Computing Logic Programming Semantics in Linear Algebra

Hien D. Nguyen[1], Chiaki Sakama[2], Taisuke Sato[3], Katsumi Inoue[4]

[1]University of Informatics Technology, VNU-HCM, Vietnam
[2]Wakayama University, Japan
[3]AI Research Center AIST, Japan
[4]National Institute of Informatics (NII), Japan
hiennd@uit.edu.vn, sakama@sys.wakayama-u.ac.jp, satou.taisuke@aist.go.jp, inoue@nii.ac.jp

Abstract. Logic programming is a logic-based programming paradigm, and provides languages for declarative problem solving and symbolic reasoning. In this paper, we develop new algorithms for computing logic programming semantics in linear algebra. We first introduce an algorithm for computing the least model of a definite logic program using matrices. Next, we introduce an algorithm for computing stable models of a normal logic program. We also develop optimization techniques for speeding-up those algorithms. Finally, the complexity of them is analyzed and tested in practice.

Keywords. logic programming, linear algebra, definite program, normal program.

1 Introduction

Logic programming provides languages for declarative problem solving and symbolic reasoning, yet symbolic computation often suffers from the difficulty to make it scalable. On the other hand, linear algebra is the core of many applications of scientific computation and has been equipped with several scalable techniques. However, linear algebra has not been used for knowledge representation, since it only handles numbers and does not manipulate symbols. Then, it has recently been recognized that the integration of linear algebraic computation and symbolic computation is one of the promising and challenging topics in AI [1]. Such integration has potential to make symbolic reasoning scalable to real-life datasets. There are some studies for realizing logical reasoning using linear algebra.

In [2], Sato translates the rules of a Datalog program into a set of matrix equations. Based on this, the least model of a Datalog program is computed by solving the matrix equations. Yang, *et al.* [3] consider learning representations of entities and relations in knowledge bases using the neural-embedding approach. This method is applied for mining Horn clauses from relational facts represented in a vector space. Real Logic [6] is a framework where learning from numerical data and logical reasoning are integrated using first order logic syntax. Logic tensor networks (LTN) are one of the modern methods for reasoning and learning from the concrete data as real logic. This method has been applied to semantic image interpretation [7].

Logic programming can be represented based on multilinear algebra. In [4, 5], the authors used multilinear maps and tensors to represent predicates, relations, and logical atoms of a predicate calculus. Lin [11] introduces linear algebraic computation of

SAT for clausal theories. Besides that, Horn, disjunctive and normal logic programs are also represented by algebraic manipulation of tensors in [8]. The study builds a new theory of logic programming, while implementation and evaluation are left open.

In this paper, we first refine the framework of [8] and present algorithms for finding the least model [9] of a definite program and stable models [12] of a normal program. Some optimization techniques for speeding-up these algorithms are studied. These methods are developed based on the structure of matrices representing logic programs. The complexity of proposed algorithms is evaluated and tested in practice.

The next section presents an algorithm for computing the least model of a definite program and an improved method for this algorithm. Section 3 presents an algorithm for finding stable models of a normal program and its improvement. Section 4 shows experimental results by testing in practice. The last section concludes the paper. Due to space limit, we omit some proofs of propositions and theorems.

2 Definite Programs

We consider a language $\mathcal{L}$ that contains a finite set of propositional variables and the logical connectives $\neg$, $\wedge$, $\vee$ and $\leftarrow$. Given a logic program P, the set of all propositional variables appearing in P is the *Herband base* of P, denoted B_P.

2.1 Preliminaries

A *definite program* is a finite set of *rules* of the form:

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

where h and b_i are propositional variables. A rule r is called *d-rule* iff r has the form:

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

where h, b_i are propositional variables. A *d-program* is a finite set of rules that are either (1) or (2). Note that the rule (2) is a shorthand of m rules: $h \leftarrow b_1, \ldots, h \leftarrow b_m$, so a d-program is also a definite program. Given a rule r as (1) or (2), define $head(r) = h$ and $body(r) = \{b_1, \ldots, b_m\}$. In particular, the rule is a *fact* if $body(r) = \emptyset$.

A set $I \subseteq B_P$ is an *interpretation* of P. An interpretation I is a *model* of a d-program P if $\{b_1, \ldots, b_m\} \subseteq I$ implies $h \in I$ for every rule (1) in P, and $\{b_1, \ldots, b_m\} \cap I \neq \emptyset$ implies $h \in I$ for every rule (2) in P. A model I is the *least model* of P if $I \subseteq J$ for any model J of P. The T_P*- operator* is a mapping $T_P : 2^{B_P} \rightarrow 2^{B_P}$ which is defined as:

$$T_P(I) = \{h \mid h \leftarrow b_1 \wedge \ldots \wedge b_m \in P \text{ and } \{b_1, \ldots, b_m\} \subseteq I\} \cup$$
$$\{h \mid h \leftarrow b_1 \vee \ldots \vee b_n \in P \text{ and } \{b_1, \ldots, b_n\} \cap I \neq \emptyset\}.$$

The powers of T_P are defined as: $T_P^{k+1}(I) = T_P\left(T_P^k(I)\right)$ and $T_P^0(I) = I$ $(k \geq 0)$.

Given $I \subseteq B_P$, there is a *fixpoint* $T_P^{n+1}(I) = T_P^n(I)$ $(n \geq 0)$. For a definite program P, the fixpoint $T_P^n(\emptyset)$ coincides with the least model of P [9].

2.2 SD-program

We first consider a subclass of definite programs, called SD-programs.

Definition 2.1: (SD-program) A definite program P is called *singly defined* (or SD-*program*) if $head(r_1) \neq head(r_2)$ for any two rules r_1 and r_2 in P $(r_1 \neq r_2)$.

Definition 2.2 [8]: **(interpretation vector)** Let P be a definite program and $B_P = \{p_1, \ldots, p_n\}$. Then an interpretation I of P is represented by a vector $v = (a_1, \ldots, a_n)^T$ where each element a_i represents the truth value of the proposition p_i such that $a_i = 1$ if $p_i \in I$ $(1 \le i \le n)$; otherwise, $a_i = 0$. We write $\text{row}_i(v) = p_i$. Given $v = (a_1, \ldots, a_n)^T \in \mathbb{R}^n$, $v[i]$ is the ith element of v $(1 \le i \le n)$ and $v[1 \ldots k]$ is a vector $(a_1, \ldots, a_k)^T \in \mathbb{R}^k$ $(k \le n)$.

Definition 2.3: (matrix representation of a SD-program)[1] Let P be an SD-program and $B_P = \{p_1, \ldots, p_n\}$. Then P is represented by a matrix $M_P \in \mathbb{R}^{n \times n}$ such that for each element a_{ij} $(1 \le i, j \le n)$ in M_P:

1. $a_{ij_k} = \dfrac{1}{m}$ $(1 \le k \le m;\ 1 \le i, j_k \le n)$ if $p_i \leftarrow p_{j_1} \wedge \ldots \wedge p_{j_m}$ is in P.
2. $a_{ii} = 1$ if $p_i \leftarrow$ is in P.
3. $a_{ij} = 0$, otherwise.

