, \ell_3 > 0 \\ &\quad \text{where } \mathbf{d} = D(1 - \min_1(Q(1-\mathbf{u}^\delta))) \end{aligned}$$

$$J_{a_{SU+c}} = J_{a_{SU}} + \ell_3 \cdot J_{a_c} \tag{11}$$

The next proposition is immediate from **Proposition** 3.

Proposition 4. *$J_{SU+c} = 0$ if-and-only-if $\mathbf{u}$ represents a supported model of P satisfying a constraint matrix Q_c.*

We compute J_{SU} in J_{SU+c} by (6) and J_c by (8), and their Jacobians $J_{a_{SU}}$ and J_{a_c} by (7) and by (9) respectively. We minimize the non-negative J_{SU+c} to zero by Newton's method using **Algorithm 1**. It finds a solution $\mathbf{u}_*$ of $J_{SU+c} = 0$ which represents a supported model of P satisfying constraint matrix Q_c. The updating formula is derived from the first order Taylor expansion of J_{SU+c} and by solving $J_{SU+c} + (J_{a_{SU*c}} \bullet (\mathbf{u}_{\text{new}} - \mathbf{u})) = 0$ w.r.t. $\mathbf{u}_{\text{new}}$. We use it with a learning rate $\alpha > 0$ as follows.

$$\mathbf{u}_{\text{new}} = \mathbf{u} - \alpha \left(\frac{J_{SU+c}}{(J_{a_{SU+c}} \bullet J_{a_{SU+c}})} \right) J_{a_{SU+c}} \tag{12}$$

Algorithm 1 is a double loop algorithm where the inner j-loop updates $\mathbf{u} \in \mathbb{R}^n$ repeatedly to minimize J_{SU+c} while thresholding $\mathbf{u}$ into a binary solution candidate $\mathbf{u}_* \in \{1,0\}^n$ for $J_{SU+c} = 0$. The outer i-loop is for retry when the inner loop fails to find a solution. The initialization at line 3 is carried out by sampling $\mathbf{u}(i) \sim \mathcal{N}(0,1) + 0.5$

Algorithm 1: minimizing J_{SU+c} to zero

1 **Input:** matricized program $P=(D,Q)$, constraint matrix Q_c, $max\_itr \in \mathbb{Z}$, $max\_try \in \mathbb{Z}$
2 **Output:** binary vector $\mathbf{u}_*$ representing a supported model of P satisfying constraints represented by Q_c
3 $\mathbf{u} \leftarrow$ random initialization
4 **for** $i \leftarrow 1$ ***to*** *max_try* **do**
5 **for** $j \leftarrow 1$ ***to*** *max_itr* **do**
6 optimally threshold $\mathbf{u}$ to a binary vector $\mathbf{u}_*$ so that
7 $error \leftarrow \|\mathbf{u}_* - \min_1(\mathbf{d}_*)\|_2^2 + (\mathbf{1} \bullet (1 - \min_1(Q_c(1-\mathbf{u}_*^\delta))))$
8 is minimum where $\mathbf{d}_* = D(1-\min_1(Q(1-\mathbf{u}_*^\delta))$
9 **if** *error = 0* **then**
  break
10 Update $\mathbf{u}$ by (12)
11 **if** *error = 0* **then**
  break
12 perturbate $\mathbf{u}$ to escape from a local minimum
13 **return** $\mathbf{u}_*$

($1 \leqslant i \leqslant n$) where $\mathcal{N}(0,1)$ is the standard normal distribution. Lines 6,7 and 8 collectively perform thresholding of $\mathbf{u}$ into a binary $\mathbf{u}_*$. As the inner loop repeats, J_{SU+c} becomes smaller and smaller and so do J_{sq} and J_{nrm} in J_{SU}. J_{sq} being small means $\mathbf{u}$ is close to a supported model of P while J_{nrm} being small means each element of $\mathbf{u}$ is close to $\{1,0\}$. So binarization $\mathbf{u}_* = [\mathbf{u} \geqslant \theta]$ with an appropriate threshold θ[9] has a good chance of yielding a binary $\mathbf{u}_*$ representing a supported model of P satisfying constraints represented by Q_c. It may happen that the inner loop fails to find a solution. In such case, we retry another j-loop with perturbated $\mathbf{u}$ at line 12. There $\mathbf{u}$ is perturbated by $\mathbf{u} \leftarrow 0.5(\mathbf{u}+\Delta+0.5)$ where $\Delta \sim \mathcal{N}(0,1)$ before the next j-loop.

3.6. Connection to neural network computation

At this point, it is quite interesting to see the connection of our approach to neural network computation. In (6), we compute M and $\mathbf{d} = DM$. We point out that the computation of this $\mathbf{d}$ is nothing but the output of a forward process of a single layer ReLU network from an input vector $\mathbf{u}$. Consider the computation of $M = (1 - \min_1(Q(1-\mathbf{u}^\delta)))$. We rewrite this using $1-\min(x,1) = \mathrm{ReLU}(1-x)$ to

$$
\begin{aligned}
M &= 1 - \min_1(Q^{(1)}(1-\mathbf{u}) + Q^{(2)}\mathbf{u}) \\
&= \mathrm{ReLU}(W\mathbf{u}+\mathbf{b}) \\
&\quad \text{where } \; Q = [Q^{(1)}\, Q^{(2)}], W = Q^{(1)} - Q^{(2)}, \mathbf{b} = 1 - Q^{(1)}\mathbf{1}
\end{aligned}
$$

So M is the output of a ReLU network having a weight matrix $W = Q^{(1)} - Q^{(2)}$ and a bias vector $\mathbf{b} = 1 - Q^{(1)}\mathbf{1}$. Then $\min_1(\mathbf{d}) = \min_1(DM) = \min_1(D \cdot \mathrm{ReLU}(W\mathbf{u}+\mathbf{b}))$ is the output of a ReLU network with a single hidden layer and a linear output layer represented by D having $\min_1(\cdot)$ as activation function. Also when we compute a supported model $\mathbf{u}$, we minimize J_{SU+c} (6) which contains an MSE error term $J_{sq} = \|\min_1(\mathbf{d}) - \mathbf{u}\|^2$ using $J_{a_{SU+c}}$ (11). This is precisely back propagation from learning data $\mathbf{u}$.

Thus we may say that our approach is an integration of ASP semantics and neural computation and provides a neuro-symbolic [23] way of ASP computation. Nonetheless, there is a big difference. In standard neural network architecture, a weight matrix W and a bias vector $\mathbf{b}$ are independent. In our setting, they are interdependent and they faithfully reflect the logical structure of a program.

[9] Currently given $\mathbf{u}$, we divide the interval $[\min(\mathbf{u}), \max(\mathbf{u})]$ into 20 equally distributed notches and use each notch as a threshold value θ.

4. Computing stable models in vector spaces

4.1. Loop formulas and stable models

Let $P=(D,Q)$ be a matricized program in a set of atoms $\mathcal{A}=\{a_1,\ldots,a_n\}$ having m rules $\{a_{i_1}\leftarrow G_1,\ldots,a_{i_m}\leftarrow G_m\}$ where $D\in\{1,0\}^{n\times m}$ and $Q\in\{1,0\}^{m\times 2n}$. We assume atoms and rules are ordered as indicated.

