Partial Evaluation of Logic Programs in Vector Spaces

Chiaki Sakama, Hien Nguyen, Taisuke Sato and Katsumi Inoue

July 11, 2018

Partial Evaluation of Logic Programs in Vector Spaces

CHIAKI SAKAMA
Wakayama University, Japan
(*e-mail:* `sakama@sys.wakayama-u.ac.jp`)

HIEN D. NGUYEN
University of Information Technology, VNU-HCM, Vietnam
(*e-mail:* `hiennd@uit.edu.vn`)

TAISUKE SATO
AI Research Center AIST, Japan
(*e-mail:* `satou.taisuke@aist.go.jp`)

KATSUMI INOUE
National Institute of Informatics (NII), Japan
(*e-mail:* `inoue@nii.ac.jp`)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

In this paper, we introduce methods of encoding propositional logic programs in vector spaces. Interpretations are represented by vectors and programs are represented by matrices. The least model of a definite program is computed by multiplying an interpretation vector and a program matrix. To optimize computation in vector spaces, we provide a method of partial evaluation of programs using linear algebra. Partial evaluation is done by unfolding rules in a program, and it is realized in a vector space by multiplying program matrices. We perform experiments using randomly generated programs and show that partial evaluation has potential for realizing efficient computation in huge scale of programs.

1 Introduction

One of the challenging topics in AI is to reason with huge scale of knowledge bases. Linear algebraic computation has potential to make symbolic reasoning scalable to real-life datasets, and several studies aim at integrating linear algebraic computation and symbolic computation. For instance, Grefenstette (2013) introduces tensor-based predicate calculus that realizes logical operations. Yang, *et al*. (2015) introduce a method of mining Horn clauses from relational facts represented in a vector space. Serafini and Garcez (2016) introduce logic tensor networks that integrate logical deductive reasoning and data-driven relational learning. Sato (2017a) formalizes Tarskian semantics of first-order logic in vector spaces, and (Sato 2017b) shows that tensorization realizes efficient computation of Datalog. Lin (2013) introduces linear algebraic computation of SAT for clausal theories.

To realize linear algebraic computation of logic programming, Sakama *et al*. (2017) introduce encodings of Horn, disjunctive and normal logic programs in vector spaces. They show that least models of Horn programs, minimal models of disjunctive programs, and stable models of normal programs are computed by algebraic manipulation of third-order tensors. The study builds a new theory of logic programming, while implementation and evaluation are left open.

In this paper, we first reformulate the framework of (Sakama *et al*. 2017) and present an algorithm for computing least models of definite programs in vector spaces. We next introduce two optimization techniques for computing: the first one is based on column reduction of matrices, and the second one is based on *partial evaluation*. We perform experimental testing and compare algorithms for computing fixpoints of definite programs. The rest of this paper is organized as follows. Section 2 reviews basic notions and Section 3 provides linear algebraic characterization of logic programming. Section 4 presents partial evaluation of logic programs in vector spaces. Section 5 provides experimental results and Section 6 summarizes the paper. Due to space limit, we omit proofs of propositions and theorems.

2 Preliminaries

We consider a language $\mathscr{L}$ that contains a finite set of propositional variables. Given a logic program P, the set of all propositional variables appearing in P is called the *Herbrand base* of P (written B_P). A *definite program* is a finite set of *rules* of the form:

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

where h and b_i are propositional variables in $\mathscr{L}$. A rule r is called a *d-rule* if r is the form:

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

where h and b_i are propositional variables in $\mathscr{L}$. 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: $h \leftarrow b_1$, ..., $h \leftarrow b_m$, so a d-program is considered a definite program.[1] For each rule r of the form (1) or (2), define $head(r) = h$ and $body(r) = \{b_1, \ldots, b_m\}$.[2] A rule is called 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. A mapping $T_P : 2^{B_P} \to 2^{B_P}$ (called a T_P*-operator*) is defined as:

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

The *powers* of T_P are defined as: $T_P^{k+1}(I) = T_P(T_P^k(I))$ $(k \geq 0)$ and $T_P^0(I) = I$. 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 (van Emden & Kowalski 1976).

3 Logic Programming in Linear Algebra

3.1 SD programs

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

[1] The notion of d-programs is useful when we consider a program such that each atom is defined by a single rule in Section 3.

[2] We assume $b_i \neq b_j$ if $i \neq j$.

Definition 1 (*SD program*)
A definite program P is called *singly defined* (*SD program*, for short) if $head(r_1) \neq head(r_2)$ for any two rules r_1 and r_2 ($r_1 \neq r_2$) in P.

Interpretations and programs are represented in a vector space as follows.

Definition 2 (*interpretation vector (Sakama et al. 2017)*)
Let P be a definite program and $B_P = \{p_1, \dots, p_n\}$. Then an interpretation $I \subseteq B_P$ is represented by a vector $\boldsymbol{v} = (a_1, \dots, a_n)^{\mathsf{T}}$ where each element a_i $(1 \leq i \leq n)$ represents the truth value of the proposition p_i such that $a_i = 1$ if $p_i \in I$; otherwise, $a_i = 0$. We write $\mathsf{row}_i(\boldsymbol{v}) = p_i$. Given $\boldsymbol{v} = (a_1, \dots, a_n)^{\mathsf{T}} \in \mathbb{R}^n$, define $\boldsymbol{v}[i] = a_i$ $(1 \leq i \leq n)$ and $\boldsymbol{v}[1 \dots k] = (a_1, \dots, a_k)^{\mathsf{T}} \in \mathbb{R}^k$ $(k \leq n)$.

Definition 3 (*matrix representation of SD programs*)
Let P be an SD program and $B_P = \{p_1, \dots, p_n\}$. Then P is represented by a matrix $\boldsymbol{M}_P \in \mathbb{R}^{n \times n}$ such that for each element a_{ij} $(1 \leq i, j \leq n)$ in $\boldsymbol{M}_P$,

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

$\boldsymbol{M}_P$ is called a *program matrix*. We write $\mathsf{row}_i(\boldsymbol{M}_P) = p_i$ and $\mathsf{col}_j(\boldsymbol{M}_P) = p_j$ $(1 \leq i, j \leq n)$.

