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.

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. Takemura and Inoue [31] proposed a neurosymbolic learning pipeline that leverages differentiable computation of supported models. Although their method does not specifically address stable model computation, it bypasses the need for a symbolic solver and illustrates how differentiable computation facilitates integration with deep learning. A step towards end-to-end computation was taken by Aspis et al. [13] and Takemura and Inoue [14]. They formulated the computation of supported models, a super class of stable models, entirely as fixedpoint computation in vector spaces, and obtain supported models represented by binary vectors. However, there still remains a gap between computing supported models and computing stable models.

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. Since our

approach is numerical and relies solely on vector and matrix operations, future work could explore the potential benefits of parallel computing technologies such as many-core CPUs and GPUs.

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. Loop formulas 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 [30]. 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. The latter would mean allowing non-stable model computation, and in our continuous approach, we modify the cost function to be affected only by these chosen loop formulas. The intuition behind this heuristic is that the modified cost function would assign higher cost to models that do not satisfy the chosen loop formulas, thus driving the search process away from them.

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. In addition, any propositional normal logic program is acceptable in our framework, since we impose no restrictions on the syntax of programs like the MD condition [13] or the SD condition [14]. More importantly, we incorporate the use of constraints, i.e., rules with an empty head, 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 propositional normal logic 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. Furthermore, the main search algorithm we propose in this paper is incomplete, in the sense that it does not guarantee reaching a global minimum if it exists, nor it cannot conclusively prove that no solution exists.

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 Hamiltonian cycle problem. We there compare performance of precomputation and loop formula heuristics. Section 6 contains related work and Section 7 is the 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 called the *head*, G is a conjunction of literals called the *body* of the rule, respectively. We equate propositional variables with atoms. A literal is an atom (positive literal) or its negation (negative literal). The logical connective $\neg$ in this paper denotes *negation as failure*. We suppose P is written in a given set of atoms $\mathcal{A}$ but usually assume $\mathcal{A} = \mathrm{atom}(P)$, i.e., the set of atoms occurring in P. We use G^+ and G^- to denote the conjunction of positive and negative literals in G, respectively. G may be empty. The empty conjunction is always true. We call $a \leftarrow G$ *rule* for a. A rule with an empty head is called a *constraint*. 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 of* P, $\mathrm{comp}(P)$, is defined as $\mathrm{comp}(P) = \{\mathrm{iff}(a) \mid \text{atom } a \text{ occurs in } P\}$. 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.

An *interpretation* (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 $F = \{F_1, \ldots, F_u\}$

by considering F as a conjunction $F_1 \wedge \cdots \wedge F_u$. 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 a *supported model* of P [19, 20]. When P is a definite program, there is at least one supported model, and its least model is also a supported model. In general, there can be multiple supported models for both definite and non-definite programs P. *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) reduct 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, as in the case of 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. 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].

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.

A walk in a directed graph is a sequence $v_1 \to v_2 \to \cdots \to v_u$ ($u \geqslant 1$) of vertices representing the corresponding non-zero sequence of edges $(v_1, v_2), \ldots, (v_{u-1}, v_u)$. When $v_u = v_1$, it is said to be *closed*. A *cycle* is a closed walk $v_1 \to v_2 \to \cdots \to v_u \to v_1$ where $\{v_1, \ldots, v_u\}$ are all distinct. A *Hamiltonian cycle (HC)* is a cycle which visits every vertex exactly 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, 16][1] when pdg(P) is acyclic, i.e., has no cycle. A *loop* $S = \{a_1, \ldots, a_u\}$ ($u > 0$) in P is a set of atoms where for any pair of atoms a_1 and a_2 in S ($a_1 = a_2$ allowed), there is a path in pdg(P) from a_1 to a_2 and also from a_2 to a_1. A singleton loop $S = \{a\}$ is induced by a self-referencing rule of the form $a \leftarrow a \wedge G$ where G is possibly empty, i.e., a self-loop $a \leftarrow a$. A *support rule* for a relative to a loop S is a rule $a \leftarrow G$ such that $G^+ \cap S = \emptyset$. Given a loop $L = \{a_1, \ldots, a_u\}$ and its external support rules $\{a_1 \leftarrow G_{11}, \ldots, a_1 \leftarrow G_{1n}, \ldots, a_u \leftarrow G_{u1}, \ldots, a_u \leftarrow G_{un}\}$, the *(conjunctive) loop formula* is the following implication: $(a_1 \wedge \cdots \wedge a_u) \to (G_{11} \vee \cdots \vee G_{1n} \vee \cdots \vee G_{u1} \vee \cdots \vee G_{un})$.

We denote vectors by bold 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. An interpretation I over a set $\mathcal{A} = \{a_1, \ldots, a_n\}$ of n ordered atoms is equated with an n-dimensional binary vector $\mathbf{s}_I \in \{0,1\}^n$ such that $\mathbf{s}_I(i) = 1$ if a_i is true in I and $\mathbf{s}_I(i) = 0$ otherwise ($1 \leqslant i \leqslant n$). $\mathbf{s}_I$ is called the vectorized I.

Bold upper case letters such as $\mathbf{A}$ stand for a matrix. We use $\mathbf{A}(i, j)$ to denote the i, j-th element of $\mathbf{A}$, $\mathbf{A}(i, :)$ the i-th row of $\mathbf{A}$ and $\mathbf{A}(:, j)$ the j-th column of $\mathbf{A}$, respectively. We often consider one dimensional matrices as (row or column) vectors. $\|\mathbf{A}\|_F$ denotes the Frobenius norm of $\mathbf{A}$, i.e., $\sqrt{\sum_{i,j} \mathbf{A}(i, j)^2}$. Let $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{m \times n}$ be $m \times n$ matrices. $\mathbf{A} \odot \mathbf{B}$ denotes their Hadamard product, i.e., $(\mathbf{A} \odot \mathbf{B})(i, j) = \mathbf{A}(i, j)\mathbf{B}(i, j)$ for $i, j (1 \leqslant i \leqslant m, 1 \leqslant j \leqslant n)$. $[\mathbf{A}; \mathbf{B}]$ designates the $2m \times n$ matrix of $\mathbf{A}$ 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 $\mathrm{ReLU}(x) = \max(x, 0)$ by $1 - \min_1(x) = \mathrm{ReLU}(1 - x)$. $\min_1(\mathbf{A})$ denotes the result of component-wise application of $\min_1(x)$ to matrix $\mathbf{A}$. 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 $[\mathbf{a} \leqslant \theta](i) = 0$ otherwise. $[\mathbf{a} \geqslant \theta]$ is treated similarly. We extend thresholding to matrices. Thus $[\mathbf{A} \leqslant 1]$ means a

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

matrix such that $[\mathbf{A} \leqslant 1](i,j) = 1$ if $\mathbf{A}(i,j) \leqslant 1$ and $[\mathbf{A} \leqslant 1](i,j) = 0$ otherwise. For convenience, we generalize bit inversion to an n-dimensional vector $\mathbf{a}$ and use an expression $\mathbf{1} - \mathbf{a}$ to denote the n-dimensional vector such that $(\mathbf{1} - \mathbf{a})(i) = 1 - \mathbf{a}(i)$ for $i\,(1 \leqslant i \leqslant n)$. $\mathbf{1} - \mathbf{A}$ 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

Definition 1 (Matricized program)**.** *A program P that has m rules in n atoms is numerically encoded as a pair $\mathbf{P} = (\mathbf{C}, \mathbf{D})$ of binary matrices $\mathbf{C} \in \{0,1\}^{m \times 2n}$ and $\mathbf{D} \in \{0,1\}^{n \times m}$, which we call a matricized program P.*

$\mathbf{C}$ 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 $\mathbf{C}(j,:)$ $(1 \leqslant j \leqslant m)$ 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 $\mathbf{C}(j,:)$ is zero except for $\mathbf{C}(j,i_1) = \cdots = \mathbf{C}(j,i_p) = \mathbf{C}(j,n+i_{p+1}) = \cdots = \mathbf{C}(j,n+i_{p+q}) = 1$. $\mathbf{D}$ 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 $\mathbf{D}(i,j_1) = \cdots = \mathbf{D}(i,j_s) = 1$ 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: $\mathrm{iff}(a_i) = a_i \Leftrightarrow G_{j_1} \vee \cdots \vee G_{j_s}$. If a_i has no rule, we put $\mathbf{D}(i,j) = 0$ for all j $(1 \leqslant j \leqslant m)$. Thus the matricized $\mathbf{P} = (\mathbf{C}, \mathbf{D})$ can represent the completed program $\mathrm{comp}(P)$.

For concreteness, we explain by an example below.

Example 1 (Encoding a program)**.** *Suppose we are given a program P_0 below containing three rules $\{r_1, r_2, r_3\}$ in a set of atoms $\mathcal{A}_0 = \{p, q, r\}$.*

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

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 $\mathbf{P}_0 = (\mathbf{C}_0, \mathbf{D}_0)$. Here $\mathbf{C}_0$ represents conjunctions (the bodies of $\{r_1, r_2, r_3\}$) and $\mathbf{D}_0$ their disjunctions so that they jointly represent P_0.

$$\mathbf{C}_0 = \begin{array}{c} \begin{array}{cccccc} p & q & r & \neg p & \neg q & \neg r \end{array} \\ \left[\begin{array}{cccccc} 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \end{array} \begin{array}{l} : r_1 \textit{ has the body } q \wedge \neg r \\ : r_2 \textit{ has the body } \neg q \\ : r_3 \textit{ has the empty body} \end{array} \tag{2}$$

$$\mathbf{D}_0 = \begin{array}{c} \begin{array}{ccc} r_1 & r_2 & r_3 \end{array} \\ \left[\begin{array}{ccc} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{array}\right] \end{array} \begin{array}{l} : p \textit{ has two rules } r_1 \textit{ and } r_2 \\ : q \textit{ has one rule } r_3 \\ : r \textit{ has no rule} \end{array} \tag{3}$$

As can be seen, $\mathbf{C}_0$ represents conjunctions in P_0 in such a way that $\mathbf{C}_0(1,:)$ for example represents the conjunction $q \wedge \neg r$ of the first rule in P_0 by setting $\mathbf{C}_0(1,2) = \mathbf{C}_0(1,6) = 1$ and so on. $\mathbf{D}_0$ represents disjunctions of rule bodies. So $\mathbf{D}_0(1,1) = \mathbf{D}_0(1,2) = 1$ 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.