M_P is called a *program matrix*. We write $\text{row}_i(M_P) = p_i$ and $\text{col}_j(M_P) = p_j$ $(1 \le i, j \le n)$.

By the condition of an SD-program, there is at most one rule $r \in P$ such that $head(r) = p$ for each $p \in B_P$. Then no two rules are encoded in a single row of M_P.

Definition 2.4: **(initial vector)** Let P be a definite program and $B_P = \{p_1, \ldots, p_n\}$. Then the *initial vector* is an interpretation $v_o = (a_1, \ldots, a_n)^T$ such that $a_i = 1$ if $\text{row}_i(v_o) = p_i$ and a fact $p_i \leftarrow$ is in P $(1 \le i \le n)$; otherwise, $a_i = 0$.

Definition 2.5 [8]: **(thresholding function)** Given a vector $v = (a_1, \ldots, a_n)^T \in \mathbb{R}^n$, define $\theta(v) = (a_1', \ldots, a_n')^T$ where $a_i' = 1$ $(1 \le i \le n)$ if $a_i \ge 1$; otherwise, $a_i' = 0$. We call it the θ-*thresholding function* of v.

Given a program matrix $M_P \in \mathbb{R}^{n \times n}$ and an initial vector $v_o \in \mathbb{R}^n$, define:

$$v_{k+1} = \theta(M_P v_k) \quad (k \ge 0).$$

It holds that $v_{k+1} = v_k$ for some $k \ge 0$. When $v_{k+1} = v_k$, we write: $v_k = FP(M_P v_o)$.

Theorem 2.1: Let P be an SD-program and $M_P \in \mathbb{R}^{n \times n}$ its program matrix. Then $m \in \mathbb{R}^n$ is a vector representing the least model of P iff $m = FP(M_P v_o)$ where v_o is the initial vector of P.

Example 2.1: Consider the program $P = \{p \leftarrow q,\ q \leftarrow p \wedge r,\ r \leftarrow s,\ s \leftarrow\ \}$ with $B_P = \{p, q, r, s\}$ then its program matrix $M_P \in \mathbb{R}^{4 \times 4}$ is the matrix (right). The initial vector of P is $v_o = (0\ 0\ 0\ 1)^T$. Then, $v_1 = \theta(M_P v_0) = (0\ 0\ 1\ 1)^T$ and $v_2 = \theta(M_P v_1) = (0\ 0\ 1\ 1)^T = v_1$. Hence, the vector v_1 represents the least model $\{r, s\}$ of P.

$$\begin{array}{c} \begin{array}{cccc} p & q & r & s \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 \\ ½ & 0 & ½ & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{array}{c} p \\ q \\ r \\ s \end{array} \end{array}$$

Note that the fact $s \leftarrow$ in P is encoded as the rule $s \leftarrow s$ in M_P. By multiplying M_P and v_0, the 4th element of v_1 represents the truth of s after one-step of deduction.

The study [8] also introduces fixpoint computation of least models. Differently from the current study, [8] works on a program satisfying the MD-condition[2] and sets the empty set as the initial vector for computing fixpoint. In this paper, we work on an

---

[1] In [8], the fact is represented by "$p_i \leftarrow \top$" and is encoded in a matrix by $a_{ij} = 1$ where $\text{row}_i(M_P) = p_i$ and $\text{col}_j(M_P) = \top$.

[2] A definite program P satisfies the *MD-condition* if it satisfies the following condition: For any two rules r_1 and r_2 in P $(r_1 \ne r_2)$: $head(r_1) = head(r_2)$ implies $|body(r_1)| \le 1$ and $|body(r_2)| \le 1$.

SD-program and start with the initial vector representing facts, and facts $p \leftarrow$ in P are encoded by the rule $p \leftarrow p$ rather than $p \leftarrow \top$. This has the effect of reducing non-zero elements in matrices during fixpoint computation. [8] allows the existence of constraints "$\leftarrow q$" in a program while the current study does not. Those constraints are handled as a rule "$p \leftarrow q, \neg p$" in Section 3.

2.3 Non-SD programs

When a definite program P contains two rules: r_1: $h \leftarrow b_1 \wedge \ldots \wedge b_m$ and r_2: $h \leftarrow c_1 \wedge \ldots \wedge c_n$, P is transformed to a d-program Q such that:

$$Q = (P \setminus \{r_1, r_2\}) \cup \{r_1', r_2', d_1\}$$

where r_1': $h_1 \leftarrow b_1 \wedge \ldots \wedge b_m$, r_2': $h_2 \leftarrow c_1 \wedge \ldots \wedge c_n$ and d_1: $h \leftarrow h_1 \vee h_2$.
Here, h_1 and h_2 are new propositional variables associated with r_1 and r_2, respectively.

Generally, a non-SD program is transformed to a d-program as follows.

Definition 2.6: (transformation) Let P be a definite program and B_P its Herband base. For each $p \in B_P$, put $P_p = \{ r \mid r \in P$ and $head(r) = p\}$ and $R_p = \{ r \mid r \in P_p$ and $\mid P_p \mid = k > 1\}$. Then define $S_p = \{p_i \leftarrow body(r) \mid r \in R_p, i = 1,\ldots,k\}$ and $D_p = \{ p \leftarrow p_1 \vee \ldots \vee p_k \}$. Build a d-program:

$$P' = \underbrace{(P \setminus \bigcup_{p \in B_P} R_p) \cup \bigcup_{p \in B_P} S_p}_{Q} \cup \underbrace{\bigcup_{p \in B_P} D_p}_{D}$$

We have $P' = Q \cup D$ where Q is an SD-program and D is a set of d-rules.

It is easily shown that a d-program P' has the least model M' such that $M' \cap B_P = M$ where M is the least model of P.

Definition 2.7: (matrix representation of a d-program) Let P' be a d-program such that $P' = Q \cup D$ where Q is an SD-program and D is a set of d-rules, and $B_{P'} = \{p_1, \ldots, p_m\}$ the Herband base of P'. Then P' is represented by a matrix $M_{P'} \in \mathbb{R}^{m \times m}$ such that for each element a_{ij} $(1 \le i, j \le m)$ in $M_{P'}$:

1. $a_{ij_k} = 1$ $(1 \le k \le l;\ 1 \le i, j_k \le m)$ if $p_i \leftarrow p_{j_1} \vee \ldots \vee p_{j_l}$ is in D.
2. Otherwise, every rule in Q is encoded as in Def. 2.3.

Theorem 2.2: Let P' be a d-program and $M_{P'} \in \mathbb{R}^{m \times m}$ its program matrix. Then $u \in \mathbb{R}^m$ is a vector representing the least model of P' iff $u = FP(M_{P'} v_o)$ where v_o is the initial vector of P'.

2.4 Algorithms for finding least models

Based on Theorem 2.2, we develop an algorithm for computing the least model of a definite program P (Fig. 1)

Example 2.2: Consider $P = \{p \leftarrow q,\ p \leftarrow r \wedge s,\ r \leftarrow s,\ s \leftarrow \}$ and $B_P = \{p, q, r, s\}$. Then P is transformed to a d-program $P' = Q \cup D$ with $B_{P'} = \{p, q, r, s, t, u\}$:

$$Q = \{ t \leftarrow q,\ \ u \leftarrow r \wedge s,\ \ r \leftarrow s,\ \ s \leftarrow \}.$$

$$D = \{ p \leftarrow t \vee u \}.$$

Algorithm 2.1:
Input: a definite program P, and its Herband base $B_P = \{p_1,\ldots,p_n\}$.
Output: a vector u representing the least model of P.