In $\boldsymbol{M}_P$ the i-th row corresponds to the atom p_i appearing in the head of a rule, and the j-th column corresponds to the atom p_j appearing in the body of a rule. On the other hand, every fact $p_i \leftarrow$ in P is represented as a tautology $p_i \leftarrow p_i$ in $\boldsymbol{M}_P$.

Example 1
Consider $P = \{\, p \leftarrow q,\ \ q \leftarrow p \wedge r,\ \ r \leftarrow s,\ \ s \leftarrow \}$ with $B_P = \{\, p, q, r, s \,\}$. Then $\boldsymbol{M}_P$ becomes

$$\begin{array}{c} \\ p \\ q \\ r \\ s \end{array} \begin{array}{c} \begin{array}{cccc} p & q & r & s \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

where $\mathsf{row}_1(\boldsymbol{M}_P) = p$ and $\mathsf{col}_2(\boldsymbol{M}_P) = q$.

Definition 4 (*initial vector*)
Let P be a definite program and $B_P = \{p_1, \dots, p_n\}$. Then the *initial vector* of P is an interpretation vector $\boldsymbol{v}_0 = (a_1, \dots, a_n)^{\mathsf{T}}$ such that $a_i = 1$ $(1 \leq i \leq n)$ if $\mathsf{row}_i(\boldsymbol{v}_0) = p_i$ and a fact $p_i \leftarrow$ is in P; otherwise, $a_i = 0$.

Definition 5 (*θ-thresholding*)
Given a vector $\boldsymbol{v} = (a_1, \dots, a_n)^{\mathsf{T}}$, define $\theta(\boldsymbol{v}) = (a'_1, \dots, a'_n)^{\mathsf{T}}$ where $a'_i = 1$ $(1 \leq i \leq n)$ if $a_i \geq 1$; otherwise, $a'_i = 0$.[3] We call $\theta(\boldsymbol{v})$ the *θ-thresholding* of $\boldsymbol{v}$.

Given a program matrix $\boldsymbol{M}_P \in \mathbb{R}^{n \times n}$ and an initial vector $\boldsymbol{v}_0 \in \mathbb{R}^n$, define

$$\boldsymbol{v}_1 = \theta(\boldsymbol{M}_P \boldsymbol{v}_0) \quad \text{and} \quad \boldsymbol{v}_{k+1} = \theta(\boldsymbol{M}_P \boldsymbol{v}_k) \quad (k \geq 1)$$

It is shown that $\boldsymbol{v}_{k+1} = \boldsymbol{v}_k$ for some $k \geq 1$. When $\boldsymbol{v}_{k+1} = \boldsymbol{v}_k$, we write $\boldsymbol{v}_k = \mathsf{FP}(\boldsymbol{M}_P \boldsymbol{v}_0)$.

[3] a_i can be greater than 1 only later when d-rules come into play.

Theorem 1
Let P be an SD program and $\boldsymbol{M}_P \in \mathbb{R}^{n\times n}$ its program matrix. Then $\boldsymbol{m} \in \mathbb{R}^n$ is a vector representing the least model of P iff $\boldsymbol{m} = \mathsf{FP}(\boldsymbol{M}_P\boldsymbol{v}_0)$ where $\boldsymbol{v}_0$ is the initial vector of P.

Example 2
Consider the program P of Example 1 and its program matrix $\boldsymbol{M}_P$. The initial vector of P is $\boldsymbol{v}_0 = (0,0,0,1)^\mathsf{T}$. Then

$$\boldsymbol{M}_P\boldsymbol{v}_0 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

and $\boldsymbol{v}_1 = \theta(\boldsymbol{M}_P\boldsymbol{v}_0) = (0,0,1,1)^\mathsf{T}$. Next,

$$\boldsymbol{M}_P\boldsymbol{v}_1 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1/2 \\ 1 \\ 1 \end{pmatrix}$$

and $\boldsymbol{v}_2 = \theta(\boldsymbol{M}_P\boldsymbol{v}_1) = \boldsymbol{v}_1$. Hence, $\boldsymbol{v}_2 = (0,0,1,1)^\mathsf{T}$ represents the least model $\{r,s\}$ of P.

Remark: The current study is different from the previous work (Sakama *et al.* 2017) in matrix representation of programs as follows.

- In (Sakama *et al.* 2017) a fact is represented as a rule "$p_i \leftarrow \top$" and is encoded in a matrix by $a_{ij} = 1$ where $\mathsf{row}_i(\boldsymbol{M}_P) = p_i$ and $\mathsf{col}_j(\boldsymbol{M}_P) = \top$. Different from the current study, the previous study sets the empty set as the initial vector and computes fixpoints. In this study, we start with the initial vector representing facts, instead of representing facts as rules in $\boldsymbol{M}_P$. This has the effect of increasing zero elements in matrices and reducing the number of required iterations in fixpoint computation. Representing matrices in sparse forms also brings storage advantages with a good matrix library.
- In (Sakama *et al.* 2017) a constraint is represented as a rule "$\perp \leftarrow p_{j_1} \wedge \cdots \wedge p_{j_m}$" and is encoded in a matrix by $a_{ij_k} = \frac{1}{m}$ $(1 \le k \le m)$ where $\mathsf{row}_i(\boldsymbol{M}_P) = \perp$ and $\mathsf{col}_{j_k}(\boldsymbol{M}_P) = p_{j_k}$. In the current study, we do not include constraints in a program as it causes a problem in partial evaluation. Still, we can handle constraints separately from a program as follows. Given a program P and constraints C, encode them by matrices $\boldsymbol{M}_P \in \mathbb{R}^{n\times n}$ and $\boldsymbol{M}_C \in \mathbb{R}^{(n+1)\times n}$, respectively, where $\boldsymbol{M}_C$ has the element $\perp$ in its row. After computing the fixpoint $\boldsymbol{v}_k = \mathsf{FP}(\boldsymbol{M}_P\boldsymbol{v}_0) \in \mathbb{R}^n$ as in Theorem 1, compute $\boldsymbol{M}_C\boldsymbol{v}_k \in \mathbb{R}^{n+1}$. If $\mathsf{row}_i(\boldsymbol{v}_k) = \perp$ and $\boldsymbol{v}_k[i] = a_i = 1$, then $P \cup C$ is inconsistent; otherwise, $\boldsymbol{v}_k$ represents the least model of $P \cup C$.