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", $I \models F$ 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 $F_1 \vee \cdots \vee F_u$ ($u \geqslant 0$), $I \models F$ iff there is some i ($1 \leqslant i \leqslant u$) s.t. $I \models F_i$. So the empty disjunction ($u = 0$) is always false. We consider a conjunction $F_1 \wedge \cdots \wedge F_u$ as a syntax sugar for $\neg(\neg F_1 \vee \cdots \vee \neg F_u)$ 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 ($a_i \in I$ 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 $\mathbf{s}_I$ such that $\mathbf{s}_I(i) = 1$ if $a_i \in I$ and $\mathbf{s}_I(i) = 0$ ($1 \leqslant i \leqslant n$) otherwise, and introduce the dualized $\mathbf{s}_I$ written as $\mathbf{s}_I^\delta$ by $\mathbf{s}_I^\delta = [\mathbf{s}_I; (\mathbf{1} - \mathbf{s}_I)]$. $\mathbf{s}_I^\delta$ is a vertical concatenation of $\mathbf{s}_I$ and the bit inversion of $\mathbf{s}_I$.

Consider a matricized program $\mathbf{P} = (\mathbf{C}, \mathbf{D})$ ($\mathbf{C} \in \{0,1\}^{m \times 2n}$, $\mathbf{D} \in \{0,1\}^{n \times m}$) and its j-th rule r_j having a body G_j represented by $\mathbf{C}(j,:)$. Compute $\mathbf{C}(j,:)\mathbf{s}_I^\delta$ which is the number of true literals in I in G_j and compare it with the number of literals $|\mathbf{C}(j,:)|_1$[2] in G_j. When $|\mathbf{C}(j,:)|_1 = \mathbf{C}(j,:)\mathbf{s}_I^\delta$ 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 $\mathbf{C}(j,:)(\mathbf{1} - \mathbf{s}_I^\delta)$ 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 $\min_1(\mathbf{C}(j,:)(\mathbf{1} - \mathbf{s}_I^\delta))$ which is 1 if there is a false literal, and 0 otherwise. Consequently $1 - \min_1(\mathbf{C}(j,:)(\mathbf{1} - \mathbf{s}_I^\delta)) = 1$ if there is no false literal in G_j and vice versa. In other words, $1 - \min_1(\mathbf{C}(j,:)(\mathbf{1} - \mathbf{s}_I^\delta))$ 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. $d_i = \sum_{t=1}^{s}(1 - \min_1(\mathbf{C}(j_t,:)(\mathbf{1} - \mathbf{s}_I^\delta)))$ is the number of rule bodies in $\{G_{j_1}, \ldots, G_{j_s}\}$ that are true in I. Noting $\mathbf{D}(i,j) = 1$ if $j \in \{j_1, \ldots, j_s\}$ and $\mathbf{D}(i,j) = 0$ otherwise by construction of $\mathbf{D}$ in $\mathbf{P} = (\mathbf{C}, \mathbf{D})$, we replace the summation $\sum_{t=1}^{s}$ by matrix multiplication and obtain $d_i = \mathbf{D}(i,:)(\mathbf{1} - \min_1(\mathbf{C}(\mathbf{1} - \mathbf{s}_I^\delta)))$. Introduce a column vector $\mathbf{d}_I = \mathbf{D}(\mathbf{1} - \min_1(\mathbf{C}(\mathbf{1} - \mathbf{s}_I^\delta)))$. We have $\mathbf{d}_I(i) = d_i$ = the number of rules for a_i whose bodies are true in I ($1 \leqslant i \leqslant n$).

In the case of $\mathbf{P}_0 = (\mathbf{C}_0, \mathbf{D}_0)$ 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 $\mathbf{s}_{I_0} = [1\ 1\ 0]^T$, $\mathbf{s}_{I_0}^\delta = [1\ 1\ 0\ 0\ 0\ 1]^T$, $\mathbf{1} - \mathbf{s}_{I_0}^\delta = [0\ 0\ 1\ 1\ 1\ 0]^T$ and finally $\mathbf{C}_0(\mathbf{1} - \mathbf{s}_{I_0}^\delta) = [0\ 1\ 0]^T$. The last equation says that the rule bodies of r_1, r_2 and r_3 have respectively zero, one and zero literal false in I_0. Hence $\min_1(\mathbf{C}_0(\mathbf{1} - \mathbf{s}_{I_0}^\delta)) = [0\ 1\ 0]^T$ indicates that only the second rule body is false and the other two bodies are true in I_0. So its bit inversion $\mathbf{1} - \min_1(\mathbf{C}_0(\mathbf{1} - \mathbf{s}_{I_0}^\delta)) = [1\ 0\ 1]^T$ 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 $\mathbf{d}_{I_0} = \mathbf{D}_0(\mathbf{1} - \min_1(\mathbf{C}_0(\mathbf{1} - \mathbf{s}_{I_0}^\delta))) = [1\ 1\ 0]^T$. The elements in $\mathbf{d}_{I_0} = [1\ 1\ 0]^T$ denote 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.

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

$$I \models comp(P) \quad iff \quad \|\mathbf{s}_I - \min_1(\mathbf{d}_I)\|_2 = 0. \tag{4}$$

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

(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 $\mathbf{s}_I(i) = 1$, $I \models a_i$, 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 $\mathbf{s}_I(i) = 0$, 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 $\mathbf{s}_I(i) = d_i$. Since i is arbitrary, we conclude $\mathbf{s}_I = \min_1(\mathbf{d}_I)$, or equivalently $\|\mathbf{s}_I - \min_1(\mathbf{d}_I)\|_2 = 0$. 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 $\mathbf{s}_I - \min_1(\mathbf{d}_I)$ 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 $\mathbf{P}_0 = (\mathbf{C}_0, \mathbf{D}_0)$ with $\mathbf{s}_{I_0} = [1\ \ 1\ \ 0\ ]^T$ as in the aforementioned example, since $\mathbf{s}_{I_0} = \min_1(\mathbf{d}_{I_0})$ 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 $\{r_1, \ldots, r_m\}$ with 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 $\mathbf{P} = (\mathbf{C}, \mathbf{D})$ by a vectorized model $\mathbf{s}_I$. Write $\mathbf{C} \in \{0,1\}^{m \times 2n}$ as $\mathbf{C} = [\mathbf{C}^{pos}\ \mathbf{C}^{neg}]$ where $\mathbf{C}^{pos} \in \{0,1\}^{m \times n}$ (resp. $\mathbf{C}^{neg} \in \{0,1\}^{m \times n}$) is the left half (resp. the right half) of $\mathbf{C}$ representing the positive literals (resp. negative literals) of each rule body in $\mathbf{C}$. Compute $\mathbf{M}^{neg} = \mathbf{1} - \min_1(\mathbf{C}^{neg}\mathbf{s}_I)$. It is an $m \times 1$ matrix (treated as a column vector here) such that $\mathbf{M}^{neg}(j) = 0$ if the body of r_j contains a negative literal false in I and $\mathbf{M}^{neg}(j) = 1$ otherwise $(1 \leqslant j \leqslant m)$. Let r_j^+ be a rule r_j with negative literals in the body deleted. We see that $P^I = \{r_j^+ \mid \mathbf{M}^{neg}(j) = 1, 1 \leqslant j \leqslant m\}$ and P^I is syntactically represented by $(\mathbf{C}^{pos}, \mathbf{D}^I)$ where $\mathbf{D}^I = \mathbf{D}$ with columns $\mathbf{D}(:, j)$ replaced by the zero column vector if $\mathbf{M}^{neg}(j) = 0$ $(1 \leqslant j \leqslant m)$. $\mathbf{D}^I(i,:)$ denotes a rule set $\{r_j^+ \mid \mathbf{D}^I(i,j) = 1, 1 \leqslant j \leqslant m\}$ in P^I for $a_i \in \mathcal{A}$. We call $P^I = (\mathbf{C}^{pos}, \mathbf{D}^I)$ the matricized reduct of P by I.

The matricized reduct $P^I = (\mathbf{C}^{pos}, \mathbf{D}^I)$ is evaluated in vector spaces as follows. Compute $\mathbf{M}^{pos} = \mathbf{M}^{neg} \odot (\mathbf{1} - \min_1(\mathbf{C}^{pos}(\mathbf{1} - \mathbf{s}_I)))$. $\mathbf{M}^{pos}$ denotes the truth values of rule bodies in P^I evaluated by I. Thus $\mathbf{M}^{pos}(j) = 1$ $(1 \leqslant j \leqslant m)$ if r_j^+ is contained in P^I and its body is true in I. Otherwise $\mathbf{M}^{pos}(j) = 0$ and r_j^+ is not contained in P^I or the body of r_j^+ is false in I. Introduce $\mathbf{d}_I^+ = \mathbf{D}\mathbf{M}^{pos}$. $\mathbf{d}_I^+(i)$ $(1 \leqslant i \leqslant n)$ 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 $\mathbf{P} = (\mathbf{C}, \mathbf{D})$ 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 $\mathbf{C} = [\mathbf{C}^{pos}\ \mathbf{C}^{neg}]$ as above. Let $\mathbf{s}_I$ be the vectorized model I. Compute $\mathbf{M}^{neg} = \mathbf{1} - \min_1(\mathbf{C}^{neg}\mathbf{s}_I)$, $\mathbf{M}^{pos} = \mathbf{M}^{neg} \odot (\mathbf{1} - \min_1(\mathbf{C}^{pos}(\mathbf{1} - \mathbf{s}_I)))$ and $\mathbf{d}_I^+ = DM^{pos}$. Also compute $\mathbf{d}_I = \mathbf{D}(\mathbf{1} - \min_1(\mathbf{C}(\mathbf{1} - \mathbf{s}_I^\delta)))$. Then, $I \models \mathrm{comp}(P)$, $\|\mathbf{s}_I - \min_1(\mathbf{d}_I)\|_2 = 0$, $\|\mathbf{s}_I - \min_1(\mathbf{d}_I^+)\|_2 = 0$ and $I \models \mathrm{comp}(P^I)$ are all equivalent. (Proof in Appendix A.1.)*

From the viewpoint of end-to-end ASP, Proposition 2 means that we can obtain a supported model I as a binary solution $\mathbf{s}_I$ of the equation $\mathbf{s}_I - \min_1(\mathbf{d}_I) = 0$ derived from P or $\mathbf{s}_I - \min_1(\mathbf{d}_I^+) = 0$ 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 $\mathbf{s}_I - \min_1(\mathbf{d}_I)$ is piecewise linear w.r.t. $\mathbf{s}_I$ whereas the latter $\mathbf{s}_I - \min_1(\mathbf{d}_I^+)$ is piecewise quadratic w.r.t. $\mathbf{s}_I$.