Step 1: Transform a definite program P to a d-program $P' = Q \cup D$ with $B_{P'} = \{p_1,\ldots,p_n, p_{n+1},\ldots,p_m\}$, where Q is an SD-program and D is a set of d-rules. **Step 2:** - Create the matrix $M_{P'} = (a_{ij})_{1\leq i,j\leq m}$ representing a d-program P'. - Create the initial vector $v_o = (v_1,\ldots v_m)$ of P'.	**Step 3:** *Compute the least model of P'* $v := v_o$ $u := \theta(M_{P'} v)$ while $u \neq v$ do $v := u;$ $u := \theta(M_{P'} v)$ end do; #while return $u[1\ldots n]$;

Fig. 1. Algorithm 2.1

We have the matrix $M_{P'} \in \mathbb{R}^{6\times6}$ representing P' (right). Let $v_0 = (0\ 0\ 0\ 1\ 0\ 0)^T$ be a vector representing facts in P'. Then, $v_1 = \theta(M_{P'}v_0) = (0\ 0\ 1\ 1\ 0\ 0)^T$, $v_2 = \theta(M_{P'}v_1) = (0\ 0\ 1\ 0\ 1)^T$, $v_3 = \theta(M_{P'}v_2) = (1\ 0\ 1\ 1\ 0\ 1)^T$, $v_4 = \theta(M_{P'}v_3) = (1\ 0\ 1\ 1\ 0\ 1)^T = v_3$. Hence, v_3 is a vector representing the least model of P', and $v_3[1\ldots4]$ is a vector representing the least model of P, that represents $\{p, r, s\}$.

$$M_{P'} = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & ½ & ½ & 0 & 0 \end{pmatrix}$$

(columns: $p\ q\ r\ s\ t\ u$; rows: $p\ q\ r\ s\ t\ u$)

In Algorithm 2.1, the complexity of computing $M_{P'}v$ is $O(m^2)$ and computing $\theta(.)$ is $O(m)$. The number of times for iterating $M_{P'}v$ is at most $(m + 1)$ times. So the complexity of Step 3 is $\mathbf{O((m + 1) \times (m + m^2)) = O(m^3)}$ in the worst case. Comparing the current study with the previous one [8], the current encoding has an advantage of increasing zero elements in matrices and reducing the number of required iterations in fixpoint computation.

2.5 Column reduction

This section introduces a method for decreasing the complexity of computing $u = \theta(M_{P'}v)$.

Definition 2.8: (submatrix representation of a d-program) Let P' be a d-program such that $P' = Q \cup D$ where Q is an SD-program and D is a set of d-rules, and $B_{P'} = \{p_1,\ldots, p_m\}$ the Herband base of P'. Then P' is represented by a matrix $N_{P'} \in \mathbb{R}^{m\times n}$ such that each element b_{ij} $(1 \leq i \leq m, 1 \leq j \leq n)$ in $N_{P'}$ is equivalent to the corresponding element a_{ij} $(1 \leq i, j \leq m)$ in $M_{P'}$ of Def.2.7. $N_{P'}$ is called a *submatrix* of P'.

Note that the size of $M_{P'} \in \mathbb{R}^{m\times m}$ of Def.2.7 is reduced to $N_{P'} \in \mathbb{R}^{m\times n}$ in Def.2.8 by $n \leq m$.

Definition 2.9: (θ_D-thresholding) Given a vector $v = (a_1,\ldots, a_m)^T$, define a vector $w = \theta_D(v) = (w_1,\ldots, w_m)^T$ such that (i) $w_i = 1$ $(1 \leq i \leq m)$ if $a_i \geq 1$, (ii) $w_i = 1$ $(1 \leq i \leq n)$ if $\exists j$ $w_j = 1$ $(n + 1 \leq j \leq m)$ and there is a d-rule $d \in D$ such that $head(d) = p_i$ and $\text{row}_j(w) \in body(d)$, and (iii) otherwise, $w_i = 0$. $\theta_D(v)$ is called a θ_D*-thresholding* of v.

Intuitively speaking, the additional condition of Def.2.9(ii) says "if an element in the body of a d-rule is 1, then the element in the head of the d-rule is set to 1". θ_D adds

this condition to θ, in return for reducing columns. By definition, it holds that $\theta_D(v) = \theta_D(\theta(v))$.

$\theta_D(v)$ is computed by checking the value of a_i for $1 \le i \le m$ and checking all d-rules for $n + 1 \le j \le m$. Since the number of d-rules is at most n, the complexity of computing $\theta_D(.)$ is $O(m + (m - n) \times n) = O(m \times n)$.

Theorem 2.3: Let P be a definite program with $B_P = \{p_1,...,p_n\}$, and P' a transformed d-program with $B_{P'} = \{p_1,...,p_n, p_{n+1},...,p_m\}$. Let $N_{P'} \in \mathbb{R}^{m\times n}$ be a submatrix of P'. Given a vector $v \in \mathbb{R}^n$ representing an interpretation I of P, let $u = \theta_D(N_{P'}v) \in \mathbb{R}^m$.

Then u is a vector representing an interpretation J of P' such that $J \cap B_P = T_P(I)$.

Proof:

Let $v = (v_1,..., v_n)^T$ then $N_{P'}v = (x_1,...,x_n,\dots,x_m)^T$ with $x_k = a_{k1}v_1 +\dots+ a_{kn}v_n$ $(1 \le k \le m)$.
Suppose $w = \theta(N_{P'}v) = (w_1,...,w_m)^T$ and $u = \theta_D(N_{P'}v) = \theta_D(w) = (u_1, ..., u_m)^T$.

(I) First, we prove $J \cap B_P \subseteq T_P(I)$.

Let $u_k = 1$ $(1 \le k \le n)$ and $p_k = row_k(u)$. We show $p_k \in T_P(I)$.

(i) Assume $w_k = u_k = 1$. By $w_k = 1$, $x_k = a_{k1}v_1 +\dots+ a_{kn}v_n \ge 1$.

Let $\{b_1,\dots, b_r\} \subseteq \{a_{k1},\dots, a_{kn}\}$ such that $b_i \ne 0$ $(1 \le i \le r)$. Then, $b_i = 1/r$ $(1 \le i \le r)$ and $b_1v_{b1} +\dots+ b_rv_{br} = 1$ imply $v_{b1} = \dots= v_{br} = 1$.

In this case, there is a rule: $p_k \leftarrow p_{b1} \wedge \dots \wedge p_{br}$ in P such that $p_{bi} = \mathrm{col}_i(N_{P'})$ for $b_i = a_{ki}$ $(1 \le i \le r)$ and $\{p_{b1},\dots, p_{br}\} \subseteq I$. Hence, $p_k \in T_P(I)$ holds.

(ii) Next assume $u_k \ne w_k$, then $w_k = 0$.