3.2 Non-SD programs

When a definite program P contains two rules: $r_1 : h \leftarrow b_1 \wedge \cdots \wedge b_m$ and $r_2 : h \leftarrow b_1 \wedge \cdots \wedge b_n$, P is transformed to a d-program $P^\delta = (P \setminus \{r_1, r_2\}) \cup \{r_1', r_2', d_1\}$ where $r_1' : h_1 \leftarrow b_1 \wedge \cdots \wedge b_m$, $r_2' : h_2 \leftarrow b_1 \wedge \cdots \wedge b_n$ 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 6 (*transformation*)
Let P be a definite program and B_P its Herbrand base. For each $p \in B_P$, put $P_p = \{ r \mid r \in P \text{ and } head(r) = p \}$ and $R_p = \{ r \mid r \in P_p \text{ and } |P_p| > 1 \}$. Then define $S_p = \{ p_i \leftarrow body(r) \mid r \in R_p \text{ and } i = 1, \ldots, k \text{ where } k = |R_p| \}$ and $D_p = \{ p \leftarrow p_1 \vee \cdots \vee p_k \mid p_i \leftarrow body(r) \text{ is in } S_p \}$ where p_i is a new propositional variable such that $p_i \notin B_P$ and $p_i \neq p_j$ if $i \neq j$. Then, build a d-program

$$P^{\delta} = (P \setminus \bigcup_{p \in B_P} R_p) \cup \bigcup_{p \in B_P} (S_p \cup D_p)$$

where $Q = (P \setminus \bigcup_{p \in B_P} R_p) \cup \bigcup_{p \in B_P} S_p$ is an SD program and $D = \bigcup_{p \in B_P} D_p$ is a set of d-rules.

P^{δ} introduces additional propositional variables and $B_P \subseteq B_{P^{\delta}}$ holds. By definition, the next result holds.

Proposition 1
Let P be a definite program and P^{δ} its transformed d-program. Suppose that P and P^{δ} have the least models M and M', respectively. Then $M = M' \cap B_P$ holds.

In this way, any definite program P is transformed to a semantically equivalent d-program $P^{\delta} = Q \cup D$ where Q is an SD program and D is a set of d-rules. A d-program is represented by a matrix as follows.

Definition 7 (*program matrix for d-programs*)
Let P^{δ} be a d-program such that $P^{\delta} = Q \cup D$ where Q is an SD program and D is a set of d-rules, and $B_{P^{\delta}} = \{p_1, \ldots, p_m\}$ the Herbrand base of P^{δ}. Then P^{δ} is represented by a matrix $\boldsymbol{M}_{P^{\delta}} \in \mathbb{R}^{m \times m}$ such that for each element a_{ij} $(1 \leq i, j \leq m)$ in $\boldsymbol{M}_{P^{\delta}}$,

1. $a_{ij_k} = 1$ $(1 \leq k \leq l;\ 1 \leq i, j_k \leq m)$ if $p_i \leftarrow p_{j_1} \vee \cdots \vee p_{j_l}$ is in D;
2. otherwise, every rule in Q is encoded as in Def. 3.

Given a program matrix $\boldsymbol{M}_{P^{\delta}}$ and the initial vector $\boldsymbol{v}_0$ representing facts in P^{δ}, the fixpoint $\boldsymbol{v}_k = \mathsf{FP}(\boldsymbol{M}_{P^{\delta}} \boldsymbol{v}_0)$ $(k \geq 1)$ is computed as before. The fixpoint represents the least model of P^{δ}.

Theorem 2
Let P^{δ} be a d-program and $\boldsymbol{M}_{P^{\delta}} \in \mathbb{R}^{m \times m}$ its program matrix. Then $\boldsymbol{m} \in \mathbb{R}^m$ is a vector representing the least model of P^{δ} iff $\boldsymbol{m} = \mathsf{FP}(\boldsymbol{M}_{P^{\delta}} \boldsymbol{v}_0)$ where $\boldsymbol{v}_0$ is the initial vector of P^{δ}.

By Proposition 1 and Theorem 2, we can compute the least model of any definite program.

Example 3
Consider the program $P = \{ p \leftarrow q,\ \ q \leftarrow p \wedge r,\ \ q \leftarrow s,\ \ s \leftarrow \}$. As P is a non-SD program, it is transformed to a d-program $P^{\delta} = \{ p \leftarrow q,\ \ t \leftarrow p \wedge r,\ \ u \leftarrow s,\ \ s \leftarrow,\ \ q \leftarrow t \vee u \}$ where t and u are new propositional variables. Then $\boldsymbol{M}_{P^{\delta}} \in \mathbb{R}^{6 \times 6}$ becomes the matrix (right). The initial vector of P^{δ} is $\boldsymbol{v}_0 = (0,0,0,1,0,0)^{\mathsf{T}}$. Then, $\boldsymbol{v}_1 = \theta(\boldsymbol{M}_{P^{\delta}} \boldsymbol{v}_0) = (0,0,0,1,0,1)^{\mathsf{T}}$, $\boldsymbol{v}_2 = \theta(\boldsymbol{M}_{P^{\delta}} \boldsymbol{v}_1) = (0,1,0,1,0,1)^{\mathsf{T}}$, $\boldsymbol{v}_3 = \theta(\boldsymbol{M}_{P^{\delta}} \boldsymbol{v}_2) = (1,1,0,1,0,1)^{\mathsf{T}}$, and $\boldsymbol{v}_4 = \theta(\boldsymbol{M}_{P^{\delta}} \boldsymbol{v}_3) = \boldsymbol{v}_3$. Then $\boldsymbol{m} = \mathsf{FP}(\boldsymbol{M}_{P^{\delta}} \boldsymbol{v}_0) = (1,1,0,1,0,1)^{\mathsf{T}}$ represents the least model $\{p,q,s,u\}$ of P^{δ}, hence $\{p,q,s,u\} \cap B_P = \{p,q,s\}$ is the least model of P.