Example 2 (Evaluation of a reduct)**.** *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 $\mathbf{P}_0 = (\mathbf{C}_0, \mathbf{D}_0)$ be the matricized P_0. We first decompose $\mathbf{C}_0$ in (2) as $\mathbf{C}_0 = [\mathbf{C}_0^{pos}\ \mathbf{C}_0^{neg}]$ where $\mathbf{C}_0^{pos}$ is the positive part and $\mathbf{C}_0^{neg}$ the negative part of $\mathbf{C}_0$. They are*

$$\mathbf{C}_0^{pos} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \textit{and} \quad \mathbf{C}_0^{neg} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

Let $\mathbf{s}_{I_0} = [1\ \ 1\ \ 0]^T$ be the vectorized I_0. We first compute $\mathbf{M}_0^{neg} = \mathbf{1} - min_1(\mathbf{C}_0^{neg}\mathbf{s}_{I_0})$ to determine rules to be removed. Since $\mathbf{M}_0^{neg} = [1\ \ 0\ \ 1]^T$, the second rule r_2, indicated by $\mathbf{M}_0^{neg}(2) = 0$, is removed from P_0, giving $P_0^{I_0} =$

$\{r_1^+, r_3^+\}$. *Using* $\mathbf{M}_0^{neg}$ *and* $\mathbf{D}_0$ *shown in (3), we then compute* $\mathbf{M}_0^{pos} = \mathbf{M}_0^{neg} \odot (\mathbf{1} - min_1(\mathbf{C}_0^{pos}(\mathbf{1} - \mathbf{s}_{I_0}))) = [1\ 0\ 1]^T$ *and* $\mathbf{d}_{I_0}^+ = \mathbf{D}_0\mathbf{M}_0^{pos} = [1\ 1\ 0]^T$. *The elements of* $\mathbf{d}_{I_0}^+$ *denote the number of rule bodies in* $P_0^{I_0}$ *that are true in* I_0 *for each atom. Thus, since* $\mathbf{s}_{I_0} = min_1(\mathbf{d}_{I_0}^+)(= [1\ 1\ 0]^T)$ *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 and Inoue 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 $\mathbf{s}_I$ representing a supported model I of a matricized program $\mathbf{P} = (\mathbf{C}, \mathbf{D})$ that satisfies $\|\mathbf{s}_I - \min_1(\mathbf{d}_I)\|_2 = 0$ where $\mathbf{d}_I = \mathbf{D}(\mathbf{1} - \min_1(\mathbf{C}(\mathbf{1} - \mathbf{s}_I^\delta)))$. For this task, we relax $\mathbf{s}_I \in \{0,1\}^n$ to $\mathbf{s} \in \mathbb{R}^n$ and introduce a non-negative cost function L_{SU}:

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

Proposition 3. *Let* L_{SU} *be defined from a program* $\mathbf{P} = (\mathbf{C}, \mathbf{D})$ *as above.*
$L_{SU} = 0$ *iff* $\mathbf{s}$ *is a binary vector representing a supported model of P.*

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

L_{SU} is piecewise differentiable and we can obtain a supported model of P as a root $\mathbf{s}$ of L_{SU} by minimizing L_{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 $\mathbf{s} = [u_1, \ldots, u_n]^T$ represents their continuous truth values where $\mathbf{s}(p) = s_p \in \mathbb{R}$ is the continuous truth value for atom a_p ($1 \leqslant p \leqslant n$). For the convenience of derivation, we introduce the dot product $(\mathbf{A} \bullet \mathbf{B}) = \sum_{i,j} \mathbf{A}(i,j)\mathbf{B}(i,j)$ of matrices $\mathbf{A}$ and $\mathbf{B}$ 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 $(\mathbf{A} \bullet (\mathbf{B} \odot \mathbf{C})) = ((\mathbf{B} \odot \mathbf{A}) \bullet \mathbf{C})$ and $(\mathbf{A} \bullet (\mathbf{BC})) = ((\mathbf{B}^T\mathbf{A}) \bullet \mathbf{C}) = ((\mathbf{AC}^T) \bullet \mathbf{B})$ hold (see Appendix B.1 for details).

Let $\mathbf{P} = (\mathbf{C}, \mathbf{D})$ be the matricized program and write $\mathbf{C} = [\mathbf{C}^{pos}\, \mathbf{C}^{neg}]$. Introduce $\mathbf{N}$, $\mathbf{M}$, $\mathbf{d}$, $\mathbf{E}$, $\mathbf{F}$ and compute L_{SU} by

$$\begin{array}{lcll}
\mathbf{N} & = & \mathbf{C}(\mathbf{1} - \mathbf{s}^\delta) = \mathbf{C}^{pos}(\mathbf{1} - \mathbf{s}) + \mathbf{C}^{neg}\mathbf{s} & \text{: (continuous) counts of false literals in the rule bodies} \\
\mathbf{M} & = & \mathbf{1} - \min_1(\mathbf{N}) & \text{: (continuous) truth values of the rule bodies} \\
\mathbf{d} & = & \mathbf{DM} & \text{: (continuous) counts of true disjuncts for each atom} \\
\mathbf{E} & = & \min_1(\mathbf{d}) - \mathbf{s} & \text{: error between the estimated truth values of atoms and } \mathbf{s} \\
\mathbf{F} & = & \mathbf{s} \odot (\mathbf{1} - \mathbf{s}) & \text{: (continuous) 0 iif } \mathbf{s} \text{ is binary} \\
L_{sq} & = & (\mathbf{E} \bullet \mathbf{E}) & \\
L_{nrm} & = & (\mathbf{F} \bullet \mathbf{F}) & \\
L_{SU} & = & 0.5 \cdot (L_{sq} + \ell_2 \cdot L_{nrm}). &
\end{array} \tag{6}$$

We then compute the Jacobian $J_{a_{SU}}$ of L_{SU} as follows (full derivation in Appendix B.2):

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

Note that since $\mathbf{s}$ is a vector, the Jacobian in this case is also a vector.

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 $\widehat{\mathbf{C}} = [\widehat{\mathbf{C}}^{pos}\ \widehat{\mathbf{C}}^{neg}]$. We call $\widehat{\mathbf{C}}$ *constraint matrix*. We introduce $\mathbf{N}_{\widehat{\mathbf{c}}}$, a non-negative function $L_{\widehat{\mathbf{c}}}$ of $\mathbf{s}$ and $L_{\widehat{\mathbf{c}}}$'s Jacobian $J_{a_{\widehat{\mathbf{c}}}}$ as follows (derivation in Appendix B.3):

$$\begin{aligned} &\mathbf{N}_{\widehat{\mathbf{c}}} = \widehat{\mathbf{C}}(\mathbf{1} - \mathbf{s}^\delta) = \widehat{\mathbf{C}}^{pos}(\mathbf{1} - \mathbf{s}) + \widehat{\mathbf{C}}^{neg}\mathbf{s} && : \text{number of literals falsified by } \mathbf{s} \\ &L_{\widehat{\mathbf{c}}} = (\mathbf{1} \bullet (\mathbf{1} - \min_1(\mathbf{N}_{\widehat{\mathbf{c}}}))) \ \text{ where } \mathbf{1} \text{ is an all-ones vector} && : \text{counts of violated constraints} \qquad (8) \\ &J_{a_{\widehat{\mathbf{c}}}} = (\widehat{\mathbf{C}}^{pos} - \widehat{\mathbf{C}}^{neg})^T[\mathbf{N}_{\widehat{\mathbf{c}}} \leqslant 1] && : \text{the Jacobian of } L_{\widehat{\mathbf{c}}} \qquad (9) \end{aligned}$$

The meaning of $\mathbf{N}_{\widehat{\mathbf{c}}}$ and $L_{\widehat{\mathbf{c}}}$ is clear when $\mathbf{s}$ is binary. Note that any binary $\mathbf{s}$ 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 $\widehat{\mathbf{C}}$ is a $k \times 2n$ binary matrix and $\mathbf{N}_{\widehat{\mathbf{c}}}$ is a $k \times 1$ matrix. $\mathbf{N}_{\widehat{\mathbf{c}}}(i)$ $(1 \leqslant i \leqslant k)$ is the number of literals falsified by $\mathbf{s}$ in a conjunction G_i of the i-th constraint $\leftarrow G_i$. So $\mathbf{N}_{\widehat{\mathbf{c}}}(i) = 0$, or equivalently $1 - \min_1(\mathbf{N}_{\widehat{\mathbf{c}}}(i)) = 1$ implies G_i has no false literal i.e., $\mathbf{s} \models G_i$, and vice versa. Hence $L_{\widehat{\mathbf{c}}} = \sum_{i=1}^{k}(1 - \min_1(\mathbf{N}_{\widehat{\mathbf{c}}}(i))) = (\mathbf{1} \bullet (\mathbf{1} - \min_1(\mathbf{N}_{\widehat{\mathbf{c}}})))$ equals the number of violated constraints. Consequently when $\mathbf{s}$ is binary, we can say that $L_{\widehat{\mathbf{c}}} = 0$ iff all constraints are satisfied by $\mathbf{s}$.

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

3.5. An algorithm for computing supported models with constraints

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

$$\begin{aligned} L_{SU+\widehat{\mathbf{c}}} &= L_{SU} + \ell_3 \cdot L_{\widehat{\mathbf{c}}} \qquad (10) \\ &= 0.5 \cdot \left(\|\mathbf{s} - \min_1(\mathbf{d})\|_2^2 + \ell_2 \cdot \|\mathbf{s} \odot (\mathbf{1} - \mathbf{s})\|_2^2\right) + \ell_3 \cdot (\mathbf{1} \bullet (\mathbf{1} - \min_1(\widehat{\mathbf{C}}(1 - \mathbf{s}^\delta)))) \quad \ell_2 > 0,\ \ell_3 > 0 \\ &\quad \text{where } \mathbf{d} = \mathbf{D}(\mathbf{1} - \min_1(\mathbf{C}(\mathbf{1} - \mathbf{s}^\delta))) \\ J_{a_{SU+\widehat{\mathbf{c}}}} &= J_{a_{SU}} + \ell_3 \cdot J_{a_{\widehat{\mathbf{c}}}} \qquad (11) \end{aligned}$$

The next proposition is immediate from Proposition 3.

Proposition 4. $L_{SU+\widehat{\mathbf{c}}} = 0$ *iff* $\mathbf{s}$ *represents a supported model of P satisfying a constraint matrix* $\widehat{\mathbf{C}}$.

