Linear Algebraic Partial Evaluation of Logic Programs*

Tuan Nguyen[1,*], Katsumi Inoue[1] and Chiaki Sakama[2]

[1]*National Institute of Informatics (NII), 2-1-2 Hitotsubashi, Chiyoda City, Tokyo, Japan*
[2]*Wakayama University, 930 Sakaedani, Wakayama, Japan*

Abstract
In logic programming, partial evaluation performs unfolding rules of a program in advance to reduce the cost of inferencing steps. Recently, partial evaluation of logic programs has been implemented in vector spaces by computing the powers of matrix representations. It has been reported that linear algebraic partial evaluation substantially enhances the practical performance of linear algebraic methods for logic programming. However, most recent research has focused exclusively on And-rules, assuming that their dependency graph is acyclic. In this paper, we introduce cycle-resolving techniques to ensure that linear algebraic partial evaluation works effectively even with cycles in the program. Additionally, we demonstrate that linear algebraic partial evaluation can also be extended to accommodate Or-rules. Moreover, we propose using eigendecomposition and Jordan normal form to conduct the partial evaluation in vector spaces. We compare the proposed techniques on a set of acyclic and cyclic logic programs to evaluate their effectiveness. It is shown that the iteration method for partial evaluation, especially with sparse format, is the most efficient one in general cases. However, the decomposition method has the potential for future research to leverage eigenvalues and eigenvectors of program matrices for reasoning with logic programming.

Keywords
Logic programming, Partial evaluation, Linear algebra

1. Introduction

Recent research has explored using linear algebraic methods as a compelling alternative to symbolic methods for logical inference [1, 2, 3, 4]. In 2017, Sakama et al. proposed linearizing logic program characteristics using matrix multiplication for deductive reasoning [1]. This involves converting a logic program into a matrix and using matrix-vector multiplication to realize the immediate consequence operator [5]. Extensions to disjunctive and normal logic programs were also discussed [6]. Another approach by Sato et al. investigates computing 2-valued and 3-valued completion semantics of finite propositional normal logic programs in vector spaces [7]. Similarly, Aspis et al. considered model computation in continuous vector spaces as a root-finding problem, using Newton's method to solve it [3]. Later, Takemura and Inoue proposed a differentiable approach by designing a continuous loss function where supported models are optimal values [4]. Additionally, using matrices and tensors to represent logical formulas and constraints is seen as a promising way to connect symbolic reasoning and machine learning [8]. Matrix representation allows constructing And/Or Boolean networks from state transitions on an unprecedented scale [9]. Program matrices can also be learned using machine learning methods, as demonstrated in [10], where a differentiable inductive logic programming framework learns logic programs from relational datasets. This idea has been extended to a differentiable first-order rule learner, shown to be robust to noisy data and scalable to large datasets [11].

Linear algebraic approaches have also been extended to Partial Evaluation (PE) in Logic Programming (LP) [12]. Nguyen et al. reported significant runtime reductions on both synthetic and real data, especially for transitive closures of large network datasets [12]. A similar linear algebraic PE concept has been applied to Propositional Horn Clause Abduction Problem (PHCAP), showing remarkable performance gains [13]. Although these methods are applied to different reasoning tasks, the main idea behind linear algebraic PE in both [12] and [13] is to compute the powers of matrix representations of logic programs ([13] employs abductive matrix, but it is actually the transposed version of the program matrix in [12]). Both papers use a unified representation of a logic program in its standardized form to perform PE in an iterative manner. However, their methods only focus on the And-rules in the program. More importantly, they assume that the dependency graph of the program is acyclic and do not consider the cyclic case.

In this work, we focus on extending the capability of linear algebraic PE. First, we propose to separate the matrix representation of a logic program into two parts: one for And-rules and the other for Or-rules. In short, an And-rule is a rule that has a conjunction of literals in its body, the head is **True** only if all its body literals are **True**. On the other hand, an Or-rule has a disjunction of literals in its body, the head is **True** if at least one of its body literals is **True**. Each part (And-rules or Or-rules) of a logic program has different logical meanings but can be treated equally in terms of PE computation which is basically computing powers of a square matrix. We also propose a solution to resolve cycles in the dependency graph of the program to extend the capability of linear algebraic PE to the cyclic case. Moreover, we introduce a novel way to realize PE in vector spaces by leveraging the eigenvalues and eigenvectors.