$$\begin{array}{c|cccccc} & p & q & r & s & t & u \\ \hline p & 0 & 1 & 0 & 0 & 0 & 0 \\ q & 0 & 0 & 0 & 0 & 1 & 1 \\ r & 0 & 0 & 0 & 0 & 0 & 0 \\ s & 0 & 0 & 0 & 1 & 0 & 0 \\ t & 1/2 & 0 & 1/2 & 0 & 0 & 0 \\ u & 0 & 0 & 0 & 1 & 0 & 0 \end{array}$$

```
Algorithm 1: Matrix Computation of Least Models

Input: a definite program $P$ and its Herband base $B_P = \{p_1,\ldots,p_n\}$.
Output: a vector $\boldsymbol{u}$ representing the least model of $P$.
 Step 1: Transform $P$ to a d-program $P^\delta = Q \cup D$ with $B_{P^\delta} = \{p_1,\ldots,p_n,p_{n+1},\ldots,p_m\}$ where $Q$ is an SD
   program and $D$ is a set of d-rules.
 Step 2: Embed $P^\delta$ into a vector space.
     - Create the matrix $\boldsymbol{M}_{P^\delta} \in \mathbb{R}^{m\times m}$ representing $P^\delta$.
     - Create the initial vector $\boldsymbol{v}_0 = (v_1,\ldots,v_m)^{\mathsf{T}}$ of $P^\delta$.
 Step 3: Compute the least model of $P^\delta$.
       $\boldsymbol{v} := \boldsymbol{v}_0$;
       $\boldsymbol{u} := \theta(\boldsymbol{M}_{P^\delta}\boldsymbol{v})$
       while $\boldsymbol{u} \neq \boldsymbol{v}$ do
           $\boldsymbol{v} := \boldsymbol{u}$;
           $\boldsymbol{u} := \theta(\boldsymbol{M}_{P^\delta}\boldsymbol{v})$
       end do
     return $\boldsymbol{u}[1\ldots n]$
```

Fig. 1. Algorithm for computing least models

An algorithm for computing the least model of a definite program P is shown in Figure 1. In the algorithm, the complexity of computing $\boldsymbol{M}_{P^\delta}\boldsymbol{v}$ is $O(m^2)$ and computing $\theta(\cdot)$ is $O(m)$. The number of times for iterating $\boldsymbol{M}_{P^\delta}\boldsymbol{v}$ is at most $(m+1)$ times. So the complexity of Step 3 is $O((m+1)\times(m+m^2)) = O(m^3)$ in the worst case.

3.3 Column Reduction

To decrease the complexity of computing $\boldsymbol{M}_{P^\delta}\boldsymbol{v}$, we introduce a technique of column reduction of program matrices.

Definition 8 (*submatrix representation of d-programs*)
Let P be a definite program such that $|B_P| = n$. Suppose that P is transformed to a d-program P^δ such that $P^\delta = Q \cup D$ where Q is an SD program and D is a set of d-rules, and $B_{P^\delta} = \{p_1,\ldots,p_m\}$. Then P^δ is represented by a matrix $\boldsymbol{N}_{P^\delta} \in \mathbb{R}^{m\times n}$ such that each element b_{ij} $(1 \le i \le m;\ 1 \le j \le n)$ in $\boldsymbol{N}_{P^\delta}$ is equivalent to the corresponding element a_{ij} $(1 \le i,j \le m)$ in $\boldsymbol{M}_{P^\delta}$ of Def. 7. $\boldsymbol{N}_{P^\delta}$ is called a *submatrix* of P^δ.

Note that the size of $\boldsymbol{M}_{P^\delta} \in \mathbb{R}^{m\times m}$ of Def. 7 is reduced to $\boldsymbol{N}_{P^\delta} \in \mathbb{R}^{m\times n}$ in Def. 8 by $n \le m$. In $\boldsymbol{N}_{P^\delta}$ the columns do not include values of newly introduced propositions and derivation of propositions in B_P via d-rules is checked by the following θ_D-thresholding.

Definition 9 (θ_D*-thresholding*)
Given a vector $\boldsymbol{v} = (a_1,\ldots,a_m)^{\mathsf{T}}$, define a vector $\boldsymbol{w} = \theta_D(\boldsymbol{v}) = (w_1,\ldots,w_m)^{\mathsf{T}}$ such that (i) $w_i = 1$ $(1 \le i \le m)$ if $a_i \ge 1$, (ii) $w_i = 1$ $(1 \le i \le n)$ if $\exists j\ w_j = 1$ $(n+1 \le j \le m)$ and there is a d-rule $d \in D$ such that $head(d) = p_i$ and $row_j(\boldsymbol{w}) \in body(d)$, and (iii) otherwise, $w_j = 0$. We call $\theta_D(\boldsymbol{v})$ the θ_D*-thresholding* of $\boldsymbol{v}$.

$\theta_D(\boldsymbol{v})$ 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 $\theta_D(\boldsymbol{v})$ is $O(m + (m-n)\times n) = O(m\times n)$. By definition, it holds that $\theta_D(\boldsymbol{v}) = \theta_D(\theta(\boldsymbol{v}))$.

Proposition 2
Let P be a definite program with $B_P = \{p_1, \ldots, p_n\}$, and P^δ a transformed d-program with $B_{P^\delta} = \{p_1, \ldots, p_n, p_{n+1}, \ldots, p_m\}$. Let $\boldsymbol{N}_{P^\delta} \in \mathbb{R}^{m \times n}$ be a submatrix of P^δ. Given a vector $\boldsymbol{v} \in \mathbb{R}^n$ representing an interpretation I of P, let $\boldsymbol{u} = \theta_D(\boldsymbol{N}_{P^\delta}\boldsymbol{v}) \in \mathbb{R}^m$. Then $\boldsymbol{u}$ is a vector representing an interpretation J of P^δ such that $J \cap B_P = T_P(I)$.

Given a program matrix $\boldsymbol{N}_{P^\delta} \in \mathbb{R}^{m \times n}$ and the initial vector $\boldsymbol{v}_0 \in \mathbb{R}^m$ of P^δ, define

$$\boldsymbol{v}_1 = \theta_D(\boldsymbol{N}_{P^\delta}\boldsymbol{v}_0[1\ldots n]) \quad \text{and} \quad \boldsymbol{v}_{k+1} = \theta_D(\boldsymbol{N}_{P^\delta}\boldsymbol{v}_k[1\ldots n]) \quad (k \geq 1)$$

where $\boldsymbol{N}_{P^\delta}\boldsymbol{v}_k[1\ldots n]$ represents the product of $\boldsymbol{N}_{P^\delta}$ and $\boldsymbol{v}_k[1\ldots n]$. Then it is shown that $\boldsymbol{v}_{k+1} = \boldsymbol{v}_k$ for some $k \geq 1$. When $\boldsymbol{v}_{k+1} = \boldsymbol{v}_k$, we write $v_k = \mathsf{FP}(\boldsymbol{N}_{P^\delta}\boldsymbol{v}_0[1\ldots n])$.