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

$$\mathbf{s}_{\rm new}=\mathbf{s}-\alpha\left(\frac{L_{SU+\widehat{\mathbf{c}}}}{(J_{a_{SU+\widehat{\mathbf{c}}}}\bullet J_{a_{SU+\widehat{\mathbf{c}}}})}\right)J_{a_{SU+\widehat{\mathbf{c}}}} \tag{12}$$

Algorithm 1: minimizing $L_{SU+\widehat{\mathbf{c}}}$ to zero

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

Algorithm 1 is a double loop algorithm where the inner j-loop updates $\mathbf{s}\in\mathbb{R}^n$ repeatedly to minimize $L_{SU+\widehat{\mathbf{c}}}$ while thresholding $\mathbf{s}$ into a binary solution candidate $\mathbf{s}_*\in\{0,1\}^n$ for $L_{SU+\widehat{\mathbf{c}}}=0$. 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 $\mathbf{s}(i)\sim\mathcal{N}(0,1)+0.5$ $(1\leqslant i\leqslant n)$ where $\mathcal{N}(0,1)$ is the standard normal distribution. Lines 6,7 and 8 collectively perform thresholding of $\mathbf{s}$ into a binary $\mathbf{s}_*$. As the inner loop repeats, $L_{SU+\widehat{\mathbf{c}}}$ becomes smaller and smaller and so do L_{sq} and L_{nrm} in L_{SU}. L_{sq} being small means $\mathbf{s}$ is close to a supported model of P while L_{nrm} being small means each element of $\mathbf{s}$ is close to $\{0,1\}$. So binarization $\mathbf{s}_*=[\mathbf{s}\geqslant\theta]$ with an appropriate threshold θ[3] has a good chance of yielding a binary $\mathbf{s}_*$ representing a supported model of P satisfying constraints represented by $\widehat{\mathbf{C}}$. It may happen that the inner loop fails to find a solution. In such a case, we retry another j-loop with perturbated $\mathbf{s}$ at line 12. There $\mathbf{s}$ is perturbated by $\mathbf{s}\leftarrow 0.5(\mathbf{s}+\Delta+0.5)$ where $\Delta\sim\mathcal{N}(0,1)$ before the next j-loop.

4. Computing stable models in vector spaces

4.1. Loop formulas and stable models

Let $\mathbf{P}=(\mathbf{C},\mathbf{D})$ 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 $\mathbf{C}\in\{0,1\}^{m\times 2n}$ and $\mathbf{D}\in\{0,1\}^{n\times m}$. We assume atoms and rules are ordered as indicated.

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

Computing a supported model of P is equivalent to computing any binary fixedpoint $\mathbf{s} \in \{0,1\}^n$ such that $\mathbf{s} = \min_1(\mathbf{D}(\mathbf{1} - \min_1(\mathbf{C}(\mathbf{1} - \mathbf{s}^\delta))))$ 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 $S = \{h_1, \ldots, h_p\} \subseteq \mathcal{A}$ be a loop in P. Recall that S is a set of atoms which are strongly connected in the positive dependency graph of P[4]. A support rule for h with respect to S is a rule $h \leftarrow H$ such that $H^+ \cap S = \emptyset$. H is called a support body for S. Introduce a (conjunctive) loop formula for S by

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

Then define *loop formulas associated with* P as $LF(P) = \{LF(S) \mid S \text{ is a loop in } P\}$, which is treated as the conjunction of its elements. We note that in the original form [1], the antecedent of $LF(S)$ is a disjunction $(h_1 \vee \cdots \vee h_p)$. Later it was shown that the disjunctive and conjunctive loop formulas are equivalent [24], and we choose to use the conjunctive form of $LF(S)$ as it is easier to satisfy using our method.

We evaluate $LF(P)$ by a real vector $\mathbf{s} \in \mathbb{R}^n$. Introduce an external support matrix $\mathbf{E}_{sup} \in \{0,1\}^{n \times m}$ by $\mathbf{E}_{sup}(i,j) = 1$ if there is a support rule $a_i \leftarrow G_j$ for $a_i \in \mathcal{A}$, else $\mathbf{E}_{sup}(i,j) = 0$ $(1 \leqslant i \leqslant n, 1 \leqslant j \leqslant m)$. Suppose there are w loops $\{S_1, \ldots, S_w\}$ in P. Introduce a loop matrix $\mathbf{L}_{oop} \in \{0,1\}^{w \times m}$ such that $\mathbf{L}_{oop}(v,j) = 1$ if the v-th loop S_v has G_j as a support body for S_v, else $\mathbf{L}_{oop}(v,j) = 0$ $(1 \leqslant v \leqslant w)$.

Example 3 (Encoding loop formulas)**.** *Suppose we are given a program* P_{L0}*:*

$$P_{L0} = \begin{cases} p \leftarrow q \wedge \neg r & : \text{rule } r_1 \text{ for } p \\ p \leftarrow \neg s & : \text{rule } r_2 \text{ for } p \\ q \leftarrow p & : \text{rule } r_3 \text{ for } q \\ r \leftarrow r & : \text{rule } r_4 \text{ for } r \end{cases} \tag{14}$$

This program contains two loops: $S_1 = \{p,q\}$ *and* $S_2 = \{r\}$*. In this case, only* S_1 *has an external support body* $\neg s$*. Thus, the external support matrix* $\mathbf{E}_{sup0}$ *and the loop matrix* $\mathbf{L}_{oop0}$ *for this program are as follows:*

$$\mathbf{E}_{sup0} = \begin{array}{c} \begin{matrix} r_1 & r_2 & r_3 & r_4 \end{matrix} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \end{array} \begin{array}{l} : p \text{ has the body of } r_2 \text{ as the support body} \\ : q \text{ has no external support} \\ : r \text{ has no external support} \\ : s \text{ is not part of any loops} \end{array} \tag{15}$$

$$\mathbf{L}_{oop0} = \begin{array}{c} \begin{matrix} r_1 & r_2 & r_3 \end{matrix} \\ \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \end{array} \begin{array}{l} : S_1 \text{ has the body of } r_2 \text{ as its support body} \\ : S_2 \text{ has no support bodies} \end{array} \tag{16}$$

We then introduce a loss function L_{LF}, which is a non-negative piecewise linear function of $\mathbf{s}$.

$$\begin{aligned} \mathbf{M} &= \mathbf{1} - \min_1(\mathbf{C}(\mathbf{1} - \mathbf{s}^\delta)) && : \text{(continuous) truth values by } \mathbf{s} \text{ of the rule bodies in } P \\ \mathbf{S}_v &= \mathbf{L}_{oop}(v,:) && : \text{represents the } v\text{-th loop in } \{\mathbf{S}_1, \ldots, \mathbf{S}_t\} \\ A_v &= \mathbf{S}_v(\mathbf{1} - \mathbf{s}) + \mathbf{S}_v \mathbf{E}_{sup} \mathbf{M} && : \text{(continuous) counts of true disjuncts by } \mathbf{s} \text{ of } LF(S_v) \\ L_{LF} &= \sum_{v=1}^{w} (1 - \min_1(A_v)) \end{aligned} \tag{17}$$

[4] In the case of a singleton loop $S = \{h\}$, 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.

Proposition 5. *Let L_{LF} be defined as above. When* **s** *is a binary vector representing a model I over $\mathcal{A}$, it holds that $L_{LF} = 0$ iff* $\mathbf{s} \models LF(P)$.

(Proof) Suppose $L_{LF} = 0$ and **s** is binary. A summand $(1 - \min_1(A_v))$ in L_{LF} (17) corresponds to the v-th loop $S_v = \{h_1, \ldots, h_p\}$ and is non-negative. Consider $LF(S_v) = (h_1 \wedge \cdots \wedge h_p) \rightarrow (H_1 \vee \cdots \vee H_q)$ as a disjunction $\neg h_1 \vee \cdots \vee \neg h_p \vee H_1 \vee \cdots \vee H_q$. Then $L_{LF} = 0$ implies $(1 - \min_1(A_v)) = 0$, or equivalently $A_v \geqslant 1$. Consequently, as **s** is binary, we have $\mathbf{S}_v(\mathbf{1} - \mathbf{s}) \geqslant 1$ or $\mathbf{S}_v\mathbf{E}_{sup}\mathbf{M} \geqslant 1$. The former means $I \models \neg h_1 \vee \cdots \vee \neg h_p$. The latter, $\mathbf{S}_v\mathbf{E}_{sup}\mathbf{M} \geqslant 1$, means $I \models H_1 \vee \cdots \vee H_q$. This is because the element $(\mathbf{E}_{sup}\mathbf{M})(i)$ is the number of support rules for $a_i \in \mathcal{A}$ whose bodies are true in I **s** ($1 \leqslant i \leqslant n$), and hence $\mathbf{S}_v\mathbf{E}_{sup}\mathbf{M} \geqslant 1$ means some support body H_r ($1 \leqslant r \leqslant q$) for S_v is true in I. So in either case $I \models LF(S_v)$. Since v is arbitrary, we have $I \models LF(P)$. The converse is straightforward and omitted. Q.E.D.

The Jacobian $J_{a_{LF}}$ of L_{LF} is computed as follows (derivation in Appendix B.4):

$$
\begin{aligned}
\mathbf{N} &= \mathbf{C}(\mathbf{1} - \mathbf{s}^{\delta}) \\
N_v &= \mathbf{S}_v(\mathbf{1} - \mathbf{s}) \\
M_v &= \min_1(N_v) \\
J_{a_{LF}} &= \frac{\partial L_{LF}}{\partial \mathbf{s}} = \sum_{v=1}^{w} - \left(\frac{\partial \min_1(A_v)}{\partial \mathbf{s}} \right) \\
&= \sum_{v=1}^{w} [A_v \leqslant 1] \left([N_v \leqslant 1] \mathbf{S}_v^T + (((\mathbf{S}_v\mathbf{E}_{sup}) \odot [\mathbf{N} \leqslant 1]^T)(\mathbf{C}^{neg} - \mathbf{C}^{pos})^T \right) \qquad (18)
\end{aligned}
$$

Here $\mathbf{C} = [\mathbf{C}^{pos}\, \mathbf{C}^{neg}]$ and $\mathbf{S}_v$, A_v and $\mathbf{M}$ are computed by (17).