The rest of this paper is organized as follows: Section 2 reviews background knowledge of LP and dependency graphs; Section 4 presents the iteration method for PE and cycle-resolving techniques; Section 5 demonstrates linear algebraic PE using eigendecomposition and Jordan normal form; Section 6 illustrates comparison of the proposed PE methods; finally Section 7 concludes the paper.

22nd International Workshop on Nonmonotonic Reasoning, November 2-4, 2024, Hanoi, Vietnam
*Corresponding author.
tuannq@nii.ac.jp (T. Nguyen); inoue@nii.ac.jp (K. Inoue); sakama@wakayama-u.ac.jp (C. Sakama)
https://profile.nqtuan0192.me/ (T. Nguyen); https://research.nii.ac.jp/il/ (K. Inoue); https://web.wakayama-u.ac.jp/~sakama/ (C. Sakama)
0000-0002-1754-9329 (T. Nguyen); 0000-0002-2717-9122 (K. Inoue); 0000-0002-9966-3722 (C. Sakama)

2. Background

In this paper, we focus on propositional logic programs over a finite (nonempty) set of atoms $\mathcal{A}$. A program *P* is called a *normal logic program* if every rule $r \in P$ follows the form:

$$h \leftarrow b_1 \ \wedge \ b_2 \ \wedge ... \wedge \ b_l \ \wedge \ \neg b_{l+1} \ \wedge ... \wedge \ \neg b_k \quad (k \geq l \geq 0) \tag{1}$$

where h and b_i are atoms in $\mathcal{A}$.
For short, we write $head(r)$ and $body(r)$ to denote the set of literals in the head and body of a rule r, respectively. Additionally, $body(r)$ can be partitioned into $body^+(r) = \{b_1, \ b_2, ..., \ b_l\}$ and $body^-(r) = \{\neg b_{l+1}, \ \neg b_{l+2}, ..., \ \neg b_k\}$ which refers to the *positive* and *negative* literals in $body(r)$.
A normal rule r is called a *fact* if $body(r) = \emptyset$, a *constraint* if $head(r) = \emptyset$. A fact or a constraint can also be written respectively as $head(r) \leftarrow \top$ and $\bot \leftarrow body(r)$, where $\top$ and $\bot$ are special symbols representing **True** and **False**. In case $body^-(r) = \emptyset$, the rule r is called a *definite rule.* A normal program *P* is a *definite program* if $body^-(r) = \emptyset$ for every rule $r \in P$.

A normal logic program can be transformed into a definite program as mentioned in [14]. Accordingly, one can obtain a definite program from a normal program *P* by replacing the negative literals in every rule (1) and rewriting as follows:

$$h \leftarrow b_1 \ \wedge \ b_2 \ \wedge ... \wedge \ b_l \ \wedge \ \overline{b}_{l+1} \ \wedge ... \wedge \ \overline{b}_k \quad (k \geq l \geq 0) \tag{2}$$

where each $\overline{b}_i$ is the positive form of the negation $\neg b_i$.
The resulting program is called the *positive form* of *P*, denoted as P^+. P^+ then can be transformed into a *standardized program* which is a definite program that satisfies there are no two rules with the same head - Singly-Defined (SD) condition [6]. A logic program *P* is called a SD program if $head(r_1) \neq head(r_2)$ for any two different rules r_1, r_2 in *P*. When *P* contains more than one rule $r_1, \ r_2, \ \ldots, \ r_n$ $(n > 1)$ with the same head h such that $head(r_1) = head(r_2), \cdots = head(r_n) = \{h\}$, replace those rules with a set of new rules: $\{h \leftarrow b_1 \vee b_2 \vee \ldots \vee b_n, \ b_1 \leftarrow body(r_1), \ \ b_2 \leftarrow body(r_2), \ \ \ldots, \ \ b_n \leftarrow body(r_n)\}$ $(n > 1)$, where $b_1, b_2, \ldots, b_n$ are newly introduced atoms. The resulting program Π is called a *standardized program.*