Theorem 3
Let P be a definite program with $B_P = \{p_1, \ldots, p_n\}$, and P^δ a transformed d-program with $B_{P^\delta} = \{p_1, \ldots, p_n, p_{n+1}, \ldots, p_m\}$. Then $\boldsymbol{m} \in \mathbb{R}^n$ is a vector representing the least model of P iff $\boldsymbol{m} = \mathsf{FP}(\boldsymbol{N}_{P^\delta}\boldsymbol{v}_0[1\ldots n])$ where $\boldsymbol{v}_0 \in \mathbb{R}^m$ is the initial vector of P^δ.

Generally, given a d-program P^δ, the value k of $\boldsymbol{v}_k = \mathsf{FP}(\boldsymbol{N}_{P^\delta}\boldsymbol{v}_0[1\ldots n])$ is not greater than the value h of $\boldsymbol{v}_h = \mathsf{FP}(\boldsymbol{M}_P\boldsymbol{v}_0)$ of Section 3.1.

Example 4
For the d-program P^δ of Example 3, we have the submatrix $\boldsymbol{N}_{P^\delta} \in \mathbb{R}^{6\times 4}$ representing P^δ. Given the initial vector $\boldsymbol{v}_0 = (0,0,0,1,0,0)^\mathsf{T}$ of P^δ, it becomes $\boldsymbol{v}_1 = \theta_D(\boldsymbol{N}_{P^\delta}\boldsymbol{v}_0[1\ldots 4]) = (0,1,0,1,0,1)^\mathsf{T}$, $\boldsymbol{v}_2 = \theta_D(\boldsymbol{N}_{P^\delta}\boldsymbol{v}_1[1\ldots 4]) = (1,1,0,1,0,1)^\mathsf{T}$, $\boldsymbol{v}_3 = \theta_D(\boldsymbol{N}_{P^\delta}\boldsymbol{v}_2[1\ldots 4]) = (1,1,0,1,0,1)^\mathsf{T} = \boldsymbol{v}_2$. Then $\boldsymbol{v}_2$ is a vector representing the least model of P^δ, and $\boldsymbol{v}_2[1\ldots 4]$ is a vector representing the least model $\{p,q,s\}$ of P. Note that the second element of $\boldsymbol{v}_i$ $(i = 1,2,3)$ becomes 1 by Def. 9(ii).

$$\begin{array}{c} \\ p \\ q \\ r \\ s \\ t \\ u \end{array} \begin{array}{c} \begin{array}{cccc} p & q & r & s \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

By Proposition 2, we can replace the computation $\boldsymbol{u} = \theta(\boldsymbol{M}_{P^\delta}\boldsymbol{v})$ in Step 3 of Algorithm 1 in Figure 1 by $\boldsymbol{u} = \theta_D(\boldsymbol{N}_{P^\delta}\boldsymbol{v}[1\ldots n])$. In the column reduction method, the complexity of computing $\boldsymbol{N}_{P^\delta}\boldsymbol{v}$ is $O(m \times n)$ and computing $\theta_D(\cdot)$ is $O(m \times n)$. The number of times for iterating $\boldsymbol{N}_{P^\delta}\boldsymbol{v}$ is at most $(m+1)$ times. So the complexity of computing $\boldsymbol{u} = \theta_D(\boldsymbol{N}_{P^\delta}\boldsymbol{v}[1\ldots n])$ is $O((m+1) \times (m \times n + m \times n)) = O(m^2 \times n)$. Comparing the complexity $O(m^3)$ of Step 3 in Algorithm 1, the column reduction reduces the complexity to $O(m^2 \times n)$ as $m \gg n$ in general.

4 Partial Evaluation

Partial evaluation is known as an optimization technique in logic programming (Lloyd & Shepherdson 1991). In this section, we provide a method of computing partial evaluation of definite programs in vector spaces.

Definition 10 (*partial evaluation*)
Let P be an SD program. For any rule r in P, put $U_r = \{\, r_i \mid r_i \in P \text{ and } head(r_i) \in body(r) \,\}$. Then construct a rule $r' = \mathsf{unfold}(r)$ such that

- $head(r') = head(r)$, and
- $body(r') = (body(r) \setminus \bigcup_{r_i \in U_r} \{head(r_i)\}) \cup \bigcup_{r_i \in U_r} body(r_i)$.

We define

$$\mathsf{peval}(P) = \bigcup_{r\in P} \mathsf{unfold}(r)$$

and call it *partial evaluation* of P.

Example 5
Consider $P = \{\, p \leftarrow q \wedge s \wedge t,\ \ q \leftarrow p \wedge t,\ \ s \leftarrow t,\ \ t \leftarrow \,\}$. Put $r_1 = (p \leftarrow q \wedge s \wedge t)$, $r_2 = (q \leftarrow p \wedge t)$, $r_3 = (s \leftarrow t)$, and $r_4 = (t \leftarrow)$. Unfolding rules produces: $\mathsf{unfold}(r_1) = (p \leftarrow p \wedge t \wedge t) = (p \leftarrow p \wedge t)$, $\mathsf{unfold}(r_2) = (q \leftarrow q \wedge s \wedge t)$, $\mathsf{unfold}(r_3) = (s \leftarrow)$, and $\mathsf{unfold}(r_4) = (t \leftarrow)$. Then it becomes $\mathsf{peval}(P) = \{\, p \leftarrow p \wedge t,\ \ q \leftarrow q \wedge s \wedge t,\ \ s \leftarrow,\ \ t \leftarrow \,\}$.

By definition, $\mathsf{peval}(P)$ is obtained from P by unfolding propositional variables appearing in the body of any rule in P in parallel. Partial evaluation preserves the least model of the original program (Lloyd & Shepherdson 1991).

Proposition 3
Let P be an SD program. Then P and $\mathsf{peval}(P)$ have the same least model.