Computing a supported model of P is equivalent to computing any binary fixedpoint $\mathbf{u}\in\{1,0\}^n$ such that $\mathbf{u}=\min_1(D(1-\min_1(Q(1-\mathbf{u}^\delta))))$ in vector spaces and in this sense, it is conceptually simple (though NP-hard). Contrastingly since stable models are a proper subclass of supported models, if one wishes to obtain precisely stable models through fixedpoint computation, the exclusion of non-stable models is necessary. Lin-Zhao's theorem [1] states that I is a stable model of P iff I is a supported model of P and satisfies a set of formulas called *loop formula*s associated with P.

Let $L=\{h_1,\ldots,h_k\}\subseteq\mathcal{A}$ be a loop in P. Recall that L is a set of atoms which are strongly connected in the positive dependency graph of P[10]. A support rule for h is a rule $h\leftarrow H$ such that $H^+\cap L=\emptyset$. H is called a support body for L. Introduce a loop formula for L[11] by

$$LF(L)=(h_1\wedge\cdots\wedge h_p)\rightarrow(H_1\vee\cdots\vee H_q)\quad\text{where }\{H_1,\ldots,H_q\}\text{ are support bodies for }L. \tag{13}$$

Then define *loop formulas associated with* P as $LF(P)=\{LF(L)\mid L\text{ is a loop in }P\}$. Logically $LF(P)$ is treated as the conjunction of its elements and we sometimes call it the loop formula associated with P. Now we evaluate $LF(P)$ by a real vector $\mathbf{u}\in\mathbb{R}^n$. Introduce an external support matrix $E_s\in\{1,0\}^{n\times m}$ by $E_s(i,j)=1$ if there is a support rule $a_i\leftarrow G_j$ for $a_i\in\mathcal{A}$, else $E_s(i,j)=0$ $(1\leqslant i\leqslant n, 1\leqslant j\leqslant m)$. Suppose there are t loops $\{L_1,\ldots,L_t\}$ in P. Introduce a loop matrix $L_{oop}\in\{1,0\}^{t\times m}$ such that $L_{oop}(s,j)=1$ if the s-th loop L_s has G_j as a support body for L_s, else $L_{oop}(s,j)=0$ $(1\leqslant s\leqslant t)$. Evaluate J_{LF} by $\mathbf{u}$ as follows.

$$\begin{aligned}
M&=1-\min_1(Q(1-\mathbf{u}^\delta)) && \text{: (continuous) truth values by }\mathbf{u}\text{ of the rule bodies in }P\\
L_s&=L_{oop}(s,:) && \text{: represents the }s\text{-th loop in }\{L_1,\ldots,L_t\}\\
A_s&=L_s(1-\mathbf{u})+L_sE_sM && \text{: (continuous) counts of true disjuncts by }\mathbf{u}\text{ of }LF(L_s)\\
J_{LF}&=\sum_{s=1}^{t}(1-\min_1(A_s)) && (14)
\end{aligned}$$

J_{LF} is a non-negative piecewise linear function of $\mathbf{u}$.

Proposition 5. *Let J_{LF} be defined as above. When $\mathbf{u}$ is binary, it holds that*
$J_{LF}=0$ *if-and-only-if* $\mathbf{u}\models LF(P)$.

(Proof) Suppose $J_{LF}=0$ and $\mathbf{u}$ is binary. A summand $(1-\min_1(A_s))$ in J_{LF} (14) corresponds to the s-th loop $L_s=\{h_1,\ldots,h_p\}$ and is non-negative. Consider $LF(L_s)=(h_1\wedge\cdots\wedge h_p)\rightarrow(H_1\vee\cdots\vee H_q)$ as a disjunction $\neg h_1\vee\cdots\vee\neg h_p\vee H_1\vee\cdots\vee H_q$. Then $J_{LF}=0$ implies $(1-\min_1(A_s))=0$, or equivalently $A_s\geqslant 1$. Consequently, as $\mathbf{u}$ is binary, we have $L_s(1-\mathbf{u})\geqslant 1$ or $L_sE_sM\geqslant 1$. The former means $\mathbf{u}\models\neg h_1\vee\cdots\vee\neg h_p$. The latter, $L_sE_sM\geqslant 1$, means $\mathbf{u}\models H_1\vee\cdots\vee H_q$. This is because $(E_sM)(i)$ is the number of support rules for $a_i\in\mathcal{A}$ whose bodies are true in $\mathbf{u}$ $(1\leqslant i\leqslant n)$, and hence $L_sE_sM\geqslant 1$ means some support body H_r $(1\leqslant r\leqslant q)$ for L_s is true in $\mathbf{u}$. So in either case $\mathbf{u}\models LF(L_s)$. Since s is arbitrary, we have $\mathbf{u}\models LF(P)$. The converse is straightforward and omitted. Q.E.D.

[10] In the case of a singleton loop $L=\{h\}$, we specifically require, following [1], that h has a self-loop, i.e. there must be a rule of the form $h\leftarrow h\wedge H$ in P.

[11] In the original form, the antecedent of $LF(L)$ is a disjunction $(h_1\vee\cdots\vee h_p)$ [1]. Later it is shown that Lin-Zhao's theorem also holds for the AND type $LF(L)$ [24]. We choose this AND type $LF(L)$ as it is easier to satisfy.

The Jacobian $J_{a_{LF}}$ of J_{LF} is computed as follows.

$$
\begin{aligned}
N &= Q(1-\mathbf{u}^{\delta}) \\
N_s &= L_s(1-\mathbf{u}) \\
M_s &= \min_1(N_s) \\
J_{a_{LF}} &= \frac{\partial J_{LF}}{\partial \mathbf{u}} = \sum_{s=1}^{t} -\left(\frac{\partial \min_1(A_s)}{\partial \mathbf{u}}\right) \\
&= -\sum_{s=1}^{t}[A_s \leqslant 1]\left(\left(\frac{\partial M_s}{\partial \mathbf{u}}\right) + L_s E_s \left(\frac{\partial M}{\partial \mathbf{u}}\right)^T\right) \\
&= \sum_{s=1}^{t}[A_s \leqslant 1]\left([N_s \leqslant 1]L_s^T + (((L_s E_s) \odot [N \leqslant 1]^T)(Q^{(2)} - Q^{(1)})^T\right) \qquad (15)
\end{aligned}
$$

Here $Q = [Q^{(1)}\, Q^{(2)}]$ and L_s, A_s and M are computed by (14).

Now introduce a new cost function $J_{SU+c+LF}$ by (16) that incorporates J_{LF} and compute its Jacobian $J_{a_{SU+c+LF}}$ by (11).

$$J_{SU+c+LF} = J_{SU+c} + \ell_4 \cdot J_{LF} \ \text{ where } \ \ell_4 > 0 \qquad (16)$$

$$J_{a_{SU+c+LF}} = J_{a_{SU+c}} + \ell_4 \cdot J_{a_{LF}} \qquad (17)$$

By combining **Proposition** 4, 5 and Lin-Zhao's theorem [1], the following is obvious.

Proposition 6. $\mathbf{u}$ *is a stable model of* P *satisfying constraints represented by* Q_c *if-and-only-if* $\mathbf{u}$ *is a root of* $J_{SU+c+LF}$.