As $w_k = 0$, by definition of $\theta_D(.)$, $\exists\ j$, $n + 1 \le j \le m$ such that $w_j = 1$. Also, $\exists d_j \in D$ such that $\mathrm{row}_j(w) = p_j \in body(d_j)$ and $head(d_j) = p_k$

where d_j has the form: $p_k \leftarrow p_{k1} \vee\dots\vee p_{kq}$ with $p_j \in\{p_{k1},\dots, p_{kq}\} \subseteq B_{P'}\backslash B_P$.

By $w_j = 1$ $(n +1 \le j \le m)$, it holds $x_j = a_{j1}v_1 +\dots+ a_{jn}v_n \ge 1$.

Let $\{b_1,\dots, b_r\} \subseteq \{a_{j1},\dots, a_{jn}\}$ such that $b_i \ne 0$ $(1 \le i \le r)$. Then,
$b_i = 1/r$ $(1 \le i \le r)$ and $b_1v_{b1} +\dots+ b_rv_{br} = 1$ imply $v_{b1} = \dots= v_{br} = 1$.

In this case, there is a rule: $p_j \leftarrow p_{b1} \wedge \dots \wedge p_{br}$ in Q such that $p_{bi} = \mathrm{col}_j(N_{P'})$ for $b_i = a_{ki}$ $(1 \le i \le r)$ and $\{p_{b1},\dots, p_{br}\} \subseteq I$. By transforming a definite program P to a d-program P', we have: $p_k \leftarrow p_{b1} \wedge \dots \wedge p_{br} \in P$ and $\{p_{b1},\dots, p_{br}\} \subseteq I$. Hence, $p_k \in T_p(I)$.

In both (i) and (ii), it holds that $p_k = \mathrm{row}_k(u) \in T_p(I)$ if $u_k = 1$ $(1 \le k \le n)$.

Then $J \cap B_P \subseteq T_P(I)$.

(II) Next, we prove $T_P(I) \subseteq J \cap B_P$. We show that $p_k \in T_P(I)$ implies $u_k = 1$ $(1 \le k \le n)$.

Let $p_k \in T_P(I)$, then there is $p_k \leftarrow p_{k1} \wedge \dots \wedge p_{kr} \in P$ $(1 \le k \le n)$ such that $\{p_{k1},\dots, p_{kr}\} \subseteq I$, so $v_{kj} = 1$ $(1 \le j \le r)$.

(i) If $p_k \leftarrow p_{k1} \wedge \dots \wedge p_{kr} \in Q$ then $p_{kj} \in B_P$. Then $\exists\ i$, $p_{kj} = \mathrm{col}_i(N_{P'})$ $(1 \le i \le n)$ and $a_{kj} = 1/r$. Hence, $x_k = a_{k1}v_1 +\dots+ a_{kn}v_n = 1$ and $w_k = 1 = u_k$.

(ii) Else if $p_k \leftarrow p_{k1} \wedge \dots \wedge p_{kr} \notin Q$ then $\exists\ j$, $n + 1 \le j \le m$, $p_j \leftarrow p_{j1} \wedge \dots \wedge p_{jr} \in Q$ and $p_k \leftarrow p_{k1} \vee\dots\vee p_{kq} \in D$ with $p_j \in\{p_{k1},\dots, p_{kq}\}$.

By $p_j \leftarrow p_{j1} \wedge \dots \wedge p_{jr} \in Q$, it holds $p_{ji} \in B_P$. Then $\exists\ l$, $p_{ji} = \mathrm{col}_l(N_{P'})$ $(1 \le l \le n)$ and $a_{jl} = 1/r$. Thus, $x_j = a_{j1}v_1 +\dots+ a_{jn}v_n = 1$ and $w_j = 1$.

Since $p_k \leftarrow p_{k1} \vee\dots\vee p_{kq} \in D$ and $\mathrm{row}_j(w) = p_j \in\{p_{k1},\dots, p_{kq}\}$, it becomes $u_k = 1$ (by the definition of θ_D in Def. 2.9(ii)).

By (i) and (ii), $p_k \in T_P(I)$ implies $u_k = 1$ $(1 \le k \le n)$, thereby $T_P(I) \subseteq J \cap B_P$

Hence: $J \cap B_P = T_P(I)$. □

Given a matrix $N_{P'} \in \mathbb{R}^{m\times n}$ and the initial vector v_0 of P', define $v_{k+1} = \theta_D(N_{P'} v_k[1\ldots n])$ $(k \geq 0)$. Then it holds that $v_{k+1} = v_k$ for some $k \geq 1$. When $v_{k+1} = v_k$, we write $v_k = FP(N_{P'} v_0[1\ldots n])$. It shows that $FP(N_{P'} v_0[1\ldots n])$ represents the least model of P.

Generally, given a d-program P', the value k of $v_k = FP(N_{P'} v_0[1\ldots n])$ is not greater than the value h of $v_h = FP(M_P v_o)$ in Section 2.2.

Example 2.3: For the d-program P' of Example 2.2, we have the submatrix $N_{P'} \in \mathbb{R}^{6\times 4}$ representing P' (right). Given the initial vector $v_0 = (0\ 0\ 0\ 1\ 0\ 0)^T$ of P', it becomes $v_1 = \theta_D(N_{P'} v_0[1\ldots 4]) = (0\ 0\ 1\ 1\ 0\ 0)^T$, $v_2 = \theta_D(N_{P'} v_1[1\ldots 4]) = (1\ 0\ 1\ 1\ 0\ 1)^T$, $v_3 = \theta_D(N_{P'} v_2[1\ldots 4]) = (1\ 0\ 1\ 1\ 0\ 1)^T = v_2$.

$$N_{P'} = \begin{array}{c} \\ \\ \\ \\ \\ \end{array}\begin{matrix} p & q & r & s \end{matrix}$$

$$N_{P'} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & ½ & ½ \end{pmatrix} \begin{matrix} p \\ q \\ r \\ s \\ t \\ u \end{matrix}$$

Then v_2 is a vector representing the least model of P', and $v_2[1\ldots 4]$ is a vector representing the least model $\{p, r, s\}$ of P. Note that the first element of v_i $(i = 2, 3)$ becomes 1 by Def. 2.9(ii).

By Theorem 2.3, we can replace the computation $u = \theta(M_{P'} v)$ in Step 3 of Algorithm 2.1 by $u = \theta_D(N_{P'} v[1\ldots n])$. In the column reduction method, the complexity of computing $N_{P'} v[1\ldots n]$ is $O(m\times n)$ and computing $\theta_D(.)$ is $O(m\times n)$. The number of times for iterating $N_{P'} v$ is at most $(m + 1)$ times. So the complexity of computing $u = \theta_D(N_{P'} v[1\ldots n])$ is $O((m + 1) \times (m \times n + m \times n)) = \mathbf{O(m^2\times n)}$. Comparing the complexity $O(m^3)$ of Step 3 of Algorithm 2.1, the column reduction reduces the complexity to $O(m^2\times n)$ as $n << m$ in general.

3 Normal Programs

In [8], normal programs are converted to disjunctive programs using the transformation by [13], and then encoded in matrices using third-order tensors. In this paper, we first transform normal programs to definite programs using the transformation in [12] and then encode them in matrices as in Section 2.