Partial evaluation is computed by matrix products in vector spaces.

Example 6
The program P of Example 5 is represented by the matrix $\boldsymbol{M}_P$, and $(\boldsymbol{M}_P)^2$ becomes

$$\boldsymbol{M}_P = \begin{array}{c} \\ p \\ q \\ s \\ t \end{array} \begin{array}{c} \begin{array}{cccc} p & q & s & t \end{array} \\ \begin{pmatrix} 0 & 1/3 & 1/3 & 1/3 \\ 1/2 & 0 & 0 & 1/2 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \qquad (\boldsymbol{M}_P)^2 = \begin{pmatrix} 1/6 & 0 & 0 & 5/6 \\ 0 & 1/6 & 1/6 & 2/3 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Intuitively speaking, non-zero elements in $(\boldsymbol{M}_P)^2$ represent conjuncts appearing in each rule. So the first row represents the rule $p \leftarrow p \wedge t$ and the second row represents the rule $q \leftarrow q \wedge s \wedge t$. $(\boldsymbol{M}_P)^2$ then represents $P' = \{\, p \leftarrow p \wedge t,\ \ q \leftarrow q \wedge s \wedge t,\ \ s \leftarrow t,\ \ t \leftarrow \,\}$. P' is different from $\mathsf{peval}(P)$ for the representation of the rule $s \leftarrow t$. This is because $t \leftarrow$ is represented as $t \leftarrow t$ in $\boldsymbol{M}_P$, so that unfolding $s \leftarrow t$ by $t \leftarrow t$ becomes $s \leftarrow t$. Thus, $(\boldsymbol{M}_P)^2$ does not represent the result of unfolding rules by facts precisely, while it does not affect the result of computing the least model of P. In fact, applying the vector $\boldsymbol{v}_0 = (0,0,0,1)^\mathsf{T}$ representing facts in P and applying θ-thresholding, we obtain $\theta((\boldsymbol{M}_P)^2\boldsymbol{v}_0) = (0,0,1,1)$ that represents the least model $\{\, s, t \,\}$ of P. We say that $(\boldsymbol{M}_P)^2$ represents the *rule by rule* (shortly, *r-r*) partial evaluation, and often say just partial evaluation when no confusion arises. Formally, we have the next result.

Proposition 4
Let P be an SD program and $\boldsymbol{v}_0$ the initial vector representing facts of P. Then $\theta((\boldsymbol{M}_P)^2\boldsymbol{v}_0) = \theta(\boldsymbol{M}_P(\theta(\boldsymbol{M}_P\boldsymbol{v}_0)))$.

Partial evaluation has the effect of reducing deduction steps by unfolding rules in advance. Proposition 4 realizes this effect by computing matrix products in advance. Partial evaluation is performed iteratively as

$$\mathsf{peval}^k(P) = \mathsf{peval}(\mathsf{peval}^{k-1}(P))\ \ (k \geq 1)\ \ \text{ and }\ \ \mathsf{peval}^0(P) = P.$$

Iterative partial evaluation is computed by matrix products as follows.

```
Algorithm 2: Partial Evaluation

Input: a definite program $P$ and its Herband base $B_P$.
       $k\,(\geq 0)$: the number of iteration of partial evaluation.
Output: a vector $\boldsymbol{u}$ representing the least model of $P$.
 Step 1: Transform $P$ to a d-program $P^\delta = Q \cup D$ where $Q$ is an SD program and $D$ is a set of d-rules.
 Step 2: Embed $P^\delta$ into a vector space.
     - Create the matrix $\boldsymbol{M}_Q$ representing $Q$.
     - Create the matrix $\boldsymbol{M}_D$ representing $D$.
 Step 3: Compute (r-r) partial evaluation of $Q$.
      $\Gamma^1_Q = (\boldsymbol{M}_Q)^2$;
      $i := 1$; while $i \leq k$ do
          $\Gamma^{i+1}_Q = (\Gamma^i_Q)^2$;
      $i := i+1$; end do
        Compute $\Gamma^k_{P\delta} := \Gamma^k_Q + \boldsymbol{M}_D$
     Create the vector $\boldsymbol{v}_0$ representing the facts of $Q$
        $\boldsymbol{v} := \boldsymbol{v}_0$;
        $\boldsymbol{u} := \theta(\Gamma^k_{P\delta}\boldsymbol{v}_0)$;
        while $\boldsymbol{u} \neq \boldsymbol{v}$ do
            $\boldsymbol{v} := \boldsymbol{u}$;
            $\boldsymbol{u} := \theta(\Gamma^k_{P\delta}\boldsymbol{v})$;
        end do
     return $\boldsymbol{u}$
```

Fig. 2. Algorithm for computing least models by partial evaluation

Let P be an SD program and $\boldsymbol{M}_P \in \mathbb{R}^{n\times n}$ its program matrix. Define $\Gamma^1_P = (\boldsymbol{M}_P)^2$ and $\Gamma^{k+1}_P = (\Gamma^k_P)^2$ $(k \geq 1)$. Then Γ^k_P is a matrix representing a program that is obtained by k-th iteration of (r-r) partial evaluation.

Theorem 4
Let P be an SD program and $\Gamma^k_P \in \mathbb{R}^{n\times n}$ $(k \geq 1)$. Then $\theta(\Gamma^k_P \boldsymbol{v}_0) = \boldsymbol{v}_{2k}$ where $\boldsymbol{v}_k = \theta(\boldsymbol{M}_P \boldsymbol{v}_{k-1})$.

When P is a non-SD program, first transform P to a d-program $P^\delta = Q \cup D$ where Q is an SD program and D is a set of d-rules (Section 3.2). Next, define $\Gamma^k_{P\delta} = \Gamma^k_Q + \boldsymbol{M}_D$. We then compute (r-r) partial evaluation of P^δ as (r-r) partial evaluation of an SD program Q plus d-rules D.

An algorithm for computing the least model of a definite program P by (r-r) partial evaluation is shown in Figure 2. We can combine partial evaluation and column reduction of Section 3.3 by slightly changing Step 3 of Algorithm 2. We evaluate this hybrid method in the next section.