Now introduce a new cost function $L_{SU+\widehat{\mathbf{c}}+LF}$ by (19) that incorporates L_{LF} and compute its Jacobian $J_{a_{SU+\widehat{\mathbf{c}}+LF}}$ by (11).

$$
\begin{aligned}
L_{SU+\widehat{\mathbf{c}}+LF} &= L_{SU+\widehat{\mathbf{c}}} + \ell_4 \cdot L_{LF} \ \text{ where } \ \ell_4 > 0 \qquad (19) \\
J_{a_{SU+\widehat{\mathbf{c}}+LF}} &= J_{a_{SU+\widehat{\mathbf{c}}}} + \ell_4 \cdot J_{a_{LF}} \qquad (20)
\end{aligned}
$$

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

Proposition 6. **s** *is a stable model of P satisfying constraints represented by $\widehat{\mathbf{C}}$ iff* **s** *is a root of $L_{SU+\widehat{\mathbf{c}}+LF}$.*

We compute such **s** by Newton's method using Algorithm 1 with a modified update rule (12) such that $L_{SU+\widehat{\mathbf{c}}}$ and $J_{a_{SU+\widehat{\mathbf{c}}}}$ are replaced by $L_{SU+\widehat{\mathbf{c}}+LF}$ and $J_{a_{SU+\widehat{\mathbf{c}}+LF}}$ 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 a case, we directly minimize $L_{SU+\widehat{\mathbf{c}}}$ instead of using $L_{SU+\widehat{\mathbf{c}}+LF}$ with the empty LF.

4.2. LF heuristics

Minimizing $L_{SU+\widehat{\mathbf{c}}+LF}$ 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 L_{LF} (17) 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 heuristic is LF_{max}. 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 heuristic, 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 $\mathrm{pdg}(P)$. 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 $L_{SU+\widehat{\mathbf{c}}+LF}$.

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 $\mathcal{A} = \mathrm{atom}(P)$ 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][5]. 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 K, we can obtain downsized ones, P' and K', as follows.

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

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

$$P' = \{a \leftarrow G' \mid a \leftarrow G \in P, a \notin \mathcal{F}_P, G^+ \cap \mathcal{F}_P = \emptyset\} \text{ where } G^+ = \text{positive literals in } G \tag{21}$$

$$K' = \{\leftarrow G' \mid \leftarrow G \in K, G^+ \cap \mathcal{F}_P = \emptyset\} \tag{22}$$

Proposition 7. *Let P' and K' be respectively the program (21) and constraints (22). 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 K' iff I is a stable model of P satisfying constraints K.

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

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

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 $G' = G \setminus \{\neg b \mid b \in \mathcal{F}_P\}$ 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 SLD 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'.

As for the constraints, consider a constraint $\leftarrow G'$ in K'. We consider two cases, where $b \in \mathcal{F}_P$ occurs positively and negatively in G and G'. In the former case, the body remains the same between G' and G, thus if $I' \models G'$ then $I \models G$ and vice versa. In the latter case, because $\neg b$ always evaluates to true, the negative occurrence $\neg b$ in the body of the constraint will not change the result of the conjunction G. Thus, combining the two cases and since the constraint $\leftarrow G'$ is arbitrary, we conclude that $I' \models K'$ iff $I \models K$. 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[6].

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

[6]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.

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.

$$P_1 = \begin{cases} a_1 \leftarrow \neg a_2 \wedge \neg a_3,\ a_2 \leftarrow \neg a_3 \wedge \neg a_1,\ a_3 \leftarrow \neg a_1 \wedge \neg a_2 \\ b_1 \leftarrow \neg b_2 \wedge \neg b_3,\ b_2 \leftarrow \neg b_3 \wedge \neg b_1,\ b_3 \leftarrow \neg b_1 \wedge \neg b_2 \\ c_1 \leftarrow \neg c_2 \wedge \neg c_3,\ c_2 \leftarrow \neg c_3 \wedge \neg c_1,\ c_3 \leftarrow \neg c_1 \wedge \neg c_2 \\ d_1 \leftarrow \neg d_2 \wedge \neg c_3,\ d_2 \leftarrow \neg d_3 \wedge \neg d_1,\ d_3 \leftarrow \neg d_1 \wedge \neg d_2 \end{cases} \tag{23}$$

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

$$K_1 = \begin{cases} \leftarrow a_1 \wedge b_1,\ \leftarrow a_2 \wedge b_2,\ \leftarrow a_3 \wedge b_3 \\ \leftarrow a_1 \wedge c_1,\ \leftarrow a_2 \wedge c_2,\ \leftarrow a_3 \wedge c_3 \\ \leftarrow b_1 \wedge c_1,\ \leftarrow b_2 \wedge c_2,\ \leftarrow b_3 \wedge c_3 \\ \leftarrow b_1 \wedge d_1,\ \leftarrow b_2 \wedge d_2,\ \leftarrow b_3 \wedge d_3 \\ \leftarrow d_1 \wedge c_1,\ \leftarrow d_2 \wedge c_2,\ \leftarrow d_3 \wedge c_3 \end{cases} \tag{24}$$

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 (23) is matricized to $\mathbf{P}_1 = (\mathbf{C}_1, \mathbf{D}_1)$ where $\mathbf{D}_1$ is a (12×12) binary identity matrix (because there are 12 atoms and each atom has just one rule) and $\mathbf{C}_1$ is a (12×24) binary matrix shown in (25). Constraints listed in (24) are a matricized to a (15×12) constraint matrix $\widehat{\mathbf{C}}_{K_1}$ (26). In (25) and (26), a for example stands for a triple $(a_1\, a_2\, a_3)$ and $\neg a$ for $(\neg a_1\, \neg a_2\, \neg a_3)$.

$$\begin{array}{c} \begin{array}{cccccccc} a & b & c & d & \neg a & \neg b & \neg c & \neg d \end{array} \\ \mathbf{C}_1 = \begin{bmatrix} & & & & \mathbf{H}_3 & & & \\ & & & & & \mathbf{H}_3 & & \\ & & & & & & \mathbf{H}_3 & \\ & & & & & & & \mathbf{H}_3 \end{bmatrix} \end{array} \text{ where } \mathbf{H}_3 = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \tag{25}$$

$$\begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \widehat{\mathbf{C}}_{K_1} = \begin{bmatrix} \mathbf{E}_3 & \mathbf{E}_3 & & \\ \mathbf{E}_3 & & \mathbf{E}_3 & \\ & \mathbf{E}_3 & \mathbf{E}_3 & \\ & \mathbf{E}_3 & & \mathbf{E}_3 \\ & & \mathbf{E}_3 & \mathbf{E}_3 \end{bmatrix} \end{array} \text{ where } \mathbf{E}_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{26}$$

We run Algorithm 1 on program P_1 with constraints K_1 to find a supported model (solution) of P_1 satisfying K_1[7].

Table 1
Time and the number of solutions

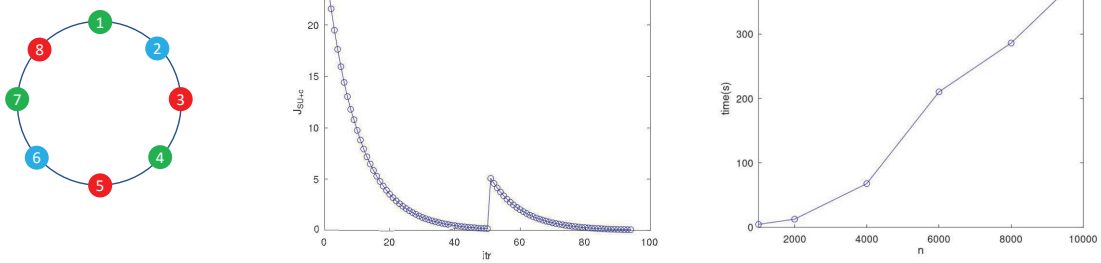
time(s)	#solutions
6.7(0.7)	5.2(0.9)

To measure time to find a model, we conduct ten trials[8] 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)[9] on average. Also to check the ability of finding

[7] Since P_1 is a tight program, every supported model of P_1 is a stable model and vice versa.
[8] One trial consists of $max\_itr \times max\_try$ parameter updates.
[9] The numbers in the parentheses indicate the standard deviation.

different solutions, we perform ten trials of Algorithm 1[10]and count the number of different solutions in the returned solutions. #solutions in Table 1 is the average of ten such measurements. Considering there are six solutions and the naive implementation, the number of different solutions found by the algorithm, which was 5.2 on average, seems rather high.

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 $\mathbf{P}_2 = (\mathbf{C}_2, \mathbf{D}_2)$ and a constraint matrix $\widehat{\mathbf{C}}_{K_2}$ where $\mathbf{D}_2(3n \times 3n)$ is an identity matrix and $\mathbf{C}(3n \times 6n)$ and $\widehat{\mathbf{C}}_{K_2}(3n \times 6n)$ represent respectively rules and constraints. There are $2^n + 2(-1)^n$ solutions ($n \geqslant 3$) in 2^{3n} possible assignments for $3n$ atoms[11]. So the problem will be exponentially difficult as n goes up.



(a) A cycle graph (b) Minimization of $L_{SU+\widehat{\mathbf{c}}}$ with retry (c) Scalability

Fig. 2. Convergence and scalability

The graph (b) in Fig. 2 is an example of convergence curve of $L_{SU+\widehat{\mathbf{c}}}$ 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 $\mathbf{s}$. Then perturbation is given to $\mathbf{s}$ which causes a small jump of $L_{SU+\widehat{\mathbf{c}}}$ at $itr = 51$ and the second cycle of j-loop starts and this time a solution is found after dozens of updates by thresholding $\mathbf{s}$ to a binary vector $\mathbf{s}_*$.

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[12].

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. $U_{j,q}$

[10]without *another solution constraint* introduced in Section 5.2

[11]Based on the *chromatic polynomial* for cycle graphs (for an introduction, see for example [32]), which is given by $P(C_n, \lambda) = (\lambda - 1)^n + (-1)^n(\lambda - 1)$ for a cycle graph C_n with n vertices ($n \geqslant 3$) colored by λ colors.

[12]Actually, [29] 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.

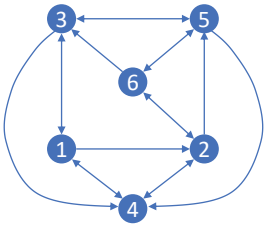
	conditions		meaning
(1)	one_of$(H_{i,j_1},\ldots,H_{i,j_k})$	:	one of outgoing edges $\{i\to j_1,\ldots,i\to j_k\}$ from vertex i is in an HC
(2)	$U_{j,q}\leftarrow H_{i,j}\wedge U_{i,q-1}$	:	if edge $i\to j$ is in an HC and vertex i is visited at time $q-1$, vertex j is visited at time q $(1\leqslant i,j,q\leqslant N)$
(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 of a Hamiltonian cycle problem.