3.1 Computing stable models of a normal program

A *normal program* P is a finite set of *rules* of the form:

$$h \leftarrow b_1 \wedge \ldots \wedge b_k \wedge \neg b_{k+1} \wedge \ldots \wedge \neg b_m \quad (m \geq 0) \qquad (3)$$

where h and b_j are propositional variables. P is transformed to a definite program by rewriting the above rule as:

$$h \leftarrow b_1 \wedge \ldots \wedge b_k \wedge \overline{b_{k+1}} \wedge \ldots \wedge \overline{b_m} \quad (m \geq 0) \qquad (4)$$

where $\overline{b_i}$ is a new proposition associated with b_i. We call b_i a positive literal and $\overline{b_j}$ a negative literal.

Given a normal program P and an interpretation $I \subseteq B_P$, the transformed definite program is denoted by P^+, called a *positive form*. As Def. 2.6, we can transform P^+ to a d-program P'. Define $\overline{I} = \{\overline{p} \mid p \in B_P \setminus I\}$ and $I^+ = I \cup \overline{I}$.

Theorem 3.1 [12]: Let P be a normal program. Then, I is a stable model of P iff I^+ is the least model of $P^+ \cup \overline{I}$.

Definition 3.1: (program matrix for a normal program) Let P be a normal program with $B_P = \{p_1, \ldots, p_k\}$, and P^+ its positive form with $B_{P+} = \{p_1, \ldots, p_k, \overline{q_{n+1}}, \ldots, \overline{q_m}\}$.

Also, let P' be a d-program that is obtained from P^+ with $B_{P'} = \{p_1, ..., p_k, p_{k+1}, ..., p_n, \overline{q_{n+1}}, ..., \overline{q_m}\}$. We have $\{q_{n+1}, ..., q_m\} \subseteq \{p_1, ..., p_k\}$, and P' has n positive literals and $(m-n)$ negative literals. Then P' is represented by a matrix $M_{P'} \in \mathbb{R}^{m\times m}$ such that for each element a_{ij} $(1 \le i, j \le m)$ in $M_{P'}$:

1. $a_{ii} = 1$ for $n + 1 \le i \le m$
2. $a_{ij} = 0$ for $n + 1 \le i \le m$ and $1 \le j \le m$ such that $i \ne j$
3. Otherwise, a_{ij} $(1 \le i \le n; 1 \le j \le m)$ is encoded as in Def. 2.7.

By definition, negative literals are encoded in $M_{P'}$ in the same manner as facts. Intuitively, $a_{ii} = 1$ for $\overline{q_i}$ represents the rule $\overline{q_i} \leftarrow \overline{q_i}$ that means a "guess" for $\overline{q_i}$.

Definition 3.2: (initial matrix) Let P be a normal program and $B_P = \{p_1, ..., p_k\}$, P' its transformed d-program (via. P^+) and $B_{P'} = \{p_1, ..., p_k, p_{k+1}, ..., p_n, \overline{q_{n+1}}, ..., \overline{q_m}\}$. The *initial matrix* $M_o \in \mathbb{R}^{m\times h}$ $(1 \le h \le 2^{m-n})$ is defined as follows:

- Each row of M_o corresponds to each element of $B_{P'}$ in a way that $\text{row}_i(M_o) = p_i$ for $1 \le i \le n$ and $\text{row}_i(M_o) = \overline{q_i}$ for $n + 1 \le i \le m$.
- $a_{ij} = 1$ $(1 \le i \le n, 1 \le j \le h)$ iff a fact $p_i \leftarrow$ is in P; otherwise, $a_{ij} = 0$.
- $a_{ij} = 0$ $(n + 1 \le i \le m, 1 \le j \le h)$ iff a fact $q_i \leftarrow$ is in P; otherwise, there are two possibilities 0 and 1 for a_{ij}, so it is either 0 or 1.

Each column of M_o corresponds to a potential stable model.

Let P be a normal program and P' its transformed d-program. For the program matrix $M_{P'} \in \mathbb{R}^{m\times m}$ and the initial matrix $M_o \in \mathbb{R}^{m\times h}$. Define: $M_{k+1} = \theta(M_{P'} M_k)$ $(k \ge 0)$. It holds that $M_{k+1} = M_k$ for some $k \ge 0$. When $M_{k+1} = M_k$, we write $M_k = FP(M_{P'} M_o)$.

Suppose $M_k = FP(M_{P'} M_o)$ $(k \ge 1)$. Let $u = (a_1 \ldots a_n, a_{n+1} \ldots a_m)^T$ be a column vector in M_k such that $a_j = 1$ (resp. $a_j = 0$) $(n + 1 \le j \le m)$ iff $a_i = 0$ (resp. $a_i = 1$) with i such that $1 \le i \le n$, $\text{row}_j(M_o) = \overline{q_j}$ and $\text{row}_i(M_o) = p_i = q_j$. Then we have the next result.

Theorem 3.2: u is a column vector representing the interpretation I of P' iff $I \cap B_P$ is a stable model of P.

3.2 Algorithm for computing stable models

Based on Theorem 3.2, we have an algorithm for finding stable models of a normal program P (Fig. 2).

Example 3.1: Consider $P = \{ p \leftarrow q \wedge \neg r \wedge s,\ q \leftarrow \neg t \wedge q,\ q \leftarrow s,\ r \leftarrow \neg t,\ s \leftarrow,\ t \leftarrow \}$ with $B_P = \{p, q, r, s, t\}$. First, P is transformed to a positive form P^+ and a d-program P' as follows:

- $P^+ = \{ p \leftarrow q \wedge \overline{r} \wedge s,\ q \leftarrow \overline{t} \wedge q,\ q \leftarrow s,\ r \leftarrow \overline{t},\ s \leftarrow,\ t \leftarrow \}$
- $P' = Q \cup D$ where: $Q = \{ p \leftarrow q \wedge \overline{r} \wedge s,\ q_1 \leftarrow \overline{t} \wedge q,\ q_2 \leftarrow s,\ r \leftarrow \overline{t},\ s \leftarrow,\ t \leftarrow \}$

$$D = \{ q \leftarrow q_1 \vee q_2 \}$$

We have the representing matrix $M_{P'} \in \mathbb{R}^{9\times 9}$ and the initial matrix $M_o \in \mathbb{R}^{9\times 2}$:

$$M_{P'} = \begin{array}{c} \begin{matrix} p & q & r & s & t & q_1 & q_2 & \overline{r} & \overline{t} \end{matrix} \\ \begin{pmatrix} 0 & 1/3 & 0 & 1/3 & 0 & 0 & 0 & 1/3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1/2 & 0 & 0 & 0 & 0 & 0 & 0 & 1/2 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \begin{matrix} p \\ q \\ r \\ s \\ t \\ q_1 \\ q_2 \\ \overline{r} \\ \overline{t} \end{matrix}$$

Algorithm 3.1:

Input: a normal program P and the Herband base $B_P = \{p_1,\dots,p_k\}$

Output: A set of vectors representing stable models of P.