5 Experimental Results

In this section, we compare runtime for computing the least model of a definite program. The testing is done on a computer with the following configuration:

- Operating system: Linux Ubuntu 16.04 LTS 64bit
- CPU: Intel Core^{TM} i7-6800K (3.4 GHz/14nm/Cores=6/Threads=12/Cache=15MB), Memory 32GB, DDR-2400
- GPU: GeForce GTX1070TI GDDR5 8GB

```
Input: a definite program $P$.
Output: the least model of $P$.
 $I := $ set of facts in $P$;
 $J := \emptyset$;
 while $(I \neq J)$ do
         $J := I$;
         for $r$ in $P$ do
            if $body(r) \subseteq J$ then $I := I \cup \{head(r)\}$;
            end do
         return $J$
```

Fig. 3. Algorithm for computing least models by T_P-operator

- Implementation language: Maple 2017, 64 bit[4]

Given the size $n = \mid B_P \mid$ of the Herband base B_P and the number $m = \mid P \mid$ of rules in P, rules are randomly generated 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)	$x < \frac{n}{3}$	4%	4%	10%	40%	35%	4%	2%	0-1%

A definite program P is randomly generated based on (n, m). We set those parameters as $n \ll m$, so generated programs are non-SD programs and they are transformed to d-programs. We compare runtime for computing the least model of P by the following four methods: (a) computation by the T_P-operator; (b) computation by program matrices; (c) computation by column reduction; and (d) partial evaluation. Computation by the T_P-operator is done by the procedure in Figure 3. Computation by program matrices is done by Algorithm 1, and computation by column reduction is done by modifying Step 3 of Algorithm 1 (see Sec. 3.3). In partial evaluation, the input parameter k of Algorithm 2 is set as $k = 1, 5, \frac{n}{2}, n$ where $n = \mid B_P \mid$. We then compute a vector representing the least model of P in two ways: program matrices and column reduction.

We perform experiments by changing parameters (n, m, k). For each (n, m, k) we measure runtime at least three times and pick average values. Tables 2, 3 and 4 show the results of testing for $n = 50, 100$ and 200, respectively. In the table,"all" means time for creating a program matrix and computing a fixpoint, and "fixpoint" means time for computing a fixpoint. Figure 4 compares runtime for computing fixpoints.

By those tables, we can observe the following facts.

- For fixpoint computation, column reduction outperforms matrix computation and the T_P-operator in almost every cases. Naive computation by program matrices becomes inefficient in large scale of programs.

[4] Maple 2017 supports to use GPU for accelerating linear algebraic computation by CUDA package (Maple 2017). The experimental results in this section do not use the CUDA package, however.

- Column reduction is effective in a large scale of programs. It is often more than 10 times faster than naive computation by program matrices.
- By performing partial evaluation, time for fixpoint computation is significantly reduced. In particular, partial evaluation + column reduction is effective when $k > 1$, and fixpoint computation by this hybrid method is often more than 10 times faster than other methods in large scale of programs (Tables 3 and 4).

Table 2. *Experimental Results ($n = 50$; sec)*

m	T_P-operator	matrix		column reduction		partial evaluation			
		fixpoint	all	fixpoint	all	k	Γ^k	matrix	col. reduct.
100	0.008	0.007	0.155	0.005	0.13	1	0.002	0.005	0.005
						5	0.006	0.006	0.005
						25	0.006	0.005	0.005
						50	0.008	0.009	0.004
1250	0.35	1.135	1.158	0.061	0.247	1	0.29	0.111	0.173
						5	0.656	0.04	0.019
						25	0.565	0.029	0.012
						50	0.56	0.032	0.012
2500	0.627	1.269	1.3	0.071	0.142	1	0.438	0.133	0.227
						5	1.41	0.066	0.05
						25	1.81	0.063	0.043
						50	1.401	0.067	0.06
12500	2.081	13.937	14.358	0.649	1.024	1	38.938	1.142	3.189
						5	78.646	0.585	0.168
						25	79.625	0.604	0.168
						50	80.181	0.606	0.168

6 Conclusion

In this paper, we introduced a method of embedding logic programs in vector spaces. We developed algorithms for computing least models of definite programs, and presented column reduction and partial evaluation for optimization. Experimental results show that column reduction is effective to realize efficient computation in a large scale of programs and partial evaluation helps to reduce runtime significantly. It is known that the least model of a definite program is computed in $O(N)$ (Dowling & Gallier 1984) where N is the size (number of literals) of a program. Since column reduction 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. Moreover, since partial evaluation is performed apart from fixpoint computation, combination of column reduction and partial evaluation would be effective in practice. The linear algebraic approach enables us to use efficient algorithms of numerical linear algebra and opens perspective for parallel computation of logic

Table 3. *Experimental Results ($n = 100$; sec)*

m	T_P-operator	matrix		column reduction		partial evaluation			
		fixpoint	all	fixpoint	all	k	Γ^k	matrix	col. reduct.
200	0.029	0.014	0.05	0.003	0.021	1	0.007	0.006	0.009
						5	0.018	0.007	0.007
						50	0.017	0.007	0.01
						100	0.026	0.008	0.009
5000	2.206	3.981	4.044	0.249	0.485	1	2.357	0.288	0.608
						5	5.921	0.143	0.117
						50	6.696	0.136	0.094
						100	6.403	0.143	0.094
10000	2.355	18.553	18.836	1.131	1.807	1	38.549	1.502	1.778
						5	79.68	0.603	0.075
						50	78.037	0.576	0.076
						100	77.58	0.575	0.074

Table 4. *Experimental Results ($n = 200$; sec)*

m	T_P-operator	matrix		column reduction		partial evaluation			
		fixpoint	all	fixpoint	all	k	Γ^k	matrix	col. reduct.
400	0.06	0.063	0.075	0.013	0.06	1	0.047	0.023	0.022
						5	0.071	0.018	0.019
						100	0.102	0.018	0.024
						200	0.087	0.016	0.019
20000	6.391	25.161	25.833	4.48	7.771	1	138.651	2.317	6.423
						5	295.462	1.173	0.529
						100	289.564	1.15	0.519
						200	285.345	1.203	0.519

programming. Performance of our implementation heavily depends on the environment of linear algebraic computation. For instance, we could use the CUDA package to accelerate linear algebraic computation on GPU. Once more powerful platforms are developed for linear algebraic computation, the proposed method would have the merit of such advanced technologies. We have used Maple for implementation, but the proposed algorithms can be realized by other programming languages. We compared runtime in experiments, while it would be interesting to compare other metrics in algorithms and matrices that are part of the computation, for instance, the number of iterations to the fixpoint, the compression $(m-n)/m$ achieved by column reduction, the sparseness of the matrices with and without partial evaluation, and so on.