We compute such $\mathbf{u}$ by Newton's method using **Algorithm 1** with a modified update rule (12) such that J_{SU+c} and $J_{a_{SU+c}}$ are replaced by $J_{SU+c+LF}$ and $J_{a_{SU+c+LF}}$ respectively.

When a program P is tight [15], for example when rules have no positive literal in their bodies, P has no loop and hence LF is empty. In such case, we directly minimize J_{SU+c} instead of using $J_{SU+c+LF}$ with the empty LF.

4.2. LF heuristics

Minimizing $J_{SU+c+LF}$ is a general way of computing stable models under constraints. It is applicable to any program and gives us a theoretical framework for computing stable models in an end-to-end way without depending on symbolic systems. However there can be exponentially many loops and they make the computation of J_{LF} (14) extremely difficult or practically impossible. To mitigate this seemingly insurmountable difficulty, we propose two heuristics which use a subset of loop formulas.

LF_{max}: The first heuristics is LF_{max} heuristics. We consider only a set LF_{max} of loop formulas associated with SCCs in the positive dependency graph $\mathrm{pdg}(P) = (V,E)$ of a program P. In the case of a singleton SCC {a}, 'a' must have a self-loop in $\mathrm{pdg}(P)$. We compute SCCs in $O(|E|+|V|)$ time by Tarjan's algorithm [25].

LF_{min}: In this heuristics, instead of SCCs (maximal strongly connected subgraphs), we choose minimal strongly connected subgraphs, i.e. cycle graphs. Denote by LF_{min} the set of loop formulas associated with cycle graphs in pdg(P). We use an enumeration algorithm described in [26] to enumerate cycles and construct LF_{min} due to its simplicity.

We remark that although LF_{max} and LF_{min} can exclude (some of) non-stable models, they do not necessarily exclude all of non-stable models. However, the role of loop formulas in our framework is entirely different from the one in symbolic ASP. Namely, the role of LF in our framework is not to logically reject non-stable models but to guide the search process by their gradient information in the continuous search space. Hence, we expect, as actually observed in experiments in the next section, some loop formulas have the power of guiding the search process to a root of $J_{SU+c+LF}$.

4.3. Precomputation

We introduce here precomputation. The idea is to remove atoms from the search space which are false in every stable model. It downsizes the program and realizes faster model computation.

When a program P in a set $\mathcal{A} = \text{atom}(P)$ is given, we transform P to a definite program P^+ by removing all negative literals from the rule bodies in P. Since $P^+ \supseteq P^I$ holds as a set of rules for any model I, we have $LM(P^+) \supseteq LM(P^I)$ where $LM(P)$ denotes the least model of a definite program P. When I is a stable model, $LM(P^I) = I$ holds and we have $LM(P^+) \supseteq I$. By taking the complements of both sides, we can say that if an atom a is outside of $LM(P^+)$, i.e. if a is false in $LM(P^+)$, so is a in *any* stable model I of P. Thus, by precomputing the least model $LM(P^+)$, we can remove a set of atoms $\mathcal{F}_P = \mathcal{A} \setminus LM(P^+)$ from our consideration as they are known to be false in any stable model. We call $\mathcal{F}_P$ *stable false atom*s. Of course, this precomputation needs additional computation of $LM(P^+)$ but it can be done in linear time proportional to the size of P^+, i.e. the total number of occurrences of atoms in P^+ [27][12]. Accordingly precomputing the least model $LM(P^+)$ makes sense if the benefit of removing stable false atoms from the search space outweighs linear time computation for $LM(P^+)$, which is likely to happen when we deal with programs with positive literals in the rule bodies.

More concretely, given a program P and a set of constraints C, we can obtain downsized ones, P' and C', as follows.

Step 1: Compute the least model $LM(P^+)$ and the set of stable false atoms $\mathcal{F}_P = \text{atom}(P) \setminus LM(P^+)$.
Step 2: Define

$$G' = \text{conjunction } G \text{ with negative literals } \{\neg a \in G \mid a \in \mathcal{F}_P\} \text{ removed}$$

$$P' = \{a \leftarrow G' \mid a \leftarrow G \in P, a \notin \mathcal{F}_P, G^+ \cap \mathcal{F}_P = \emptyset\} \text{ where } G^+ = \text{positive literals in } G \tag{18}$$

$$C' = \{\leftarrow G' \mid \leftarrow G \in C, G^+ \cap \mathcal{F}_P = \emptyset\} \tag{19}$$

Proposition 7. *Let P' and C' be respectively the program (18) and constraints (19). Also let I' be a model over atom(P'). Expand I' to a model I over atom(P) by assuming every atom in $\mathcal{F}_P$ is false in I. Then*

I' is a stable model of P' satisfying constraints C' if-and-only-if I is a stable model of P satisfying constraints C.

(Proof) We prove first I' is a stable model of P' if-and-only-if I is a stable model of P. To prove it, we prove $LM(P'^{I'}) = LM(P^I)$ as set.

Let $a \leftarrow G'^+$ be an arbitrary rule in $P'^{I'}$. Correspondingly there is a rule $a \leftarrow G'$ in P' such that $I' \models G'^-$. So there is a rule $a \leftarrow G$ in P such that $G' = G \setminus \{\neg b \mid b \in \mathcal{F}_P\}$ and $G^+ \cap \mathcal{F}_P = \emptyset$. $I' \models G'^-$ implies $I \models G^-$ by construction of I from I'. So $a \leftarrow G^+$ is contained in P^I, which means $a \leftarrow G'^+$ is contained in P^I because $G'^+ = G^+$ (recall that

[12] We implemented the linear time algorithm in [27] linear algebraically using vector and matrix and confirmed its linearity.

$G' = G \setminus \{\neg b \mid b \in \mathcal{F}_P\}$ and G' and G have the same set of positive literals). Thus since $a \leftarrow G'^{+}$ is an arbitrary rule, we conclude $P'^{I'} \subseteq P^I$, and hence $LM(P'^{I'}) \subseteq LM(P^I)$.

Now consider $a \in LM(P^I)$. There is an SLD derivation for $\leftarrow a$ from P^I. Let $b \leftarrow G^+ \in P^I$ be a rule used in the derivation which is derived from the rule $b \leftarrow G \in P$ such that $I \models G^-$. Since $P^I \subseteq P^+$, we have $LM(P^I) \subseteq LM(P^+)$ and hence $LM(P^I) \cap \mathcal{F}_P = \emptyset$, i.e., $LM(P^I)$ contains no stable false atom. So $b \notin \mathcal{F}_P$ and $G^+ \cap \mathcal{F}_P = \emptyset$ because every atom in the SDL derivation must belong in $LM(P^I)$. Accordingly $b \leftarrow G' \in P'$. On the other hand, $I \models G^-$ implies $I' \models G'^-$. So $b \leftarrow G'$ is in P' and $b \leftarrow G'^{+}$ is in $P'^{I'}$. Therefore $b \leftarrow G^+$ is in $P'^{I'}$ because $G'^{+} = G^+$. Thus every rule used in the derivation for $\leftarrow a$ from P^I is also a rule contained in $P'^{I'}$, which means $a \in LM(P'^{I'})$. Since a is arbitrary, it follows that $LM(P^I) \subseteq LM(P'^{I'})$. By putting $LM(P'^{I'}) \subseteq LM(P^I)$ and $LM(P^I) \subseteq LM(P'^{I'})$ together, we conclude $LM(P^I) = LM(P'^{I'})$.