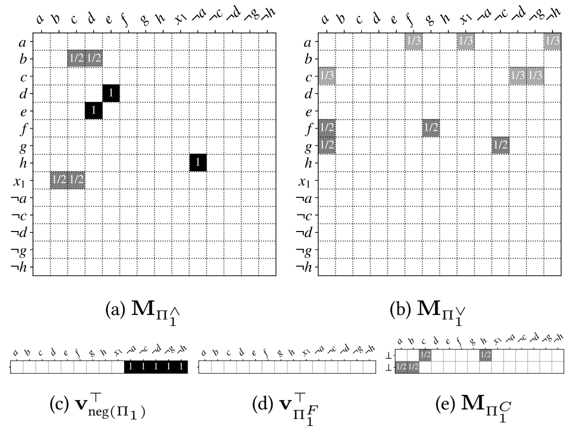
Accordingly, Π can be seen as a finite set of rules of the form And-rules (3) and Or-rules (4), and there are no two rules with the same head (*SD* condition):

$$h \leftarrow b_1 \ \wedge \ b_2 \ \wedge ... \wedge \ b_l \qquad (l \geq 0) \tag{3}$$
$$h \leftarrow b_1 \ \vee \ b_2 \ \vee ... \vee \ b_l \qquad (l \geq 2) \tag{4}$$

For simplicity, we still use the notation $\neg p$ in a standardized program Π but, without ambiguity, imply that $\neg p$ and p are two “distinct” variables with a “special” relation.

Example 1. *Given a logic program* $P_1 = \{a \leftarrow b \wedge c, \ a \leftarrow f, \ a \leftarrow \neg h, \ b \leftarrow c \wedge d, \ c \leftarrow a, \ c \leftarrow \neg g, \ c \leftarrow \neg d, \ d \leftarrow e, \ e \leftarrow d, \ f \leftarrow a, \ f \leftarrow g, \ g \leftarrow a, \ g \leftarrow \neg c, \ h \leftarrow \neg a, \ \leftarrow c \wedge h, \ \leftarrow b \wedge a\}$.
Standardized logic program: $\Pi_1 = \{a \leftarrow x_1 \vee f \vee \neg h, \ b \leftarrow c \wedge d, \ c \leftarrow a \vee \neg g \vee \neg d, \ d \leftarrow e, \ e \leftarrow d, \ f \leftarrow a \vee g, \ g \leftarrow a \vee \neg c, \ h \leftarrow \neg a, \ x_1 \leftarrow b \wedge c, \ \leftarrow c \wedge h, \ \leftarrow b \wedge a\}$.

Here in Example 1, note that we do not need to introduce new variables for each body atom in $f \leftarrow a, \ f \leftarrow g$ and



(a) $\mathbf{M}_{\Pi_1^\wedge}$
(b) $\mathbf{M}_{\Pi_1^\vee}$
(c) $\mathbf{v}^\top_{\text{neg}(\Pi_1)}$
(d) $\mathbf{v}^\top_{\Pi_1^F}$
(e) $\mathbf{M}_{\Pi_1^C}$

Figure 1: Matrix/vector representations of Π_1.

$g \leftarrow a, \ g \leftarrow \neg c$, because these rules have single-literal bodies. In case the rule body has more than one atom, we need to introduce a new variable for each body atom and rewrite the rule as a disjunction of these new variables. Further details about the standardization method can be found in [15].

3. Logic Programs - Program Matrices - Dependency Graphs

3.1. Matrix representation of logic programs

We follow a similar program matrix definition as [6]. Our new observation is that a standardized program Π can be seen as a quadruple $\Pi = \langle \Pi^\wedge, \Pi^\vee, \Pi^F, \Pi^C \rangle$ where $\Pi^\wedge$ is the set of non-factual And-rules ((3) but strictly $l > 0$), $\Pi^\vee$ is the set of Or-rules (4), Π^F is the set of facts ((3) where $l = 0$) and Π^C is the set of constraints ((3) where $h = \bot$). For convenience, we assume there is a way to index all literals in a logic program incrementally without ambiguity so that we can easily map sets of literals to vectors. We shall define the matrix representation of Π as a set of matrices and vectors as follows.