<table>
<tr>
<td>

Step 1: Transform a normal program P to d-program P' (via P^+) with $B_{P'} = \{p_1,\dots,p_k, p_{k+1},\dots, p_n, \overline{q_{n+1}},\dots,\overline{q_m}\}$
Step 2:
- Create the matrix $M_{P'} \in \mathbb{R}^{m\times m}$ representing a d-program P'.
- Create the initial matrix $M_o \in \mathbb{R}^{m\times h}$ of P'.
Step 3: *Compute a fixpoint $FP(M_P{\cdot}M_o)$*
$M := M_o$
$U := \theta(M_P{\cdot}M)$
while $U \neq M$ do
$M := U$;
$U := \theta(M_P{\cdot}M)$;
end do;

</td>
<td>

Step 4: *Find stable models of P*
result:={ };
for i from 1 to h do
$v := (a_1,\dots a_n, a_{n+1},\dots, a_m)^T$ is i^{th}-column of M
for j from $n+1$ to m do
$\overline{q_j}$:= row$_j(M)$;
for l from 1 to n do
if row$_l(M) = q_j$ then
if $a_l + a_j \neq 1$ then break;
end for; #l
if $l \leq n$ then break;
end for; #j
if $j \leq m$ then break;
else result := result $\cup$ $\{v\}$;
end for;
return result;

</td>
</tr>
</table>

Fig. 2. Algorithm 3.1

$$M_0 = \begin{matrix} p \\ q \\ r \\ s \\ t \\ q_1 \\ q_2 \\ \bar{r} \\ \bar{t} \end{matrix}\begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \quad M_1 = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 0 & 0 \\ 1 & 1 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \quad M_2 = \begin{pmatrix} 0 & 0 \\ 1 & 1 \\ 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 0 & 0 \\ 1 & 1 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \quad M_3 = \begin{pmatrix} 0 & 1 \\ 1 & 1 \\ 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 0 & 0 \\ 1 & 1 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$$

Then $M_1 = \theta(M_P{\cdot}M_o)$, $M_2 = \theta(M_P{\cdot}M_1)$, $M_3 = \theta(M_P{\cdot}M_2)$. $M_4 = \theta(M_P{\cdot}M_3) = M_3$ becomes the fixpoint. In this case, the column vector $u = (1\ 1\ 0\ 1\ 1\ 0\ 1\ 1\ 0)^T$ satisfies the condition $u[8] = 1$ iff $u[3] = 0$ where row$_8(u) = \bar{r}$ and row$_3(u) = r$. The vector u represents the set $\{p, q, s, t, q_2, \bar{r}\}$ and $\{p, q, s, t, q_2, \bar{r}\} \cap B_P = \{p, q, s, t\}$ is the stable model of P.

In Algorithm 3.1, the complexity of $M_P{\cdot}M$ is $\mathbf{O}(\boldsymbol{m}^2\times\boldsymbol{h})$. The number of times for iterating $M_P{\cdot}M$ is at most $(\boldsymbol{m} + 1)$ times. Thus, the complexity of Step 3 is $\mathbf{O}((\boldsymbol{m} + \mathbf{1})\times\boldsymbol{m}^2\times\boldsymbol{h}) = \mathbf{O}(\boldsymbol{m}^3\times\boldsymbol{h})$ in the worst case. In this method, we encode a normal program into a program matrix, while [8] encodes a normal program into a 3rd-order tensor. Since the number of slices in a 3rd-order tensor increases exponentially in general, the current method would have a computational advantage over [8].

3.3 Column reduction

We apply the submatrix technique of Def.2.8 to program matrices for normal programs. That is, instead of considering a program matrix $M_{P'} \in \mathbb{R}^{m\times m}$ of Def.3.1, we consider a submatrix $N_{P'} \in \mathbb{R}^{m\times r}$ where $r = k + (m - n)$. Note that, $r << m$ in general.

Definition 3.4: (**initial vector**) Let P be a normal program with $B_P = \{p_1,\ldots,p_k\}$, and P^+ its positive form with $B_{P+} = \{p_1,\ldots,p_k, \overline{q_{n+1}},\ldots \overline{q_m}\}$ where $\{q_{n+1},\ldots,q_m\} \subseteq \{p_1,\ldots,p_k\}$. Then the set of initial vectors of a positive form P^+ is defined as follows:

Let $v_1 \in \mathbb{R}^k$ be a vector representing facts in P where $\text{row}_j(v_1) = p_j$.

Consider $A \subseteq \mathbb{R}^{r-k}$ where $A = \{(1\ 0\ \ldots\ 0)^T, (1\ 1\ \ldots\ 0)^T, \ldots., (1\ 1\ \ldots\ 1)^T\}$ with $\text{card}(A) = 2^{r-k}$, and $B = \{v \in A \mid \exists\, i\ (1 \le i \le r-k)$ s.t. $v[i] = 1$ and $\exists\, j\ (1 \le j \le k)$ s.t. $v_1[j] = 1$ and $\text{row}_j(v_1) = p$ iff $\text{row}_i(v) = \overline{p}$ where v_1 represents facts in $P\}$. Put $v_2 \in \mathbb{R}^{r-k}$ s.t. $v_2 \in A \backslash B$. The set of initial vectors V of P^+ is:

$$V = \left\{ v \in \mathbb{R}^r \mid v = \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}, v_2 \in A \setminus B \right\} \quad \text{and} \quad h = \text{card}(V)$$

Intuitively, the set of initial vectors V represents facts in P together with possible negative literals in P'. By Theorem 2.3, if $v \in \mathbb{R}^r$ is a vector representing an interpretation I of P^+, and $u = \theta_D(N_{P'}\ v) \in \mathbb{R}^m$, then u is a vector representing an interpretation J of P' and $J \cap B_{P+} = T_{P+}(I)$.

For this reason, in Step 3 of Algorithm 3.1, we can replace the computation of a fixpoint $FP(M_P \cdot M_o)$ by the computation of a fixpoint $FP(N_P \cdot u_o[1\ldots r])$ with the initial vector $u_o \in V$. In the column reduction method, the complexity of computing $N_P \cdot u_o[1\ldots n']$ is $O(m \times r)$ and computing $\theta_D(.)$ is $O(m \times r)$. Since the number of times for iterating $N_P \cdot u_o[1\ldots r]$ is at most $(m + 1)$ times and $|V| = h$, the complexity of step 3 of this algorithm is $O((m + 1)\times(m\times r + m\times r)\times h)$ = $\mathbf{O(m^2 \times r \times h)}$. Comparing the complexity $O(m^3\times h)$ of Step 3 of Algorithm 3.1, the column reduction reduces the complexity to $O(m^2\times r\times h)$ by $r << m$ in general.

Example 3.2: Consider a normal program P and a d-program P' of Example 3.1.