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 $K_3=\{(4),(5),(6)\}$. To be more precise, the first condition (1) is translated into a tight program just like a program P_1 (23). The conditions $\{(2),(3)\}$ constitute a tight definite program. Constraints $K_2=\{(4),(5),(6)\}$ 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_3 satisfying K_3 gives an HC.

We apply the above encoding to a simple Hamiltonian cycle problem for a graph G_2 in Fig. 4[13]. There are six vertices and six HCs[14]. To solve this HC problem, we matricize P_3 and K_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 $(\mathbf{C}_3,\mathbf{D}_3)$ where $\mathbf{D}_3$ is a 72×197 binary matrix for disjunctions[15]and $\mathbf{C}_3$ is a 197×144 matrix for conjunctions (rule bodies). Likewise $K_3=\{(4),(5),(6)\}$ is translated into a constraint matrix $\widehat{\mathbf{C}}_{K_3}$ which is a 67×144 binary matrix. A more detailed description of the encoding is available in the appendix (Appendix C). Then our task is to find a root $\mathbf{s}$ of $L_{SU+\widehat{\mathbf{c}}}$ (10) constructed from these $\mathbf{C}_3$, $\mathbf{D}_3$ and $\widehat{\mathbf{C}}_{K_3}$ in a 72 dimensional vector space by minimizing $L_{SU+\widehat{\mathbf{c}}}$ to zero.

We apply precomputation in the previous section to $\mathbf{P}_3=(\mathbf{C}_3,\mathbf{D}_3)$ and $\widehat{\mathbf{C}}_{K_3}$ to reduce program size. It takes 2.3ms and detects 32 false stable atoms. It outputs a precomputed program $\mathbf{P}'_3=(\mathbf{C}'_3,\mathbf{D}'_3)$ and a constraint matrix $\widehat{\mathbf{C}}'_{K_3}$ of size $\mathbf{D}'_3(40\times 90)$, $\mathbf{C}'_3(90\times 80)$ and $\widehat{\mathbf{C}}'_{K_3}(52\times 80)$ 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 $\mathbf{P}_3=(\mathbf{C}_3,\mathbf{D}_3)$ with $\widehat{\mathbf{C}}_{K_3}$ (no precomputation) and also on $\mathbf{P}'_3=(\mathbf{C}'_3,\mathbf{D}'_3)$ with $\widehat{\mathbf{C}}'_{K_3}$ (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

[13] G_2 is taken from: Section 6.2 in *Potassco User Guide* (`https://github.com/potassco/guide/releases/tag/v2.2.0`).

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

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

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[16], 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 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.

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

We here consider an even n, then 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.

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

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 $\mathbf{P}_{4\_n} = (\mathbf{C}_{4\_n}, \mathbf{D}_{4\_n})$ where $\mathbf{C}_{4\_n}$ is a $(2n+3) \times (2n+4)$ binary matrix and $\mathbf{D}_{4\_n}$ 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 $L_{SU+\widehat{\mathbf{c}}+LF}$ (19) with coefficient $\ell_3 = 0$ for the constraint term (because of no use of constraints) using Jacobian $J_{a_{SU+\widehat{\mathbf{c}}+LF}}$ (11).

[16] This includes the time for precomputation.

Below is an example of the program P_{4_n} where $n = 4$.

$$P_{4\_4} = \begin{cases} a_0 \leftarrow a_1 \wedge a_2 \wedge a_3 \wedge a_4 & \\ a_0 \leftarrow \neg a_5 & \\ a_1 \leftarrow a_0 & a_1 \leftarrow a_2 \\ a_2 \leftarrow a_0 & a_2 \leftarrow a_1 \\ a_3 \leftarrow a_0 & a_3 \leftarrow a_4 \\ a_4 \leftarrow a_0 & a_4 \leftarrow a_3 \\ a_5 \leftarrow a_5 & \end{cases}$$

This program has three minimal loops $\{a_1,a_2\},\{a_3,a_4\},a_5$ and a maximal loop $\{a_0,a_1,a_2,a_3,a_4\}$. There are 11 rules and six atoms, so $\mathbf{C}_{4\_4}$ is a (11×12) binary matrix.

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 a 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.

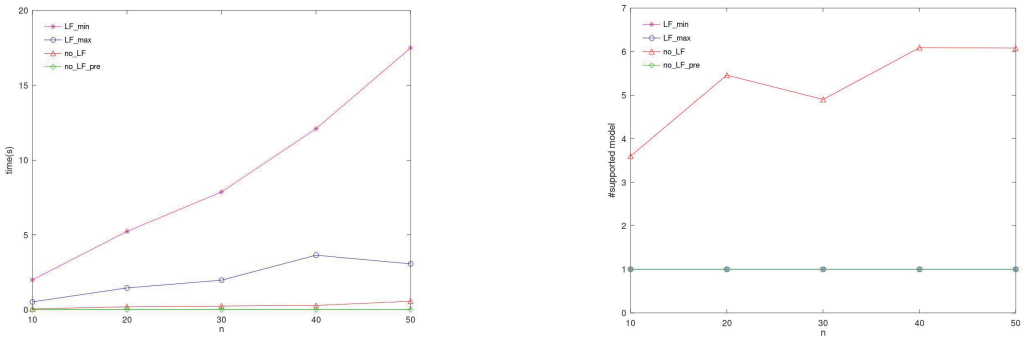
The figure $10^4(0)$ in Table 2 in the case of no use of another solution constraint means Algorithm 1 always exhausts 10^4 trials without finding a stable model (due to implicit bias in Algorithm 1). When another solution constraint is used however, it finds a stable model in 0.09s after 3.5 trials 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 $L_{SU+\widehat{\mathbf{c}}+LF}$ with $max\_try = 10, max\_itr = 100$ on matricized $\mathbf{P}_{4\_n} = (\mathbf{C}_{4\_n}, \mathbf{D}_{4\_n})$ (and no constraint matrix) to compute supported (stable) models. Coefficients in $L_{SU+\widehat{\mathbf{c}}+LF}$ 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.

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}[17].

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

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



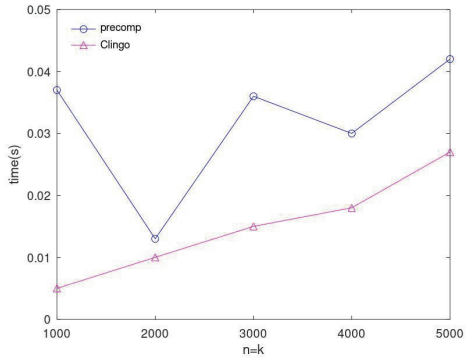
(a) Time to find a stable model (b) #computed models

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

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 $\{a_0, a_1, \ldots, a_n\}$.

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



(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 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 $\mathbf{P}_{5\_nk} = (\mathbf{C}_{5\_nk}, \mathbf{D}_{5\_nk})$ from $\mathbf{D}_{5\_nk}(10001 \times 15002)$ to $\mathbf{D}'_{5\_nk}(5001 \times 10002)$ and from $\mathbf{C}_{5\_nk}(15002 \times 20002)$ to $\mathbf{C}'_{5\_nk}(10002 \times 10002)$. Thus precomputed $\mathbf{P}'_{5\_nk} = (\mathbf{C}'_{5\_nk}, \mathbf{D}'_{5\_nk})$ is downsized to 1/3 of the original 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 $\mathbf{P}'_{5\_nk} = (\mathbf{C}'_{5\_nk}, \mathbf{D}'_{5\_nk})$ detects the stable model only by thresholding $\mathbf{s}$ before

starting any update of **s**. 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. Both NeurASP and SLASH are examples of symbolic ASP solver-based neuro-symbolic systems, where they include a neural frontend to process the perception part of the problem, and a symbolic backend which typically is the ASP solver. Therefore, the neural frontend does not need to be involved in the computational details and problems associated with computing stable models (Section 4).

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. More recently, Takemura and Inoue [31] proposed a neuro-symbolic learning pipeline for distant supervision tasks, which leverages differentiable computation of supported models. Similarly to this work, they encode normal logic programs into matrices and define a differentiable loss function which is based on the supported model semantics.

As for the non-differentiable linear-algebraic approaches to logic programming, Nguyen 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].

6.1. 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 $\mathbf{M}$ and $\mathbf{d} = \mathbf{DM}$. 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 $\mathbf{s}$. Consider the computation of $\mathbf{M} = (\mathbf{1} - \min_1(\mathbf{C}(\mathbf{1} - \mathbf{s}^\delta)))$. We rewrite this using $1 - \min(x, 1) = \mathrm{ReLU}(1 - x)$ to

$$\begin{aligned}\mathbf{M} &= \mathbf{1} - \min_1(\mathbf{C}^{pos}(\mathbf{1} - \mathbf{s}) + \mathbf{C}^{neg}\mathbf{s}) \\ &= \mathrm{ReLU}(\mathbf{W}\mathbf{s} + \mathbf{b}) \\ &\quad \text{where } \mathbf{C} = [\mathbf{C}^{pos}\, \mathbf{C}^{neg}],\ \mathbf{W} = \mathbf{C}^{pos} - \mathbf{C}^{neg},\ \mathbf{b} = \mathbf{1} - \mathbf{C}^{pos}\mathbf{1}\end{aligned}$$

So $\mathbf{M}$ is the output of a ReLU network having a weight matrix $\mathbf{W} = \mathbf{C}^{pos} - \mathbf{C}^{neg}$ and a bias vector $\mathbf{b} = \mathbf{1} - \mathbf{C}^{pos}\mathbf{1}$. Then $\min_1(\mathbf{d}) = \min_1(\mathbf{DM}) = \min_1(\mathbf{D} \cdot \mathrm{ReLU}(\mathbf{Ws}+\mathbf{b}))$ is the output of a ReLU network with a single hidden layer and a linear output layer represented by $\mathbf{D}$ having $\min_1(\cdot)$ as activation function.

Also when we compute a supported model $\mathbf{s}$, we minimize $L_{SU+\widehat{\mathbf{c}}}$ (6) which contains an MSE error term $L_{sq} = \|\min_1(\mathbf{d}) - \mathbf{s}\|^2$ using $J_{a_{SU+\widehat{\mathbf{c}}}}$ (11). This is precisely back propagation from learning data $\mathbf{s}$.

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 $\mathbf{W}$ and a bias vector $\mathbf{b}$ are independent. In our setting, they are interdependent and they faithfully reflect the logical structure of a program.

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.