Definition 1 (Matrix of *And*-rules/*Or*-rules). *Let* $\Pi = \langle \Pi^\wedge, \Pi^\vee, \Pi^F, \Pi^C \rangle$ *be a standardized program. Then the matrix of* And*-rules* $\mathbf{M}_{\Pi^\wedge}$ *(*Or*-rules* $\mathbf{M}_{\Pi^\vee}$*), where* $\mathbf{M}_{\Pi^\wedge} \in \mathbb{R}^{n_\Pi \times n_\Pi}$ *(*$\mathbf{M}_{\Pi^\vee} \in \mathbb{R}^{n_\Pi \times n_\Pi}$*,* n_Π *is the number of atoms in* Π*), are defined as follows:*
• $\mathbf{M}_{\Pi^{\{\_\}}}[i, j] = \dfrac{1}{l}$ *if there is a rule* r_i *in* $\Pi^{\{\_\}}$ *(*r_i *either in the form of (3) or (4) respectively if* $\Pi^{\{\_\}}$ *is* $\Pi^\wedge$ *or* $\Pi^\vee$*) where* $l = |body(r_i)| \neq 0$,
• $\mathbf{M}_{\Pi^{\{\_\}}}[i, j] = 0$ *otherwise.*

We define *vector of negations* as a column vector $\mathbf{v}$ such that $\mathbf{v}[i] = 1$ if the corresponding atom at index i is a negation. Similarly, *vector of facts* is defined as a column vector $\mathbf{v}$ such that $\mathbf{v}[i] = 1$ if the corresponding atom at index i is a fact. Figure 1 demonstrates the visualization of matrix/vector representations of Π_1 in Example 1. By definitions, non-zero elements of $\mathbf{M}_{\Pi^\wedge}$, $\mathbf{M}_{\Pi^\vee}$, and $\mathbf{M}_{\Pi^C}$ are normalized by the number of atoms in the body of the corresponding rule. It is possible to define the matrix without normalization as long as being consistent. In the context of logic inferencing, we follow the normalized representation as it is more convenient to define **True** as 1 and **False** as 0.

We shall show the connection between this matrix representation and the one defined in [6] that has been adopted in [12] and [13] to define linear algebraic PE. Before that, we need to define two thresholding functions:

Definition 2 (Thresholding functions).
$$\theta^{\Downarrow}(x) = \begin{cases} 1 & \text{if } x \geq 1 \\ 0 & \text{otherwise} \end{cases}, \text{ and } \theta^{\Uparrow}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$
where x is a scalar and can be extended to a vector, or a matrix in an element-wise way.

The program matrix $\mathbf{M}_\Pi$ can be constructed as follows:

$$\mathbf{M}_\Pi = \mathbf{M}_{\Pi^\wedge} + \theta^{\Uparrow}(\mathbf{M}_{\Pi^\vee}) + \mathtt{diag}(\mathbf{v}_{\Pi^F} \oplus_{\theta^{\Downarrow}} \mathbf{v}_{\mathrm{neg}(\Pi)}) \quad (5)$$

where $\oplus_{\theta^{\Downarrow}}$ is vector add with $\theta^{\Downarrow}$-thresholding, `diag` turns an input vector into a diagonal matrix. The reason for $\oplus_{\theta^{\Downarrow}}$ is that there might be a chance where atoms (known to be **False** are included as facts) and negations are overlapping. Program matrix $\mathbf{M}_\Pi$ in (5) is equivalent to the one defined in [6] that can be used either for fixpoint computation in stable model computation [16] or for 1-step abduction in Horn abduction (with restrictions to Horn clauses) [13]. The reason for the redefinition is to make the matrix representation more intuitive so that we can develop a general PE approach and cycle-resolving techniques to both And-rules and Or-rules.

3.2. Dependency graphs

In order to illustrate the relationship between program completion [17] and stable models [18], the concept of "dependency graph" was employed in several studies i.e. [19]. In this section, we will extend the concept of dependency graphs to the case of standardized programs.

Definition 3 (Dependency graph). *Given a normal logic program P. The* dependency graph *of P is a directed graph $\mathbf{G}_P = (\mathbf{V}_P, \mathbf{E}_P)$ where $\mathbf{V}_P$ is the set of atoms in P and $\mathbf{E}_P$ is determined as follows:*

- *There is a positive edge (u, v) in $\mathbf{E}_P$ if there is a rule $r \in P$ such that $u = head(r)$ and $v \in body^+(r)$.*
- *There is a negative edge (u, v) in $\mathbf{E}_P$ if there is a rule $r \in P$ such that $u = head(r)$ and $v \in body^-(r)$.*

Note that the direction of an edge (u, v) does not matter unless we are consistent. However, in our paper, we persist in the direction of edges toward the head atom of a rule. In many studies, the definition of *positive dependency graph* is usually preferred over the general dependency graph [20]. Given a normal logic program *P*. The *positive dependency graph* of *P* is a directed graph $\mathbf{G}^+_P = (\mathbf{V}_P, \mathbf{E}^+_P)$ such that $\mathbf{G}^+_P \subseteq \mathbf{G}_P$ where $\mathbf{G}_P$ is the dependency graph of *P* such that $\mathbf{E}^+_P$ includes only positive edges of $\mathbf{E}_P$. *P* is called a *tight program* if $\mathbf{G}^+_P$ is acyclic [21], in other words, there is no positive loop in $\mathbf{G}^+_P$. For tight programs, the completion semantics and the answer set semantics are equivalent to each other [21].