This paper studies algorithms for computing least models of definite programs. An important question is whether linear algebraic computation is applied to answer set programming. A method for computing stable models of normal logic programs was reported in (Sakama *et al.* 2017) in which normal programs are represented by third-order tensors. Computing large scale of programs in third-order tensors requires scalable techniques and optimization, however. As an alternative approach, we can use a technique of transforming normal programs to definite programs, and computing stable models as least models of the transformed programs (Alferes *et al.* 2000). A preliminary report based on this method is under preparation (Nguyen *et al.* 2018), and partial evaluation would help to reduce runtime. We also plan to develop a new algorithm for ASP in vector spaces and evaluate it using benchmark testing. Recently, Sato *et al.* (2018) introduce a method of linear algebraic computation of abduction in Datalog. We consider that abductive logic programming would be realized in vector spaces by extending the framework introduced in this paper. There is a number of interesting topics to be investigated and rooms for improvement in this new approach to logic programming.

References

ALFERES, J. J., LEITE, J. A., PEREIRA, L. M., PRZYMUSINSKA, H. AND PRZYMUSINSKI, T. 2000. Dynamic updates of non-monotonic knowledge bases. *Journal of Logic Programming* 45:43–70.

DOWLING, W. F. AND GALLIER, J. H. 1984. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming* 1(3):267–284.

GREFENSTETTE, E. 2013. Towards a formal distributional semantics: simulating logical calculi with tensors. In: *Proc. 2nd Joint Conf. Lexical and Computational Semantics*, pp. 1–10

LIN, F 2013. From satisfiability to linear algebra. Invited talk. *26th Australasian Joint Conference on Artificial Intelligence*.

LLOYD, J. W. AND SHEPHERDSON, J. C. 1991. Partial evaluation in logic programing. *Journal of Logic Programming* 11:217–242.

MAPLE 2017. https://www.maplesoft.com/support/help/maple/view.aspx?path=CUDA

NGUYEN, G. D., SAKAMA, C., SATO, T. AND INOUE, K. 2018. Computing logic programming semantics in linear algebra (draft).

SAKAMA, C., INOUE, K. AND SATO, T. 2017. Linear algebraic characterization of logic programs. In: *Proc. 10th International Conference on Knowledge Science, Engineering and Management*, Lecture Notes in Artificial Intelligence 10412, Springer, pp. 520–533.

SATO, T. 2017a. Embedding Tarskian semantics in vector spaces. In: *Proceedings of the AAAI-17 Workshop on Symbolic Inference and Optimization*.

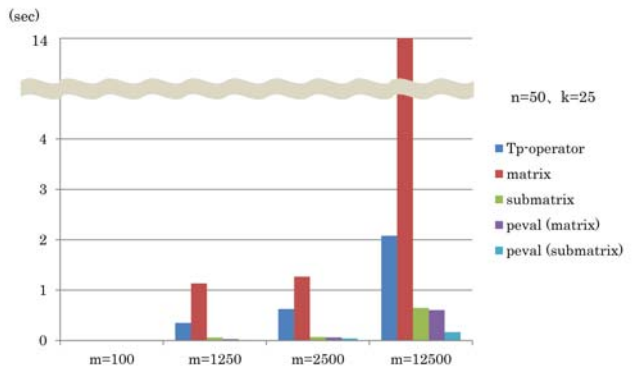
SATO, T. 2017b. A linear algebraic approach to Datalog evaluation. TPLP 17(3):244–265.

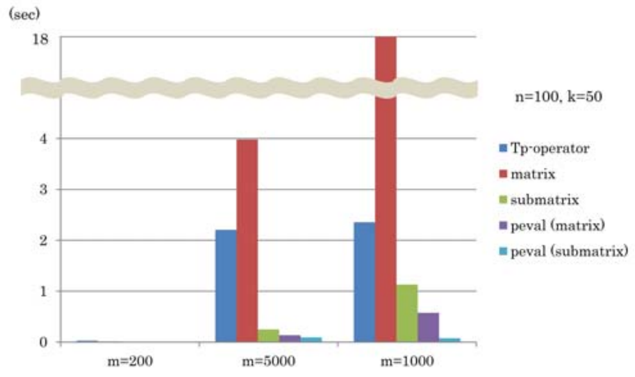
SATO, T., INOUE, K. AND SAKAMA, C. 2018. Abducing relations in continuous spaces. In: *Proc. IJCAI-18*, forthcoming.

SERAFINI, L. AND D'AVILA GARCEZ, A. 2016. Learning and reasoning with logic tensor networks. In: *AI*IA 2016: Advances in Artificial Intelligence, Lecture Notes in Artificial Intelligence* 10037, Springer, pp. 334–348.

VAN EMDEN, M. H. AND KOWALSKI, R. A. 1976. The semantics of predicate logic as a programming language. *Journal of the ACM* 23(4):733–742.

YANG, B., YIH, W.-T., HE, X., GAO, J., AND DENG, L. 2015. Embedding entities and relations for learning and inference in knowledge bases. In: *Proceedings of the International Conference on Learning Representations*.





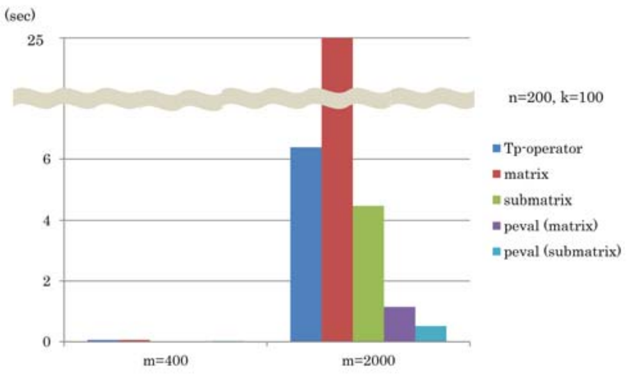


Fig. 4. Comparison of runtime for fixpoint computation