Future work could focus on improving the integration of neural networks with this proposed end-to-end approach to tackle neuro-symbolic benchmark tasks that require both perception and reasoning. We also aim to improve the optimization techniques, such as precomputation, to enhance efficiency and scalability.

Acknowledgements

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

Appendix A. Proofs

A.1. Proposition 2

Recall that we write $\mathbf{C} \in \{0,1\}^{m\times 2n}$ as $\mathbf{C} = [\mathbf{C}^{pos}\ \mathbf{C}^{neg}]$ where $\mathbf{C}^{pos} \in \{0,1\}^{m\times n}$ (resp. $\mathbf{C}^{neg} \in \{0,1\}^{m\times n}$) is the left half (resp. the right half) of $\mathbf{C}$ representing the positive literals (resp. negative literals) of each rule body in $\mathbf{C}$. Let $\mathbf{M}^{neg} = \mathbf{1} - \min_1(\mathbf{C}^{neg}\mathbf{s}_I)$, and let $\mathbf{M}^{pos} = \mathbf{M}^{neg} \odot (\mathbf{1} - \min_1(\mathbf{C}^{pos}(\mathbf{1} - \mathbf{s}_I)))$.

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 $\|\mathbf{s}_I - \min_1(\mathbf{d}_I)\|_2 = 0$ iff $\|\mathbf{s}_I - \min_1(\mathbf{d}_I^+)\|_2 = 0$.

$I \models \mathrm{comp}(P^I)$ iff $\|\mathbf{s}_I - \min_1(\mathbf{d}_I^+)\|_2 = 0$ is proved similarly to Proposition 1. Firstly, suppose that I is a supported model of P^I. By definition, for each a_i that is true in I, there is at least one rule body in P^I that is true in I, i.e., $I \models a_i$ and $I \models G_{j1} \vee \cdots \vee G_{js}$. Since $\mathbf{M}^{pos}$ denotes the truth values of the rule bodies in P^I evaluated by I, $\mathbf{M}^{pos}(j) = 1$ ($1 \leqslant j \leqslant m$) if r_j^+ is contained in P^I and its body is true in I, otherwise $\mathbf{M}^{pos}(j) = 0$. Let $\mathbf{d}_I^+ = \mathbf{DM}^{pos}$ where $\mathbf{d}_I^+(i) \geqslant 1$ if a_i is true in I, then we have $\mathbf{s}_I(i) = \min_1(\mathbf{d}_I^+)(i)$. Since i is arbitrary, we conclude $\mathbf{s}_I = \mathbf{d}_I^+$, and that $\|\mathbf{s}_I - \min_1(\mathbf{d}_I^+)\|_2 = 0$. Secondly, suppose that $\|\mathbf{s}_I - \min_1(\mathbf{d}_I^+)\|_2 = 0$. Since we have $\min_1(\mathbf{d}_I^+)(i) = 1$ if any of

the rule bodies for a_i is true in I, $I \models G_{j1} \vee \cdots \vee G_{js}$ which we denote $I \models \mathrm{iff}(a_i)$. Because i is arbitrary, and for all atoms that are true in I the above condition holds, we conclude $I \models \mathrm{comp}(P^I)$.

Here, we proved $\|\mathbf{s}_I - \min_1(\mathbf{d}_I)\|_2 = 0$ iff $\|\mathbf{s}_I - \min_1(\mathbf{d}_I^+)\|_2 = 0$, and $I \models \mathrm{comp}(P^I)$ iff $\|\mathbf{s}_I - \min_1(\mathbf{d}_I^+)\|_2 = 0$. From Proposition 1, we have that $I \models \mathrm{comp}(P)$ iff $\|\mathbf{s}_I - \min_1(\mathbf{d}_I)\|_2 = 0$. Hence, $I \models \mathrm{comp}(P)$, $\|\mathbf{s}_I - \min_1(\mathbf{d}_I)\|_2 = 0$, $\|\mathbf{s}_I - \min_1(\mathbf{d}_I^+)\|_2 = 0$ and $I \models \mathrm{comp}(P^I)$ are all equivalent. Q.E.D.

Appendix B. Derivations

B.1. A note on the matrix-matrix dot product notation

B.1.1. $\mathbf{A} \bullet (\mathbf{B} \odot \mathbf{C}) = (\mathbf{B} \bullet \mathbf{A}) \odot \mathbf{C}$

In various parts of this paper, we use the notation of the dot product of matrices $(\mathbf{A} \bullet \mathbf{B}) = \sum_{i,j} \mathbf{A}(i,j)\mathbf{B}(i,j)$. This is essentially an element-wise multiplication (the Hadamard product) followed by the summation of matrix elements. Since the Hadamard product is associative and commutative, it follows that $(\mathbf{A} \odot (\mathbf{B} \odot \mathbf{C})) = ((\mathbf{B} \odot \mathbf{A}) \odot \mathbf{C})$ holds. The summation is also associative and commutative; thus $(\mathbf{A} \bullet (\mathbf{B} \odot \mathbf{C})) = ((\mathbf{B} \bullet \mathbf{A}) \odot \mathbf{C})$ holds.

B.1.2. $\mathbf{A} \bullet (\mathbf{BC}) = (\mathbf{B}^T\mathbf{A}) \bullet \mathbf{C}$

Using the aforementioned $\bullet$ notation, we have $(\mathbf{A} \bullet (\mathbf{BC})) = \sum_{i,j} \mathbf{A}(i,j)(BC)(i,j) = ((\mathbf{B}^T\mathbf{A}) \bullet \mathbf{C}) = \sum_{i,j}(B^T\mathbf{A})(i,j)\mathbf{C}(i,j)$. Without the summation, this will not hold because the matrix multiplication is not commutative in general. Consider a square $N \times N$ matrix $\mathbf{A}, \mathbf{B}$ and $\mathbf{C}$, then we have:

$$\mathbf{A} \odot (\mathbf{BC}) = \begin{bmatrix} \mathbf{A}(1,1)\sum_{i=j}^N \mathbf{B}(1,j)\mathbf{C}(i,1) & \cdots & \mathbf{A}(1,N)\sum_{i=j}^N \mathbf{B}(1,j)\mathbf{C}(i,N) \\ \vdots & & \vdots \\ \mathbf{A}(N,1)\sum_{:i=j}^N \mathbf{B}(N,j)\mathbf{C}(i,1) & \cdots & \underline{\mathbf{A}(N,N)\sum_{i=j}^N \mathbf{B}(N,j)\mathbf{C}(i,N)} \end{bmatrix}$$

and

$$(\mathbf{B}^T\mathbf{A}) \odot \mathbf{C} = \begin{bmatrix} \mathbf{C}(1,1)\sum_i^N \mathbf{A}(i,1)\mathbf{B}(i,1) & \cdots & \mathbf{C}(1,N)\sum_i^N \mathbf{A}(i,N)\mathbf{B}(i,1) \\ \vdots & & \vdots \\ \mathbf{C}(N,1)\sum_i^N \mathbf{A}(i,1)\mathbf{B}(i,N) & \cdots & \mathbf{C}(N,N)\sum_i^N \mathbf{A}(i,N)\mathbf{B}(i,N) \end{bmatrix}$$

Then, observe that an element in one matrix appears in the expanded elements in the columns of the other matrix. For example, consider the $\mathbf{A}(N,N)$ term in $\mathbf{A} \odot (\mathbf{BC})$ (bottom-right), and notice that the term containing $\mathbf{A}(N,N)$ appears only in the rightmost column, and that they appear in the last elements after expansion, i.e.:

$$\begin{aligned} \mathbf{C}(1,N)\sum_i^N \mathbf{A}(i,N)\mathbf{B}(i,1) &= \mathbf{C}(1,N)\Big(\mathbf{A}(1,N)\mathbf{B}(1,1) + \cdots + \mathbf{A}(N,N)\mathbf{B}(N,1)\Big) \\ &= \mathbf{A}(1,N)\mathbf{B}(1,1)\mathbf{C}(1,N) + \cdots + \underline{\mathbf{A}(N,N)\mathbf{B}(N,1)\mathbf{C}(1,N)} \\ \mathbf{C}(N,N)\sum_i^N \mathbf{A}(i,N)\mathbf{B}(i,N) &= \mathbf{C}(N,N)\Big(\mathbf{A}(1,N)\mathbf{B}(1,N) + \cdots + \mathbf{A}(N,N)\mathbf{B}(N,N)\Big) \\ &= \mathbf{A}(1,N)\mathbf{B}(1,N)\mathbf{C}(N,N) + \cdots + \underline{\mathbf{A}(N,N)\mathbf{B}(N,N)\mathbf{C}(N,N)} \end{aligned}$$

Since the summation operation in $\bullet$ is commutative, one can collect the underlined parts to the summation, $\mathbf{A}(N,N)\sum_{i=j}^N \mathbf{B}(N,j)\mathbf{C}(i,N)$, which is the bottom-right element in $\mathbf{A} \odot (\mathbf{BC})$. One can show that this applies to all other elements of the matrix, e.g., the $\mathbf{C}(N,N)$ term in $(\mathbf{B}^T\mathbf{A}) \odot \mathbf{C}$ appears in the last elements in the rightmost column of $\mathbf{A} \odot (\mathbf{BC})$. Therefore, $(\mathbf{A} \bullet (\mathbf{BC})) = ((\mathbf{B}^T\mathbf{A}) \bullet \mathbf{C})$ holds.