Dependency graphs are sufficient to illustrate the relationship between literals in a normal logic program. However, it is difficult to capture how to "interpret" a rule in a dependency graph. For example, in Figure 2a, we can see that a depends on b, c, f, and $\neg h$, however, it is not clear to see that all of them are required to deduce a or which combination is sufficient. A similar argument holds for the case of the positive dependency graph in Figure 2b.

To capture the "actual meaning" of a rule, we introduce the concept of And-Or *dependency graph* which is defined over a standardized program Π. As mentioned, $\Pi = \langle \Pi^\wedge, \Pi^\vee, \Pi^F, \Pi^C \rangle$. We consider to define the dependency graph of Π only over $\Pi^\wedge$ (And-rules) and $\Pi^\vee$ (Or-rules). Regardless of a rule in $\Pi^\wedge$ or $\Pi^\vee$ may differ as a conjunction or disjunction, we can always define the *dependency graph* of $\Pi^\wedge$ and $\Pi^\vee$ separately using Definition 3, denoted as $\mathbf{G}_{\Pi^\wedge}$ and $\mathbf{G}_{\Pi^\vee}$ respectively. Note that $\Pi^\wedge$ and $\Pi^\vee$ do not contain any negation by definition, therefore, $\mathbf{G}_{\Pi^\wedge}$ and $\mathbf{G}_{\Pi^\vee}$ are also positive dependency graphs. To distinguish edges of $\mathbf{G}_{\Pi^\wedge}$ and $\mathbf{G}_{\Pi^\vee}$ from each other, we use solid and dash lines respectively.

Definition 4 (And-Or **dependency graph).** *Given a normal logic program P, its standardized program is Π. The And-Or dependency graph of Π is a directed graph $\mathbf{G}_\Pi$ such that $\mathbf{G}_\Pi = \mathbf{G}_{\Pi^\wedge} \cup \mathbf{G}_{\Pi^\vee}$.*

As can be seen in Figure 2d and Figure 2e, each graph $\mathbf{G}_{\Pi^\wedge}$ or $\mathbf{G}_{\Pi^\vee}$ only contains edges of the same type, either solid or dash lines. However, in the And-Or dependency graph $\mathbf{G}_\Pi$ in Figure 2c, both types of edges are presented. It is easy to construct $\mathbf{G}_\Pi$ from $\mathbf{G}_{\Pi^\wedge}$ and $\mathbf{G}_{\Pi^\vee}$ by merging the two graphs without any conflict. The following important properties of $\mathbf{G}_\Pi$ can be observed:

- A node in $\mathbf{G}_\Pi$ is called an And-node if it has only incoming solid edges. Similarly, a node in $\mathbf{G}_\Pi$ is an Or-node if it has only incoming dash edges.
- A node cannot have both types of incoming edges (it is not the case for outgoing edges). In other words, a node can only be either an And-node or an Or-node.
- From $\mathbf{G}_\Pi$, we can interpret that an And-node is **True** *iff* all original nodes of its incoming edges are **True**. Similarly, an Or-node is **True** *iff* at least one of the original nodes of its incoming edges is **True**.

By definition, the And-Or dependency graph can capture the semantical meaning of the original $\Pi^\wedge$ and $\Pi^\vee$. More importantly, a program $\Pi^\wedge$ and its dependency graph $\mathbf{G}_{\Pi^\wedge}$ (similar to the case of $\Pi^\vee$ and $\mathbf{G}_{\Pi^\vee}$) are related directly because the program matrix and the adjacency matrix of the dependency graph are equivalent. Note that if all non-zero elements are 1, the program matrix $\mathbf{M}_{\Pi^\wedge}$ is exactly the adjacency matrix of the dependency graph $\mathbf{G}_{\Pi^\wedge}$. However, to be consistent with the choice of normalizing rule body to define truth values in the previous section, we denote the adjacency matrix of $\mathbf{G}_{\Pi^\wedge}$ by $\theta^{\Uparrow}(\mathbf{M}_{\Pi^\wedge})$. Similarly, we denote $\theta^{\Uparrow}(\mathbf{M}_{\Pi^\vee})$ as the adjacency matrix of $\mathbf{G}_{\Pi^\vee}$.