Then, if I' is a stable model of P', we have $I' = LM(P'^{I'}) = LM(P^I)$ as set. Since $I = I'$ as set, we have $I = LM(P^I)$ as set, which means I is a stable model of P. Likewise when I is a stable model of P, we have $I = LM(P^I) = LM(P'^{I'})$ and $I = I'$ as set. So $I' = LM(P'^{I'})$ as set and I' is a stable model of P'. We can also similarly prove that I' satisfies C' if-and-only-if I satisfies C. So we are done. Q.E.D.

5. Programming examples

In this section, we apply our ASP approach to examples as a proof of concept and examine the effectiveness of precomputation and heuristics. Since large scale computing is out of scope in this paper, the program size is mostly small[13].

5.1. The 3-coloring problem

We first deal with the 3-coloring problem. Suppose we are given a graph G_1. The task is to color the vertices of the graph blue, red and green so that no two adjacent vertices have the same color like (b) in Fig. 1.

(a) Graph G_1 (b) A 3-coloring

Fig. 1. 3-coloring problem

There are four nodes $\{a,b,c,d\}$ in the graph G_1. We assign a set of three color atoms (Boolean variables) to each node to represent their color. For example, node a is assigned three color atoms $\{a_1(\text{red}), a_2(\text{blue}), a_3(\text{green})\}$. We need to represent two facts by these atoms.

– Each node has a unique color chosen from $\{red, blue, green\}$. So color atoms assigned to each node are in an XOR relation. We represent this fact by a tight program P_1 below containing three rules for each node.

[13] Matricized programs in this paper are all written in GNU Octave 6.4.0 and run on a PC with Intel(R) Core(TM) i7-10700@2.90GHz CPU with 26GB memory.

$$P_1 = \begin{cases} a_1 \leftarrow \neg a_2 \wedge \neg a_3,\ a_2 \leftarrow \neg a_3 \wedge \neg a_1,\ a_3 \leftarrow \neg a_1 \wedge \neg a_2 \\ b_1 \leftarrow \neg b_2 \wedge \neg b_3,\ b_2 \leftarrow \neg b_3 \wedge \neg b_1,\ b_3 \leftarrow \neg b_1 \wedge \neg b_2 \\ c_1 \leftarrow \neg c_2 \wedge \neg c_3,\ c_2 \leftarrow \neg c_3 \wedge \neg c_1,\ c_3 \leftarrow \neg c_1 \wedge \neg c_2 \\ d_1 \leftarrow \neg d_2 \wedge \neg c_3,\ d_2 \leftarrow \neg d_3 \wedge \neg d_1,\ d_3 \leftarrow \neg d_1 \wedge \neg d_2 \end{cases} \tag{20}$$

– Two nodes connected by an edge must have a different color. We represent this fact in terms of constraints.

$$C_1 = \begin{cases} \leftarrow a_1 \wedge b_1,\ \leftarrow a_2 \wedge b_2,\ \leftarrow a_3 \wedge b_3 \\ \leftarrow a_1 \wedge c_1,\ \leftarrow a_2 \wedge c_2,\ \leftarrow a_3 \wedge c_3 \\ \leftarrow b_1 \wedge c_1,\ \leftarrow b_2 \wedge c_2,\ \leftarrow b_3 \wedge c_3 \\ \leftarrow b_1 \wedge d_1,\ \leftarrow b_2 \wedge d_2,\ \leftarrow b_3 \wedge d_3 \\ \leftarrow d_1 \wedge c_1,\ \leftarrow d_2 \wedge c_2,\ \leftarrow d_3 \wedge c_3 \end{cases} \tag{21}$$

Assuming an ordering of atoms $\{a_1, a_2, a_3, \ldots, d_1, d_2, d_3\}$, the normal logic program P_1 shown in (20) is matricized to $P_1 = (D_1, Q_1)$ where D_1 is a (12×12) binary identity matrix (because there are 12 atoms and each atom has just one rule) and Q_1 is a (12×24) binary matrix shown in (22). Constraints listed in (21) are a matricized to a (15×12) constraint matrix Q_{C_1} (23). In (22) and (23), a for example stands for a triple $(a_1\, a_2\, a_3)$ and $\neg a$ for $(\neg a_1\, \neg a_2\, \neg a_3)$.

$$Q_1 = \begin{array}{c} \begin{matrix} a & b & c & d & \neg a & \neg b & \neg c & \neg d \end{matrix} \\ \begin{bmatrix} & & & & H_3 & & & \\ & & & & & H_3 & & \\ & & & & & & H_3 & \\ & & & & & & & H_3 \end{bmatrix} \end{array} \quad \text{where } H_3 = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \tag{22}$$

$$Q_{C_1} = \begin{array}{c} \begin{matrix} a & b & c & d \end{matrix} \\ \begin{bmatrix} E_3 & E_3 & & \\ E_3 & & E_3 & \\ & E_3 & E_3 & \\ & E_3 & & E_3 \\ & & E_3 & E_3 \end{bmatrix} \end{array} \quad \text{where } E_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{23}$$

We run **Algorithm 1** on program P_1 with constraints C_1 to find a supported model (solution) of P_1 satisfying C_1[14].

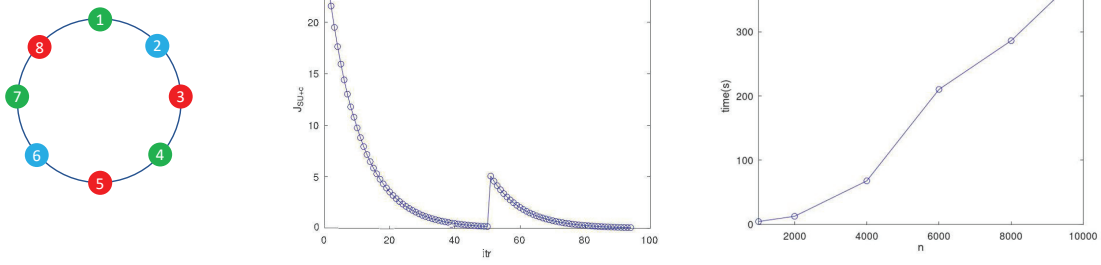
Table 1
Time and the number of solutions

time(s)	#solutions
6.7(0.7)	5.2(0.9)

To measure time to find a model, we conduct ten trials[15] of running **Algorithm 1** with $max\_try = 20$, $max\_itr = 50$, $\ell_2 = \ell_3 = 0.1$ and take the average. The result is 0.104s(0.070) on average. Also to check the ability of finding different solutions, we perform ten trials of **Algorithm 1**[16]and count the number of different solutions in the returned solutions. #solutions in Table 1 is the average of ten such measurements. Due to naive implementation, computation is slow but the number of different solutions, 5.2 on average, seems rather high considering there are six solutions.

[14]Since P_1 is a tight program, every supported model of P_1 is a stable model and vice versa.
[15]One trial consists of $max\_itr \times max\_try$ parameter updates.
[16]without *another solution constraint* introduced in Section 5.2

Next we check the scalability of our approach by a simple problem. We consider the 3-coloring of a cycle graph like (a) in Fig. 2. In general, given a cycle graph that has n nodes, we encode its 3-coloring problem as in the previous example by a matricized program $P_2 = (D_2, Q_2)$ and a constraint matrix Q_{C_2} where $D_2(3n \times 3n)$ is an identity matrix and $Q(3n \times 6n)$ and $Q_{C_2}(3n \times 6n)$ represent respectively rules and constraints. There are $2^n + 2(-1)^n$ solutions ($n \geqslant 3$) in 2^{3n} possible assignments for $3n$ atoms. So the problem will be exponentially difficult as n goes up.