B.2. $J_{a_{SU}}$: Jacobian for supported model computation (Section 3.3)

Let $\mathbf{P} = (\mathbf{C}, \mathbf{D})$ be the matricized program and write $\mathbf{C} = [\mathbf{C}^{pos}\, \mathbf{C}^{neg}]$. Introduce **N**, **M**, **d**, **E**, **F** and compute L_{SU} by

$$
\begin{array}{lcll}
\mathbf{N} & = & \mathbf{C}(\mathbf{1}-\mathbf{s}^{\delta}) = \mathbf{C}^{pos}(\mathbf{1}-\mathbf{s}) + \mathbf{C}^{neg}\mathbf{s} & \text{: (continuous) counts of false literals in the rule bodies} \\
\mathbf{M} & = & \mathbf{1} - \min_1(\mathbf{N}) & \text{: (continuous) truth values of the rule bodies} \\
\mathbf{d} & = & \mathbf{DM} & \text{: (continuous) counts of true disjuncts for each atom} \\
\mathbf{E} & = & \min_1(\mathbf{d}) - \mathbf{s} & \text{: error between the estimated truth values of atoms and } \mathbf{s} \\
\mathbf{F} & = & \mathbf{s} \odot (\mathbf{1}-\mathbf{s}) & \text{: (continuous) 0 iif } \mathbf{s} \text{ is binary} \\
L_{sq} & = & (\mathbf{E} \bullet \mathbf{E}) & \\
L_{nrm} & = & (\mathbf{F} \bullet \mathbf{F}) & \\
L_{SU} & = & 0.5 \cdot (L_{sq} + \ell_2 \cdot L_{nrm}). &
\end{array}
$$

We first compute $\dfrac{\partial L_{sq}}{\partial s_p}$ where $s_p = \mathbf{s}(p)$ $(1 \leqslant p \leqslant n)$.

$$
\begin{aligned}
\frac{\partial \mathbf{M}}{\partial s_p} &= -[\mathbf{N} \leqslant 1] \odot \left((\mathbf{C}^{neg} - \mathbf{C}^{pos})\mathbf{I}_p\right) = [\mathbf{N} \leqslant 1] \odot \left((\mathbf{C}^{pos} - \mathbf{C}^{neg})\mathbf{I}_p\right) \\
\frac{\partial L_{sq}}{\partial s_p} &= \left(\mathbf{E} \bullet [\mathbf{DM} \leqslant 1] \odot \left(D\left(\frac{\partial M}{\partial s_p}\right)\right) - \mathbf{I}_p\right) \\
&= \left(\mathbf{E} \bullet [\mathbf{DM} \leqslant 1] \odot (\mathbf{D}([N \leqslant 1] \odot ((\mathbf{C}^{pos} - \mathbf{C}^{neg})\mathbf{I}_p))) - \mathbf{I}_p\right) \\
&= \left(\mathbf{D}^T([\mathbf{DM} \leqslant 1] \odot E) \bullet [N \leqslant 1] \odot (((\mathbf{C}^{pos} - \mathbf{C}^{neg})\mathbf{I}_p)\right) - (E \bullet \mathbf{I}_p) \\
&= \left((\mathbf{C}^{pos} - \mathbf{C}^{neg})^T([\mathbf{N} \leqslant 1] \odot (\mathbf{D}^T([\mathbf{DM} \leqslant 1] \odot E))) - E \bullet \mathbf{I}_p\right)
\end{aligned}
$$

Since p is arbitrary, we have $\dfrac{\partial L_{sq}}{\partial \mathbf{s}} = (\mathbf{C}^{pos} - \mathbf{C}^{neg})^T([\mathbf{N} \leqslant 1] \odot (\mathbf{D}^T([\mathbf{DM} \leqslant 1] \odot \mathbf{E}))) - \mathbf{E}$.

Next we compute $\dfrac{\partial L_{nrm}}{\partial s_p}$:

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

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

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

B.3. $J_{a_{\widehat{\mathbf{c}}}}$: Jacobian for constraints (Section 3.4)

Let $\widehat{\mathbf{C}} = [\widehat{\mathbf{C}}^{pos}\,\widehat{\mathbf{C}}^{neg}]$ represent the rule bodies of constraints in a binary matrix. The rest of the derivation is similar to the previous section, namely the derivation of $\dfrac{\partial L_{sq}}{\partial \mathbf{s}}$.

$$\mathbf{N}_{\widehat{c}} = \widehat{\mathbf{C}}(\mathbf{1}-\mathbf{s}^{\delta}) = \widehat{\mathbf{C}}^{pos}(\mathbf{1}-\mathbf{s}) + \widehat{\mathbf{C}}^{neg}\mathbf{s}$$

$$L_{\widehat{\mathbf{c}}} = (\mathbf{1} \bullet (\mathbf{1} - \min_1(\mathbf{N}_{\widehat{c}}))) \quad \text{where } \mathbf{1} \text{ is an all-ones vector}$$

$$\begin{aligned}\frac{\partial J_{a_{\widehat{\mathbf{c}}}}}{\partial \mathbf{s}} &= -[\mathbf{N}_{\widehat{c}} \leqslant 1] \odot \big((\widehat{\mathbf{C}}^{neg} - \widehat{\mathbf{C}}^{pos})\mathbf{I}\big) \\ &= (\widehat{\mathbf{C}}^{pos} - \widehat{\mathbf{C}}^{neg})^T [\mathbf{N}_{\widehat{c}} \leqslant 1]\end{aligned}$$

B.4. $J_{a_{LF}}$: Jacobian for loop formula (Section 4.1)

Let $\mathbf{C} = [\mathbf{C}^{pos}\,\mathbf{C}^{neg}]$, and let S_v, A_v and $\mathbf{M}$ be computed by (17).

$$\begin{aligned}\mathbf{N} &= \mathbf{C}(\mathbf{1}-\mathbf{s}^{\delta}) \\ N_v &= \mathbf{S}_v(\mathbf{1}-\mathbf{s}) \\ M_v &= \min_1(N_v) \\ J_{a_{LF}} &= \frac{\partial L_{LF}}{\partial \mathbf{s}} = \sum_{v=1}^{w} -\left(\frac{\partial \min_1(A_v)}{\partial \mathbf{s}}\right) \\ &= -\sum_{v=1}^{w} [A_v \leqslant 1]\left(\left(\frac{\partial M_v}{\partial \mathbf{s}}\right) + \mathbf{S}_v\mathbf{E}_{sup}\left(\frac{\partial \mathbf{M}}{\partial \mathbf{s}}\right)^T\right) \\ &= \sum_{v=1}^{w} [A_v \leqslant 1]\left([N_v \leqslant 1]\mathbf{S}_v^T + (((\mathbf{S}_v\mathbf{E}_{sup}) \odot [\mathbf{N} \leqslant 1]^T)(\mathbf{C}^{neg} - \mathbf{C}^{pos})^T\right)\end{aligned}$$

Appendix C. Encoding the Hamiltonian Cycle Problem

This section describes the encoding and program used in solving the Hamiltonian cycle problem (Section 5.2).

Firstly, looking at the graph (Fig. 4a), it is evident that there are six vertices. We use an atom $H_{i,j}$ to indicate there exists an edge from vertices i to j in an HC. Then there are 36 atoms $\{H_{1,1}, H_{1,2}, \cdots, H_{1,6}, H_{2,1}, \cdots, H_{6,1}, \cdots, H_{6,6}\}$. We also use an atom $U_{j,q}$ to indicate that the vertex j is visited at time q. Then there are 36 atoms $\{U_{1,1}, U_{1,2}, \cdots, U_{1,6}, U_{2,1}, \cdots, U_{6,1}, \cdots, U_{6,6}\}$. Thus, in total, there are 72 atoms consisting of $H_{i,j}$ and $U_{j,q}$.

C.1. Encoding (1) (2) (3) into a program

The condition one_of is encoded as a set of normal rules whose body consists solely of negative literals. For example, looking at vertex 1, there are 3 outgoing edges to vertices 2, 3 and 4. Then, we construct rules for (1):

$$P_{H_{1,j}} = \begin{cases} H_{1,2} \leftarrow \neg H_{1,3} \wedge \neg H_{1,4} \\ H_{1,3} \leftarrow \neg H_{1,2} \wedge \neg H_{1,4} \\ H_{1,4} \leftarrow \neg H_{1,2} \wedge \neg H_{1,3} \end{cases}$$

In encoding this into a program matrix, we also create an empty (zero) row for a rule with a head atom that does not appear in the program, so there will be $36 = 6 \times 6$ rules for (1).

For encoding rules with $U_{j,q}$ in the head, consider the following: (a) since we know $U_{1,1}$ is always true (starting point), $U_{1,q}(2 \leqslant q \leqslant 6)$ will always be false, and (b) for atoms $U_{j,q(j\neq 1)}$, $U_{j,1}$ is always false, and for each q in $2 \leqslant q \leqslant 6$, we generate 6 rules, i.e., there will be $31 = 1 + (5 \times 6)$. Therefore, there will be $161 = 6 + (5 \times 31)$ rules for (2).

By combining the program matrices for (1) and (2) (because (3) is a fact, it is omitted here), we obtain a 197×144 program matrix $\mathbf{C}_3$.

C.2. Encoding (4) (5) (6) into constraints

Encoding of (5) is straightforward and results in 5 constraints $(2 \leqslant i \leqslant 6)$. Encoding of (4) involves an XOR constraint for each vertex, for example, for vertex $i = 1$, we have the following constraint: $\leftarrow \neg H_{1,1} \wedge \neg H_{2,1} \wedge \cdots \wedge \neg H_{6,1}$. Thus, encoding of (4) results in 6 constraints. For encoding (6), we construct these constraints in two parts: (a) each vertex is visited at least once, and (b) each vertex is visited at most once. The first part (a) is straightforward: for example, the constraint for $i = 2$ is $\leftarrow \neg U_{2,1} \wedge \neg U_{2,2} \wedge \neg \cdots \neg U_{2,6}$. Thus, (a) results in 6 constraints. The encoding of (b) requires $\mathrm{C}_2^5 = 10$ constraints for each $U_j (j \geqslant 2)$. For example, for U_2:

$$\begin{cases} \leftarrow U_{2,2} \wedge U_{2,3} \\ \leftarrow U_{2,2} \wedge U_{2,4} \\ \vdots \\ \leftarrow U_{2,2} \wedge U_{2,6} \\ \leftarrow U_{2,3} \wedge U_{2,4} \\ \vdots \\ \leftarrow U_{2,3} \wedge U_{2,6} \\ \leftarrow U_{2,4} \wedge U_{2,5} \\ \leftarrow U_{2,4} \wedge U_{2,6} \\ \leftarrow U_{2,5} \wedge U_{2,6} \end{cases}$$

Thus, (b) results in 50 constraints. By combining all constraints, we obtain $67 = 5 + 6 + 6 + 50$ constraints K_3.

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.Q. Nguyen, 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.
[30] V. Lifschitz and A. Razborov, Why Are There so Many Loop Formulas?, *ACM Transactions on Computational Logic*, **7**, (2), (2006), pp. 261–268.
[31] A. Takemura and K. Inoue, Differentiable Logic Programming for Distant Supervision, *ECAI 2024 - 27th European Conference on Artificial Intelligence, Santiago de Compostela, Spain - Including 13th Conference on Prestigious Applications of Intelligent Systems (PAIS 2024)*, U. Endriss, F.S. Melo, K. Bach, A.J. Bugarín Diz, J.M. Alonso-Moral, S. Barro and F. Heintz, Frontiers in Artificial Intelligence and Applications, **392**, 2024, pp. 1301–1308, IOS Press, doi:10.3233/FAIA240628.
[32] D.B. West, *Introduction to Graph Theory*, Prentice Hall, 2001, 978-0-13-014400-3.