4. Linear Algebraic Partial Evaluation

4.1. Partial evaluation with iteration method

Sakama et al. first proposed the idea of PE for computing least models of logic programs using linear algebra [22]. Later, a refined version of the method was published in [12]. Extending from this idea, Nguyen et al. have developed PE with *reduct abductive matrix* (Definition 5 in [13]) for Horn abduction [13]. The *reduct abductive matrix* is obtained by taking the *abductive matrix* (simply a transposed matrix of

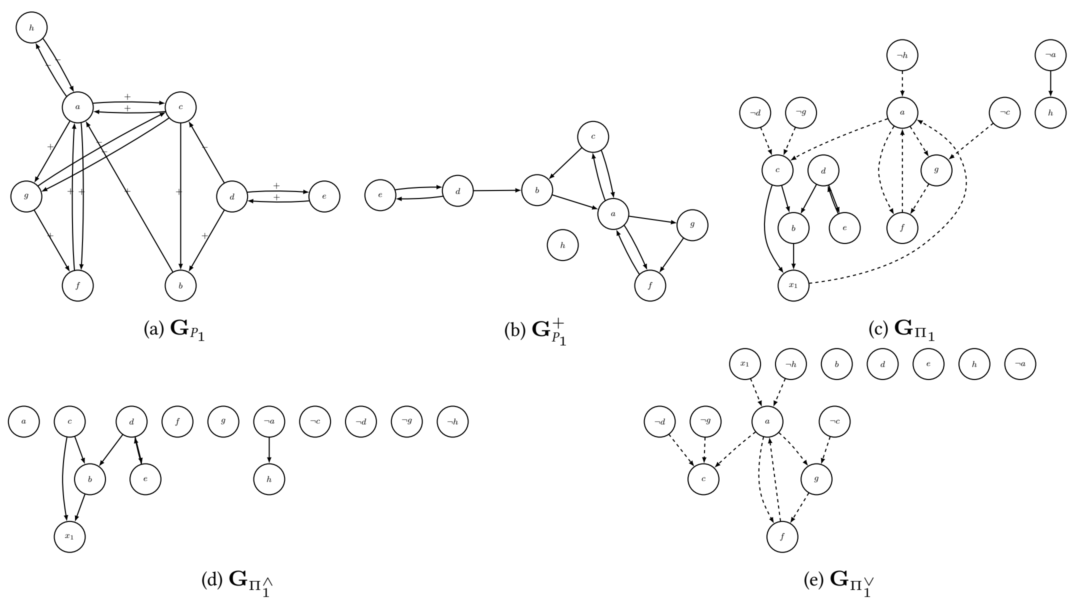

(a) $\mathbf{G}_{P_1}$
(b) $\mathbf{G}^{+}_{P_1}$
(c) $\mathbf{G}_{\Pi_1}$
(d) $\mathbf{G}_{\Pi_1^{\wedge}}$
(e) $\mathbf{G}_{\Pi_1^{\vee}}$

Figure 2: Illustrations of dependency graphs of the normal logic program P_1 and its standardized program Π_1 in Example 1.

$\mathbf{M}_\Pi$ - Definition 4 in [23]) then removing all columns w.r.t. Or-rules (4) and setting 1 at the diagonal corresponding to all atoms which are heads of these Or-rules. The basic idea can be simplified as we take $\mathbf{M}_{\Pi^\wedge}$ then append to the diagonal of $\mathbf{M}_{\Pi^\wedge}$ all atoms we want to preserve (Or-rule heads, facts, negations, ...) in the partially evaluated program. Then we take the resulting matrix to multiply with itself iteratively until a fixed point is reached. We formalize this idea in the following definitions.