We have the submatrix $N_{P'} \in \mathbb{R}^{9\times 7}$ representing P':

- $v_1 \in \mathbb{R}^5$ represents the facts in P, $v_1 = (0\ 0\ 0\ 1\ 1)^T$
- $A = \{(0\ \ 0)^T, (1\ \ 0)^T, (0\ \ 1)^T, (1\ \ 1)^T\}$ with $\text{card(A)} = 2^2 = 4$
- $B = \{(0\ \ 1)^T, (1\ \ 1)^T\}$

  $v_2 \in A \setminus B = \{(0\ \ 0)^T, (1\ \ 0)^T\}$

  $V = \{(0\ 0\ 0\ 1\ 1\ 0\ 0)^T, (0\ 0\ 0\ 1\ 1\ 1\ 0)^T\}$

  with $h = \text{card(V)} = 2$

$$N_{P'} = \begin{array}{c} \\ \end{array}\begin{array}{ccccccc} p & q & r & s & t & \overline{r} & \overline{t} \end{array}$$

$$N_{P'} = \begin{pmatrix} 0 & 1/3 & 0 & 1/3 & 0 & 1/3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & ½ & 0 & 0 & 0 & 0 & ½ \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} \begin{array}{c} p \\ q \\ r \\ s \\ t \\ \overline{r} \\ \overline{t} \\ q_1 \\ q_2 \end{array}$$

Compute a fixpoint $FP(N_{P'}\, u_o)$ $(u_o \in V)$:

(i) For $u_o = (0\ 0\ 0\ 1\ 1\ 0\ 0)^T$:

$u_1 = \theta_D(N_P \cdot u_o) = (0\ 1\ 0\ 1\ 1\ 0\ 0\ 0\ 1)^T$

$u_2 = \theta_D(N_P \cdot u_1[1\ldots 7]) = (0\ 1\ 0\ 1\ 1\ 0\ 0\ 0\ 1)^T = u_1$.

$\text{row}_3(u_1) = r$ and $\text{row}_6(u_1) = \overline{r}$ then $u_1[3] + u_1[6] = 0$, so u_1 does not represent a stable model of P.

(ii) For $u_o = (0\ 0\ 0\ 1\ 1\ 1\ 0)^T$: $u_1 = \theta_D(N_P \cdot u_o) = (0\ 1\ 0\ 1\ 1\ 1\ 0\ 0\ 1)^T$, $u_2 = \theta_D(N_P \cdot u_1[1\ldots 7]) = (1\ 1\ 0\ 1\ 1\ 1\ 0\ 0\ 1)^T$, $u_3 = \theta_D(N_P \cdot u_2[1\ldots 7]) = (1\ 1\ 0\ 1\ 1\ 1\ 0\ 0\ 1)^T = u_2$.

$\text{row}_3(u_2) = r$ and $\text{row}_6(u_2) = \overline{r}$ then $u_2[3] + u_2[6] = 1$

$\text{row}_5(u_2) = t$ and $\text{row}_7(u_2) = \overline{t}$ then $u_2[5] + u_2[7] = 1$

u_2 represents the set $\{p, q, s, t, \overline{r}, q_2\}$ and $\{p, q, s, t, \overline{r}\} \cap B_P = \{p, q, s, t\}$ is the stable model of P.

4 Experimental Results

In this section, we compare runtime for finding the least model of a definite program and the set of stable models of a normal program by the following three algorithms: (i) computing the fixpoint of the T_P-operator; (ii) matrix computation; (iii) column reduction computation. The testing is run on a GPU that has configuration as follows:

- Operating system: Linux Ubuntu 16.04 LTS 64bit.
- CPU: Intel® Core™ i7-6800K <3.4 GHz /14nm / Cores = 6 / Threads = 12 / Cache = 15 MB>, Memory 32GB, DDR-2400
- GPU: GeForce GTX1070TI GDDR5 8GB
- Implementation language: Maple 2017, 64 bit [14].

GPU speeds up the computing and further speed-up would be gained by using CUDA package in Maple.

4.1 Testing on definite programs

In this testing, given the size $n = |B_P|$ of the Herband base B_P and the number $m = |P|$ of rules in P, rules are created randomly as in Table 1:

Table 1. Proportion of rules in P based on the number of propositional variables in their bodies

Number of elements in body	0	1	2	3	4	5	6	7	8
Number of rules (proportion)	x, where x < n/3	4%	4%	10%	40%	35%	4%	2%	~1%

Based on (n, m), generate a definite program P randomly. After that, we transform P to a d-program $P' = Q \cup D$ where Q is an SD-program and D is a set of d-rules. The program P' will have n' variables and m' rules. Table 2 shows the results of testing Algorithm 2.1 on P' and computation by the T_P-operator [9] for constructing the least model of a program P. There are two important steps in Algorithm 2.1: Step 2 for creating a program matrix $M_{P'}$ to represent a definite program P', and step 3 for computing the fixpoint. We compare three cases—**(Case 1):** Use the T_P-operator on a program P; **(Case 2):** naive computation on a program P' by Algorithm 2.1; **(Case 3):** computation by column reduction on a program P' as Section 2.5.

Table 2. Results of testing on definite programs

n	m	T_P - operator (sec.)	n'	m'	Matrix Fixpoint / All (sec.)	Column reduction Fixpoint/All (sec.)
20	200	0.066	173	173	0.277 / 0.288	0.009 / 0.018
20	400	0.07	396	396	0.225 / 0.238	0.019 / 0.034
20	8,000	0.628	6,047	6,047	6.491 / 6.709	0.103 / 0.251
50	2,500	0.499	2,430	2,430	3.797 / 3.925	0.114 / 0.205
50	12,500	1.952	8,858	8,858	8.709 / 9.023	0.377 / 0.812
100	5,000	2.056	4,707	4,707	13.23 / 13.326	0.661 / 0.978
100	10,000	1.935	7,000	7,000	11.166 / 11.479	0.79 / 1.27
200	400	0.037	451	428	0.059 / 0.073	0.012 / 0.06
200	20,000	5.846	16,052	16,052	25.093 / 25.945	3.973 / 6.73

"All" means the time for creating a program matrix and computing the fixpoint.

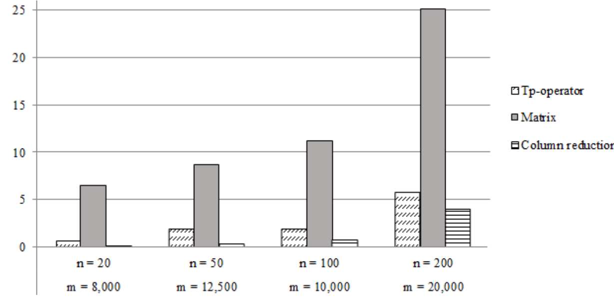

Fig. 3. Results of testing about fixpoint computing on definite programs

Note that in some cases the number of rules in *P'* is smaller than those of *P* (*m'* < *m*). This is because we eliminate every rule *r* in *P'* such that *head*(*r*) = *p* and the fact *p*← is in *P*. Comparison of fixpoint computation is shown in Fig. 3. We can observe the following facts: The naive method is slower than the T_P-operator while the column reduction technique significantly reduces runtime and is fastest among three algorithms. The runtime efficiency of column reduction comes from the fact *n* << *n'*.

4.2 Testing on normal programs

In this testing, given the size $n = |B_P|$ of the Herband base B_P and the number $m = |P|$ of rules in *P*, rules are created randomly as before (Table 1).

Based on (*n, m*), generate a normal program *P* randomly. After that, we transform *P* to a d-program *P'* with *k* negative literals in a program *P*.