(a) A cycle graph (b) Minimization of J_{SU+c} with retry (c) Scalability

Fig. 2. Convergence and scalability

The graph (b) in Fig. 2 is an example of convergence curve of J_{SU+c} by **Algorithm 1** with $n = 10$, $max\_try = 100$, $max\_itr = 50$. The curve tells us that in the first cycle of j-loop, the inner for loop of **Algorithm 1**, no solution is found after $max\_itr = 50$ iterations of update of continuous assignment vector $\mathbf{u}$. Then perturbation is given to $\mathbf{u}$ which causes a small jump of J_{SU+c} at $itr = 51$ and the second cycle of j-loop starts and this time a solution is found after dozens of updates by thresholding $\mathbf{u}$ to a binary vector $\mathbf{u}_*$.

The graph (c) in Fig. 2 shows the scalability of computation time to find a solution up to $n = 10000$. We set $max\_try = 100$, $max\_itr = 2000$ and plot the average of ten measurements of time to find a solution. The graph seems to indicate good linearity w.r.t. n up to $n = 10000$.

5.2. *The Hamiltonian cycle problem, precomputation and another solution constraint*

A Hamiltonian cycle (HC) is a cycle in a graph that visits every vertex exactly once and the Hamiltonian cycle problem is to determine if an HC exists in a given graph. It is an NP-complete problem and has been used as a programming example since the early days of ASP. Initially it is encoded by a non-tight program containing positive recursion [3]. Later a way of encoding by a program that is not tight but tight on its completion models is proposed [28]. We here introduce yet another encoding by a tight ground program inspired by SAT encoding proposed in [29] where Zhou showed that the problem is solvable by translating six conditions listed in Fig. 3 into a SAT problem[17].

In what follows, we assume vertices are numbered from 1 to N = the number of vertices in a graph. We use $i \to j$ to denote an edge from vertex i to vertex j and $H_{i,j}$ to indicate there exists an edge from i to j in an HC. $U_{j,q}$ means vertex j is visited at time q ($1 \leqslant j,q \leqslant N$) and one_of$(a_1,\ldots,a_k)$ means that one of $\{a_1,\ldots,a_k\}$ is true. We translate these conditions into a program $P_3 = \{(1),(2),(3)\}$ and constraints $C_3 = \{(4),(5),(6)\}$. To be more precise, the first condition (1) is translated into a tight program just like a program P_1 (20). The conditions $\{(2),(3)\}$ constitute a tight definite program. Constraints $C_2 = \{(4),(5),(6)\}$ are encoded as a set of implications of the form $\leftarrow L_1 \wedge \cdots \wedge L_k$ where $L_1,\ldots,L_k$ are literals. A set of $U_{j,q}$ atoms contained in a stable model of P_2 satisfying C_2 gives an HC.

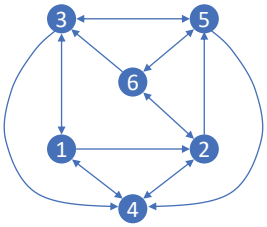
[17] Actually he listed seven conditions to be encoded as a SAT problem. However, one of them is found to be redundant and we use the remaining six conditions.

conditions			meaning
(1)	one_of$(H_{i,j_1},\ldots,H_{i,j_k})$	:	one of outgoing edges $\{i\to j_1,\ldots,i\to j_k\}$ from vertex i is in an HC
(2)	$U_{j,q}\leftarrow H_{i,j}\wedge U_{i,q-1}$	:	if edge $i\to j$ is in an HC and vertex i is visited at time $q-1$, vertex j is visited at time q $(1\leqslant i,j,q\leqslant N)$
(3)	$U_{1,1}$	:	vertex 1 is visited at time 1 (starting point)
(4)	one_of$(H_{i_1,j},\ldots,H_{i_k,j})$	:	one of incoming edges $\{i_1\to j,\ldots,i_k\to j\}$ to vertex j is in an HC
(5)	$\leftarrow H_{i,1}\wedge\neg U_{i,N}$	:	if edge $i\to 1$ is in an HC, vertex i is visited at time N $(2\leqslant i\leqslant N)$
(6)	one_of$(U_{i,1},\ldots,U_{i,N})$	:	vertex i is visited once $(1\leqslant i\leqslant N)$

Fig. 3. Conditions for SAT encoding

We apply the above encoding to a simple Hamiltonian cycle problem for a graph G_2 in Fig. 4[18]. There are six vertices and six HCs[19]. To solve this HC problem, we matricize P_3 and C_3. There are 36 $H_{i,j}$ atoms $(1\leqslant i,j\leqslant 6)$ and 36 $U_{j,q}$ atoms $(1\leqslant j,q\leqslant 6)$. So there are 72 atoms in total. $P_3=\{(1),(2),(3)\}$ contains 197 rules in these 72 atoms and we translate P_3 into a pair of matrices (D_3,Q_3) where D_3 is a 72×197 binary matrix for disjunctions[20]and Q_3 is a 197×144 matrix for conjunctions (rule bodies). Likewise $C_3=\{(4),(5),(6)\}$ is translated into a constraint matrix Q_{C_3} which is a 67×144 binary matrix. Then our task is to find a root $\mathbf{u}$ of J_{SU+c} (10) constructed from these D_3, Q_3 and Q_{C_3} in a 72 dimensional vector space by minimizing J_{SU+c} to zero.

We apply precomputation in the previous section to $P_3=(D_3,Q_3)$ and Q_{C_3} to reduce program size. It takes 2.3ms and detects 32 false stable atoms. It outputs a precomputed program $P'_3=(D'_3,Q'_3)$ and a constraint matrix Q'_{C_3} of size $D'_3(40\times 90)$, $Q'_3(90\times 80)$ and $Q'_{C_3}(52\times 80)$ respectively, which is $1/4$ or $1/2$ of the original size. So precomputation removes 45% of atoms from the search space and returns much smaller matrices.



(a) Graph G_2

	no precomp.	precomp.
time(s)	2.08(2.01)	0.66(0.52)
#solutions	4.9	5.7

(b) Time and the number of different solutions

Fig. 4. A HC problem

We run **Algorithm 1** on $P_3=(D_3,Q_3)$ with Q_{C_3} (no precomputation) and also on $P'_3=(D'_3,Q'_3)$ with Q'_{C_3} (precomputation) using $max\_try=20$, $max\_itr=200$ and $\ell_2=\ell_3=0.1$ and measure time to find a solution, i.e. stable model satisfying constraints. The result is shown by Table (b) in Fig. 4 as time(s) where time(s) is an average of ten trials. Figures in the table, 2.08s vs. 0.66s, clearly demonstrate the usefulness of precomputation.