Definition 5 (**Partial evaluation of *And*-rules**). *Given a normal logic program P, its standardized program is* Π. *The partial evaluated matrix of* Π *w.r.t.* And*-rules, denoted as* $\texttt{peval}(\Pi^\wedge)$*, is defined as follows:*

$$\begin{aligned}\widehat{\mathbf{M}}_{\Pi^\wedge} &= \mathbf{M}_{\Pi^\wedge} + \texttt{diag}\big(\mathbf{v}_{\Pi^F} \oplus_{\theta\Downarrow} \mathbf{v}_{neg(\Pi)} \oplus_{\theta\Downarrow} \mathbf{v}_{head(\Pi^\vee)}\big)\\ \mathbf{M}_0 &= \widehat{\mathbf{M}}_{\Pi^\wedge}\\ \mathbf{M}_i &= \mathbf{M}_{i-1}\cdot\mathbf{M}_{i-1} \qquad (i \geq 1) \end{aligned} \tag{6}$$

where $\mathbf{v}_{\text{head}(\Pi^\vee)}$ is a column vector such that $\mathbf{v}_{\text{head}(\Pi^\vee)}[i] = 1$ if the corresponding atom at index i is a head of an Or-rule.

Definition 6 (**Partial evaluation of *Or*-rules**). *Given a normal logic program P, its standardized program is* Π. *The partial evaluated matrix of* Π *w.r.t.* Or*-rules, denoted as* $\texttt{peval}(\Pi^\vee)$*, is defined as follows:*

$$\begin{aligned}\widehat{\mathbf{M}}_{\Pi^\vee} &= \mathbf{M}_{\Pi^\vee} + \texttt{diag}\big(\mathbf{v}_{\Pi^F} \oplus_{\theta\Downarrow} \mathbf{v}_{neg(\Pi)} \oplus_{\theta\Downarrow} \mathbf{v}_{head(\Pi^\wedge)}\big)\\ \mathbf{M}_0 &= \widehat{\mathbf{M}}_{\Pi^\vee}\\ \mathbf{M}_i &= \mathbf{M}_{i-1}\cdot\mathbf{M}_{i-1} \qquad (i \geq 1) \end{aligned} \tag{7}$$

where $\mathbf{v}_{\text{head}(\Pi^\wedge)}$ is a column vector such that $\mathbf{v}_{\text{head}(\Pi^\wedge)}[i] = 1$ if the corresponding atom at index i is a head of an And-rule.

Both Definition 5 and Definition 6 are almost identical except for the starting point with different matrices $\mathbf{M}_{\Pi^\wedge}$ and $\mathbf{M}_{\Pi^\vee}$ respectively. We say (6) and (7) reach a fixed point at a step k ($k \geq 1$) if $\mathbf{M}_k = \mathbf{M}_{k-1}$. Because the matrix multiplication performs unfolding rules [12], intuitively, the fixed point is reached when the program is fully unfolded. For the case of acyclic programs, it is guaranteed that the fixed point is reached after a finite step of iterations [23]. Proposition 1 shows the minimum number of PE steps to reach a fixed point for acyclic case.

Proposition 1. *For any program P with* $\mathbf{M}_{\Pi^\wedge}$ *(and* $\mathbf{M}_{\Pi^\vee}$*) of the size* $n \times n$ *such that the corresponding dependency graph* $\mathbf{G}_{\Pi^\wedge}$ *(and* $\mathbf{G}_{\Pi^\wedge}$*) is acyclic, the sufficient number of PE steps to reach a fixed point is* $k = \lceil log_2(n) \rceil$.

Proof. Consider the case with a program $P_2 = \{a_1 \leftarrow a_2, a_2 \leftarrow a_3, \ldots, a_{n-1} \leftarrow a_n, \}$. Obviously, this program has the longest dependency chain we can create from n atoms. Indeed, unfolding P_2 at the first step we have $\{a_1 \leftarrow a_3, a_2 \leftarrow a_4, a_3 \leftarrow a_5, \ldots, a_{n-1} \leftarrow a_n\}$, at the second step we have $\{a_1 \leftarrow a_5, a_2 \leftarrow a_6, a_3 \leftarrow a_7 \ldots, a_{n-1} \leftarrow a_n\}$, and so on. According to the pattern, if we perform the PE for k steps, then the condition of the fixed point is reached when $2^k \geq n \Leftrightarrow k \geq log_2(n)$. k is an integer, so we have $k = \lceil log_2(n) \rceil$. The proof is identical for the case of $\Pi^\vee$. □