Table 3 is the results of testing Algorithm 3.1 on *P'* and computation by the T_P-operator (using Theorem 3.1) on *P* [12] for computing the stable models of a program *P*. There are three important steps in Algorithm 3.1: Step 2 for creating a program matrix for representing a d-program *P'*, Step 3 for computing the fixpoint and Step 4 for finding the set of vectors which represent stable models. We compare three cases—**(Case 1):** Computation by the T_P-operator (using Theorem 3.1) for computing stable models of on *P*; **(Case 2):** the naive computation on a program *P'* by Algorithm 3.1; **(Case 3):** the column reduction computation on a program *P'* as presented in Section 3.2 of Step 3. Comparison of fixpoint computation is shown in Fig. 4.

Table 3. Results of testing on normal programs

n	*m*	*k*	T_P-operator (sec.)	Matrix Fixpoint / All (sec.)	Column reduction Fixpoint/All (sec.)
20	400	8	2.432	19.603 / 19.714	3.338 / 3.362
20	8,000	6	5.531	12.368 / 12.696	4.502 / 4.603
50	100	8	0.221	1.155 / 1.224	0.278 / 0.291
50	2,500	8	36.574	37.863 / 38.463	29.582 / 29.786
50	12,500	7	49.485	30.819 / 32.00	48.883 / 49.32
100	5,000	8	103.586	31.68 / 32.338	69.579 / 69.851
100	10,000	8	264.547	84.899 / 87.142	192.981 / 194.003
200	400	6	0.429	1.928 / 2.021	1.222 / 1.342
200	13,300	6	185.778	48.185 / 49.185	124.119 / 126.255

"All" means the time for creating a program matrix and computing the fixpoint.

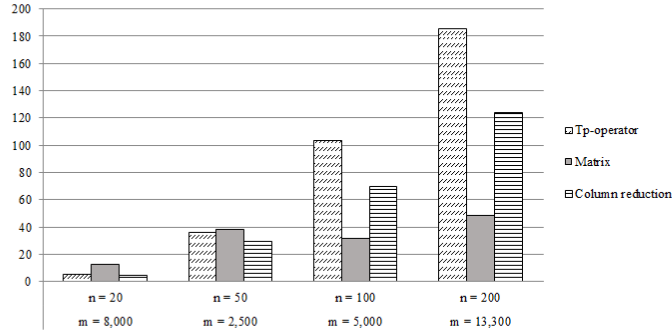

Fig. 4. Results of testing about fixpoint computing on normal programs

By the table we can observe the following fact: matrix computation is effective when the size of n is large (n =100 or 200). Computation by column reduction is faster than computation by the T_P-operator, while it is slower than the naive method in case of n = 100 or 200. To see the effect of computation by column reduction, we would need further environment that realizes efficient computation of matrices.

5 Conclusion

In this paper, we proposed methods for representing logic programming based on linear algebra. We develop new algorithms for computing logic programming semantics in linear algebra and the improvement methods for speeding-up those algorithms.

The results of testing show that the computation by column reduction is fastest in computing least models, while the naive matrix computation on a d-program is often better than column reduction in computing stable models. It is known that the least model of a definite program is computed in O(N) [15] where N is the size (number of literals) of a program. Since the column reduction computation takes $O(m^2 \times n)$ time, it would be effective when $m^2 \times n < N$, i.e., the size of a program is large with a relatively small number of atoms. For computation of stable models of a normal program, although the size of the program matrix and the initial matrix are large, they have many zero elements. We can improve the method for representing matrices in sparse forms which also brings storage advantages with a good matrix library. Introducing partial evaluation [18] would also help to reduce runtime. We need further optimization and comparison with existing answer set solvers such as clasp [16], DLV [17].

The performance of our linear algebraic implementation heavily depends on the manipulation of matrices. We have used Maple for implementation, but our methods can be realized by other computer languages and architectures. It is now expected that more powerful platforms are developed for linear algebraic computation in the near future. Our methods would have the merits when such advanced technologies become available. Yet, linear algebraic computation for logic programming has just started, so there will be a lot of rooms for improvement and optimization.

Acknowledgment

This work was supported by JSPS KAKENHI Grant Numbers JP17H00763 and JP18H03288.

References

1. Saraswat, V.: Reasoning 2.0 or machine learning and logic–the beginnings of a new computer science. Data Science Day, Kista Sweden (2016)
2. Sato, T.: A linear algebraic approach to Datalog evaluation. Theory and Practice of Logic Programming 17(3), 244–265 (2017)
3. Yang, B., Yih, W., He, X., Gao, J., Deng, L.: Embedding entities and relations for learning and inference in knowledge bases, In: In: Third Int. Conf. Learning Representations (ICLR 2015), San Diego, USA (2015)
4. Grefenstette, E.: Towards a Formal Distributional Semantics: Simulating Logical Calculi with Tensors, In: Proceedings of Second Joint Conference on Lexical and Computational Semantics (*SEM), Vol. 1, pp. 1–10, Atlanta, USA (2013).
5. Coecke, B., Sadrzadeh, M., Clarky, S.: Mathematical Foundations for a Compositional Distributional Model of Meaning, Linguistic Analysis, 36, pp. 345–384 (2011)
6. Serafini, L., Garcez, A.: Learning and Reasoning with Logic Tensor Networks, In: 15th Int. Conf. of the Italian Association for Artificial Intelligence (AI*IA 2017), LNAI 10037, pp. 334-348, Springer, Genoa, Italy (2016)
7. Serafini, L., Donadello, I., Garcez, A.: Learning and Reasoning with Logic Tensor Networks: Theory and application to semantic image interpretation, In: Proc. of 32nd ACM SIGAPP Symposium On Applied Computing (SAC 2017), pp. 125-130, Marrakech, Morocco (2017)
8. Sakama, C., Inoue, K., Sato, T.: Linear Algebraic Characterization of Logic Programs, In: 10th International conference on Knowledge Science, Engineering and Management (KSEM 2017), LNAI 10412, pp.530-533, Springer, Melbourne, Australia (2017).
9. van Emden, M. H., Kowalski, R. A.: The semantics of predicate logic as a programming language. Journal of the ACM 23(4), 733–742 (1976).
10. Kolda, T., Bader, B.: Tensor Decompositions and Applications, *SIAM Review* 51(3), 455-500 (2009).
11. Lin, F.: From satisfiability to linear algebra. Invited talk, 26th Australian Joint Conference on Artificial Intelligence (2013)
12. J.J. Alferes, J.A. Leite, L.M. Pereira, H. Przymusinska, T. Przymusinski: Dynamic updates of non-monotonic knowledge bases, *Journal of Logic Programming* 45(1-3), 43-70 (2000).
13. Fernandez, J. A., Lobo, J., Minker, J., Subrahmanian, V. S.: Disjunctive LP + integrity constraints = stable model semantics, *AMAI* 8(3-4), 449–474 (1993).
14. Maple: https://www.maplesoft.com/support/install/maple2017_install.html
15. Dowling, W. F., Gallier, J. H.: Linear-time algorithms for testing the satisfiability of propositional Horn formulae, *Journal of Logic Programming* 1(3), 267-284 (1984).
16. clasp: https://potassco.org/clasp/
17. DLV system: http://www.dlvsystem.com/dlv/
18. Sakama, C., Nguyen, H.D., Sato, T., Inoue, K.: Partial Evaluation of Logic Programs in Vector Space, 11th Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP 2018), Oxford, UK, July 2018.