In addition to computation time, we examine the search power of different solutions in our approach by measuring the number of obtainable solutions. More concretely, we run **Algorithm 1** seven times, and each time a stable model is obtained as a conjunction $L_1\wedge\ldots\wedge L_{72}$ of literals, we add a new constraint $\leftarrow L_1\wedge\ldots\wedge L_{72}$ to previous constraints, thereby forcibly computing a new stable model in next trial. We call such use of constraint *another solution constraint*. Since there are at most six solutions, the number of solutions obtained by seven trials is at most six. We repeat a batch of seven trials ten times and take the average of the number of solutions obtained by

[18] G_2 is taken from: Section 4.2 in A User's Guide to gringo, clasp, clingo, and iclingo (`http://wp.doc.ic.ac.uk/arusso/wp-content/uploads/sites/47/2015/01/clingo_guide.pd`).

[19] They are $1\to 2\to 5\to 6\to 3\to 4\to 1$, $1\to 2\to 6\to 3\to 5\to 4\to 1$, $1\to 2\to 6\to 5\to 3\to 4\to 1$, $1\to 3\to 5\to 6\to 2\to 4\to 1$, $1\to 4\to 2\to 5\to 6\to 3\to 1$, $1\to 4\to 2\to 6\to 5\to 3\to 1$.

[20] For example, for each $U_{j,q}$ $(1\leqslant j,q\leqslant 6)$, condition (2) generates six rules $\{U_{j,q}\leftarrow H_{i,j}\wedge U_{i,q-1}\mid 1\leqslant i\leqslant 6\}$.

each batch. The average is denoted as #solutions in Table (b) which indicates that 5.7 solutions out of 6, almost all solutions, are obtained by seven trials using another solution constraint.

Summing up, figures in Table (b) exemplify the effectiveness of precomputation which significantly reduces computation time and returns a more variety of solutions when combined with another solution constraint.

5.3. LF heuristics and precomputation on loopy programs

So far we have been dealing with tight programs which have no loop and hence have no loop formulas. We here deal with non-tight programs containing loops and examine how LF heuristics, LF_{max} and LF_{min}, introduced in the previous section work. We use an artificial non-tight program P_{4_n} (with no constraint) shown below that has exponentially many loops.

$$P_{4\_n} = \begin{cases} a_0 \leftarrow a_1 \wedge \ldots \wedge a_n \\ a_0 \leftarrow \neg a_{n+1} \\ \ldots \\ a_{2i-1} \leftarrow a_0 \vee a_{2i} \quad \text{for } i : 1 \leqslant i \leqslant n/2 \\ a_{2i} \leftarrow a_0 \vee a_{2i-1} \quad \text{for } i : 1 \leqslant i \leqslant n/2 \\ \ldots \\ a_{n+1} \leftarrow a_{n+1} \end{cases}$$

For an even n, P_{4_n} program has $n+2$ atoms $\{a_0, a_1, \ldots, a_n, a_{n+1}\}$, $2^{n/2}+1$ supported models and one stable model $\{a_0, a_1, \ldots, a_n\}$. There are $n/2+1$ minimal loops $\{a_1, a_2\}, \ldots, \{a_{n-1}, a_n\}, \{a_{n+1}\}$ and a maximal loop $\{a_0, a_1, \ldots, a_n\}$. The set of loop formulas for LF heuristics are computed as follows.

$$LF_{max} = \{(a_0 \wedge a_1 \wedge \ldots \wedge a_n) \rightarrow \neg a_{n+1}, a_{n+1} \rightarrow \bot\}$$
$$LF_{max} = \{(a_1 \wedge a_2) \rightarrow a_0, \ldots, (a_{n-1} \wedge a_n) \rightarrow a_0, a_{n+1} \rightarrow \bot\}$$

Note that although there are $2^{n/2}+1$ supported models, there is only one stable model. So LF_{max} and LF_{min} are expected to exclude $2^{n/2}$ supported models.

After translating P_{4_n} into a matricized program $P_{4\_n} = (Q_{4\_n}, D_{4\_n})$ where Q_{4_n} is a $(2n+3) \times (2n+4)$ binary matrix and D_{4_n} is a $(n+2) \times (2n+3)$ binary matrix respectively, we compute a stable model of P_{4_n} for various n by **Algorithm 1** that minimizes $J_{SU+c+LF}$ (16) with coefficient $\ell_3 = 0$ for the constraint term (because of no use of constraints) using Jacobian $J_{a_{SU+c+LF}}$ (11).

Since all supported models of P_{4_n} except for one stable model are non-stable, even if LF_{max} and LF_{min} are used to guide the search process towards a stable model, **Algorithm 1** is likely to return a non-stable model. We can avoid such situation by the use of another solution constraint.

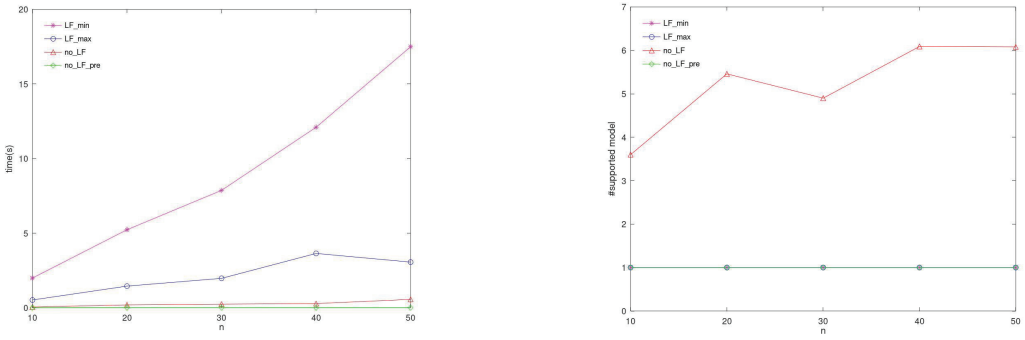
Table 2
The effect of another solution constraint

another solution constraint	time(s)	#trials
not used	11.46(0.41)	10^4(0)
used	0.09(0.13)	3.5(1.6)

To verify it, we examine the pure effect of another solution constraint that guides the search process to compute a model different from previous ones. Without using LF_{max} or LF_{min} heuristics, we repeatedly run **Algorithm 1** with/without another solution constraint for 10^4 trials with $n = 4$, $max\_try = 20$, $max\_itr = 50$, $\ell_2 = \ell_3 = 0.1$ and measure time to find a stable model and count the number of trials until then. We repeat this experiment ten times and take the average. The result is shown in Table 2.

The figure $10^4(0)$ in Table 2 in the case of no use of another solution constraint means **Algorithm 1** always exhausts 10^4 trials without finding a stable model (due to implicit bias in **Algorithm 1**). When another solution constraint is used however, it finds a stable model in 0.09s after 3.5 trials on average. Thus Table 2 demonstrates the necessity and effectiveness of another solution constraint to efficiently explore the search space.

We next compare the effectiveness of LF heuristics and that of precomputation under another solution constraint. For $n = 10, \ldots, 50$, we repeatedly run **Algorithm 1** using $J_{SU+c+LF}$ with $max\_try = 10, max\_itr = 100$ on matricized $P_{4\_n} = (Q_{4\_n}, D_{4\_n})$ (and no constraint matrix) to compute supported (stable) models. Coefficients in $J_{SU+c+LF}$ are set to $\ell_2 = 0.1, \ell_3 = 0, \ell_4 = 1$. To be more precise, for each n and each case of LF_{max}, LF_{min}, precomputation (without $\{LF_{max}, LF_{min}\}$) and no $\{LF_{max}, LF_{min}, \text{precomputation}\}$, we run **Algorithm 1** at most 100 trials to measure time to find a stable model and count the number of supported models computed till then. We repeat this computation ten times and take the average and obtain graphs in Fig. 5.



(a) Time to find a stable model

(b) #computed models

Fig. 5. The effect of LF heuristics and precomputation on program P_{4_n}

In Fig. 5, no_LF means no use of $\{LF_{max}, LF_{min}\}$ heuristics. Also no_LF_pre means no_LF is applied to precomputed P_{4_n}[21].

We can see from graph (a) in Fig. 5 that computation time is $LF_{min} > LF_{max} >$ no_LF $>$ no_LF_pre. This means that using LF heuristics is not necessarily a good policy. They might cause extra computation to reach the same model. Concerning the number of non-stable models computed redundantly, graph (b) in Fig. 5 tells us that LF_{min} allows computing redundant non-stable models but the rest, LF_{max}, no_LF and no_LF_pre, return a stable model without computing redundant non-stable models. This shows first that LF_{max} works correctly to suppress the computation of non-stable models and second that the LF_{min} heuristics works adversely, i.e. guiding the search process away from the stable model. This somewhat unexpected result indicates the need of (empirical) choice of LF heuristics.

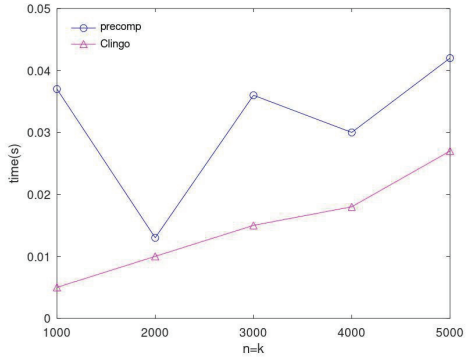
Finally to examine the effectiveness of precomputation more precisely, we apply precomputation to a more complex program P_{5_nk}. It is a modification of P_{4_n} by adding self-loops of k atoms as illustrated by (a) in Fig. 6. The addition of self-loop causes the choice of a_{n+j} $(1 \leqslant j \leqslant k)$ being true or being false in the search process. P_{5_nk} has $(2^{n/2} - 1)(2^k - 1) + 1$ supported models but has just one stable model $\{a_0, a_1, \ldots, a_n\}$.

We compute a stable model by running **Algorithm 1** on precomputed P_{5_nk} without using LF heuristics up to $n = k = 5000$. When precomputation is applied to P_{5_nk} where $n = k = 5000$, it detects 5000 false stable atoms and downsizes the matrices in $P_{5\_nk} = (D_{5\_nk}, Q_{5\_nk})$ from $D_{5\_nk}(10001 \times 15002)$ to $D'_{5\_nk}(5001 \times 10002)$ and from $Q_{5\_nk}(15002 \times 20002)$ to $Q'_{5\_nk}(10002 \times 10002)$. Thus precomputed $P'_{5\_nk} = (D'_{5\_nk}, Q'_{5\_nk})$ is downsized to 1/3 of the original P_{5_nk}.

[21] Precomputation takes 0.006s and removes only one stable false atom. So precomputation is not helpful in the current case.

$$P_{5\_nk} = \begin{cases} a_0 & \leftarrow a_1 \wedge \ldots \wedge a_n \\ a_0 & \leftarrow \neg a_{n+1} \wedge \ldots \wedge \neg a_{n+k} \\ \ldots & \\ a_{2i-1} & \leftarrow a_0 \vee a_{2i} \\ a_{2i} & \leftarrow a_0 \vee a_{2i-1} \\ \ldots & \\ a_{n+1} & \leftarrow a_{n+1} \\ \ldots & \\ a_{n+k} & \leftarrow a_{n+k} \end{cases}$$

(a) A non-tight program P_{5_nk}



(b) Scalability of precomputation w.r.t. P_{5_nk}

Fig. 6. Precomputation applied to program P_{5_nk}

.

We run **Algorithm 1** on P'_{5_nk} with $\ell_2 = \ell_3 = 0.1$ and $max\_try = 10$, $max\_itr = 100$ at most 100 trials to measure time to find a stable model ten times for each $n = 1000, \ldots, 5000$ and take the average. At the same time, we also run Clingo (version 5.6.2) on P_{5_nk} and similarly measure time. Graph (b) in Fig. 6 is the result. It shows that as far as computing a stable model of P_{5_nk} is concerned, our approach comes close to Clingo. However, this is due to a very specific situation that precomputation removes all false atoms $\{a_{n+1}, \ldots, a_{n+k}\}$ in the stable model of P_{5_nk} and **Algorithm 1** run on the precomputed $P'_{5\_nk} = (D'_{5\_nk}, Q'_{5\_nk})$ detects the stable model only by thresholding **u** before starting any update of **u**. So what graph (b) really suggests seems to be the importance of optimization of a program like precomputation, which is to be developed further in our approach.

6. Related work

The most closely related work is [13] and [14]. As mentioned in Section 1, our approach differs from them in three points: (1) theoretically, the exclusion of non-stable models by loop formulas, (2) syntactically, no restriction on acceptable programs and (3) practically, incorporation of constraints. Concerning performance, they happen to use the same N-negative loops program which consists of N copies (alphabetic variants) of a program $\{a \leftarrow \neg b, b \leftarrow \neg a\}$. According to [13], the success rate w.r.t. N of returning a supported model goes from one initially to almost zero at $N = 64$ in [13] while it keeps one till $N = 100$ in [14]. We tested the same program with $max\_try = 20$, $max\_itr = 100$ and observed that the success rate keeps one till $N = 10000$.

Although our approach is non-probabilistic, i.e. purely linear algebraic, there are probabilistic differentiable approaches for ASP. Differentiable ASP/SAT [10] iteratively samples a stable model by an ASP solver a la ASSAT [1]. The solver decides the next decision literal based on the derivatives of a cost function which is the MSE between the target probabilities and predicted probabilities computed from the sampled stable models via parameters associated with "parameter atoms" in a program.

NeurASP [11] uses an ASP solver to obtain stable models including "neural atoms" for a program. They are associated with probabilities learned by deep learning and the likelihood of an observation (a set of ASP constraints) is computed from them. The whole learning is carried out by backpropagating the likelihood to neural atoms to parameters in a neural network.

Similarly to NeurASP, SLASH [12] uses an ASP solver to compute stable models for a program containing "neural probabilistic predicates". Their probabilities are dealt with by neural networks and probabilistic circuits. The latter makes it possible to compute a joint distribution of the class category and data.

Independently of ASP solver based approaches mentioned above, Sato and Kojima proposed a differentiable approach to sampling supported models of (non-propositional) probabilistic normal logic programs [17, 18]. They encode programs by matrices and formulate the problem of sampling supported models as repeatedly computing a

fixedpoint of some differentiable equations. They solve the equations in vector spaces by minimizing a non-negative cost function defined by Frobenius norm.

7. Conclusion

We proposed an end-to-end approach for computing stable models satisfying given constraints. We matricized a program and constraints and formulated stable model computation as a minimization problem in vector spaces of a non-negative cost function. We obtain a stable model satisfying constraints as a root the cost function by Newton's method.

By incorporating all loop formula constraints introduced in Lin-Zhao's theorem [1] into the cost function to be minimized, we can prevent redundant computation of non-stable models, at the cost of processing exponentially many loop formulas. Hence, we introduced precomputation which downsizes a program while preserving stable model semantics and also two heuristics that selectively use loop formulas. Then we confirmed the effectiveness of our approach including precomputation and loop formula heuristics by simple examples.

Acknowledgments: This work is supported by JSPS KAKENHI Grant Number JP21H04905 and JST CREST Grant Number JPMJCR22D3.

References

[1] F. Lin and Y. Zhao, ASSAT: computing answer sets of a logic program by SAT solvers, *Artificial Intelligence* **157**(1) (2004), 115–137.
[2] M. Gelfond and V. Lifshcitz, The stable model semantics for logic programming (1988), 1070–1080.
[3] I. Niemelä, Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm **25**(3–4) (1999), 241–273–.
[4] V.W. Marek and M. Truszczyński, *Stable Models and an Alternative Logic Programming Paradigm*, in: *The Logic Programming Paradigm: A 25-Year Perspective*, K.R. Apt, V.W. Marek, M. Truszczynski and D.S. Warren, eds, Springer Berlin Heidelberg, 1999, pp. 375–398. doi:10.1007/978-3-642-60085-2_17.
[5] V. Lifschitz, What is Answer Set Programming?, in: *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3*, AAAI'08, AAAI Press, 2008, pp. 1594–1597–.
[6] I. Niemelä and P. Simons, Smodels — an implementation of the stable model and well-founded semantics for normal logic programs, in: *Logic Programming And Nonmonotonic Reasoning*, J. Dix, U. Furbach and A. Nerode, eds, Springer Berlin Heidelberg, 1997, pp. 420–429.
[7] Y. Lierler, CMODELS: SAT-Based Disjunctive Answer Set Solver, in: *Proceedings of the 8th International Conference on Logic Programming and Nonmonotonic Reasoning*, LPNMR'05, Springer-Verlag, Berlin, Heidelberg, 2005, pp. 447–451–.
[8] M. Gebser, B. Kaufmann, AndréNeumann and T. Schaub, clasp: A Conflict-Driven Answer Set Solver, in: *LPNMR*, Lecture Notes in Computer Science, Vol. 4483, Springer, 2007, pp. 260–265.
[9] M. Gebser, B. Kaufmann and T. Schaub, Conflict-driven answer set solving: From theory to practice, *Artif. Intell.* **187** (2012), 52–89.
[10] M. Nickles, Differentiable SAT/ASP, in: *Proceedings of the 5th International Workshop on Probabilistic Logic Programming, PLP 2018*, 2018, pp. 62–74.
[11] Z. Yang, A. Ishay and J. Lee, NeurASP: Embracing Neural Networks into Answer Set Programming, in: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, C. Bessiere, ed., International Joint Conferences on Artificial Intelligence Organization, 2020, pp. 1755–1762.
[12] A. Skryagin, W. Stammer, D. Ochs, D.S. Dhami and K. Kersting, Neural-Probabilistic Answer Set Programming, in: *Proceedings of the 19th International Conference on Principles of Knowledge Representation and Reasoning*, 2022, pp. 463–473.
[13] Y. Aspis, K. Broda, A. Russo and J. Lobo, Stable and Supported Semantics in Continuous Vector Spaces, in: *Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning, KR 2020, Rhodes, Greece, September 12-18, 2020*, D. Calvanese, E. Erdem and M. Thielscher, eds, 2020, pp. 59–68.
[14] A. Takemura and K. Inoue, Gradient-Based Supported Model Computation in Vector Spaces, in: *Logic Programming and Nonmonotonic Reasoning - 16th International Conference, LPNMR 2022, Genova, Italy, September 5-9, 2022, Proceedings*, G. Gottlob, D. Inclezan and M. Maratea, eds, Lecture Notes in Computer Science, Vol. 13416, Springer, 2022, pp. 336–349.
[15] F. Fages, Consistency of Clark's completion and existence of stable models, *Journal of Methods of Logic in Computer Science* **1** (1994), 51—60.
[16] E. Erdem and V. Lifschitz, Tight Logic Programs, *Theory and Practice of Logic Programming (TPLP)* **3**(4–5) (2003), 499–518.
[17] T. Sato and R. Kojima, Logical inference as cost minimization in vector spaces, in: *Proceedings of the Fourth International Workshop on Declarative Learning Based Programming (DeLBP'19)*, 2019. https://delbp.github.io/.

[18] T. Sato and R. Kojima, in: *Artificial Intelligence IJCAI 2019 International Workshops, Revised Selected Best Papers (Amal El Fallah Seghrouchni,David Sarne (Eds.))*, Lecture Notes in Computer Science, Vol. 12158, Springer, 2020, pp. 239–252. doi:10.1007/978-3-030-56150-5.

[19] K.R. Apt, H.A. Blair and A. Walker, Foundations of Deductive Databases and Logic Programming, J. Minker, ed., 1988, pp. 89–148, Chapter Towards a Theory of Declarative Knowledge.

[20] W. Marek and V.S.Subrahmanian, The relationship between stable, supported, default and autoepistemic semantics for general logic programs, *Theoretical Computer Science* **103**(2) (1992), 365–386.

[21] C. Sakama, K. Inoue and T. Sato, Linear Algebraic Characterization of Logic Programs, in: *Proceedings of the 10th International Conference on Knowledge Science, Engineering and Management (KSEM2017), LNAI 10412, Springer-Verlag*, 2017, pp. 520–533.

[22] T.N. Quoc, K. Inoue and C. Sakama, Enhancing Linear Algebraic Computation of Logic Programs Using Sparse Representation, *New Gen. Comput.* **40**(1) (2022), 225–254–. doi:10.1007/s00354-021-00142-2.

[23] M.K. Sarker, L. Zhou, A. Eberhart and P. Hitzler, Neuro-Symbolic Artificial Intelligence: Current Trends, 2021.

[24] P. Ferraris, J. Lee and V. Lifschitz, A Generalization of the Lin-Zhao Theorem, *Annals of Mathematics and Artificial Intelligence* **47** (2006), 79–101.

[25] R.E. Tarjan, Depth-first search and linear graph algorithms, *SIAM Journal on Computing* **1**(2) (1972), 146–160.

[26] H. Liu and J. Wang, A new way to enumerate cycles in graph, *Advanced Int'l Conference on Telecommunications and Int'l Conference on Internet and Web Applications and Services (AICT-ICIW'06)* (2006), 57–59.

[27] W.F. Dowling and J.H. Gallier, Linear-time Algorithms for Testing the Satisfiability of Propositional Horn Formula, *Journal of Logic Programming* **3** (1984), 267–284.

[28] F. Lin and J. Zhao, On Tight Logic Programs and yet Another Translation from Normal Logic Programs to Propositional Logic, in: *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI'03)*, Morgan Kaufmann Publishers Inc., 2003, pp. 853–858–.

[29] N.-F. Zhou, In Pursuit of an Efficient SAT Encoding for the Hamiltonian Cycle Problem, in: *proceeding of the Principles and Practice of Constraint Programming: 26th International Conference (CP 2020)*, Springer-Verlag, 2020, pp. 585–602.