At a fixed point, we can also compute $\mathbf{M}_k = (\widehat{\mathbf{M}}_{\Pi^\wedge})^{2^k}$ ($k \geq 1$) (or $\mathbf{M}_k = (\widehat{\mathbf{M}}_{\Pi^\vee})^{2^k}$ ($k \geq 1$) for the case of Or-rules) that is basically computing powers of a matrix. Then, we define $\texttt{peval}(\Pi^\wedge) = unpack((\widehat{\mathbf{M}}_{\Pi^\wedge})^{2^k})$ is the partially evaluated program of $\Pi^\wedge$, where $unpack((\widehat{\mathbf{M}}_{\Pi^\wedge})^{2^k})$ is a series of actions including: (**s1**) reversing the effect of appending $\mathbf{v}_{\Pi^F} \oplus_{\theta\Downarrow} \mathbf{v}_{\text{neg}(\Pi)} \oplus_{\theta\Downarrow} \mathbf{v}_{\text{head}(\Pi^\vee)}$ to the diagonal, (**s2**) removing all row r if the sum of non-zero elements on that row in $(\widehat{\mathbf{M}}_{\Pi^\wedge})^{2^k}$ is less than 1, and (**s3**) normalizing non-zero elements of $(\widehat{\mathbf{M}}_{\Pi^\wedge})^{2^k}$ to satisfy Definition 1. Step (**s2**) is important as an And-node is **True** only if all its body atoms are **True**. Similarly, we define $\texttt{peval}(\Pi^\vee) = unpack((\widehat{\mathbf{M}}_{\Pi^\vee})^{2^k})$ is the partially evaluated program of $\Pi^\vee$, where $unpack((\widehat{\mathbf{M}}_{\Pi^\vee})^{2^k})$ is a series of actions including: (**s1**) reversing the effect of appending $\mathbf{v}_{\Pi^F} \oplus_{\theta\Downarrow} \mathbf{v}_{\text{neg}(\Pi)} \oplus_{\theta\Downarrow} \mathbf{v}_{\text{head}(\Pi^\wedge)}$ to the diagonal, and (**s2**) normalizing non-zero elements of $(\widehat{\mathbf{M}}_{\Pi^\vee})^{2^k}$ to satisfy

Definition 1. $\texttt{peval}(\Pi^{\wedge})$ and $\texttt{peval}(\Pi^{\vee})$ are introduced to simplify the notation in the following sections.

We have presented the basic idea of linear algebraic PE of logic programs through iteratively compute powers of matrix ($\widehat{\mathbf{M}}_{\Pi^{\wedge}}$ and $\widehat{\mathbf{M}}_{\Pi^{\vee}}$) until a fixed point is reached. However, a fixed point is not guaranteed in case there is a cycle in the corresponding dependency graph ($\mathbf{G}_{\Pi^{\wedge}}$ or $\mathbf{G}_{\Pi^{\vee}}$ respectively). For example, consider the visualization of P_1 in Figure 2 where $\mathbf{G}_{\Pi_1^{\wedge}}$ has a cycle $\{d, e\}$ while $\mathbf{G}_{\Pi_1^{\vee}}$ has two cycles $\{a, f\}$ and $\{a, f, g\}$. In this example, (6) and (7) cannot reach a fixed point, consequently $\texttt{peval}(\Pi_1^{\wedge})$ and $\texttt{peval}(\Pi_1^{\vee})$ cannot be computed. In the next section, we will introduce cycle-resolving techniques to ensure that this method also works even with cycles in $\mathbf{G}_{\Pi^{\wedge}}$ and $\mathbf{G}_{\Pi^{\vee}}$.

4.2. Cycle resolving

First, we define the *local cycle* in $\mathbf{G}_{\Pi^{\wedge}}$ and $\mathbf{G}_{\Pi^{\vee}}$.

Definition 7 (Local cycle in $\mathbf{G}_{\Pi^{\wedge}}$ and $\mathbf{G}_{\Pi^{\vee}}$). *Given a normal logic program P, its standardized program is* Π. *A set L of atoms is called a local cycle in $\mathbf{G}_{\Pi^{\wedge}}$ (or $\mathbf{G}_{\Pi^{\vee}}$) if L is strongly connected in $\mathbf{G}_{\Pi^{\wedge}}$ (or $\mathbf{G}_{\Pi^{\vee}}$).*

The term *local cycle* is used to distinguish from the general concept of a cycle in $\mathbf{G}_{\Pi}$. For example in Figure 2, there are *cycles mixing both solid and dash edges* at the same time such as $\{a, c, x_1\}$. These are not (yet) the target of our cycle-resolving techniques in this paper. Our main focus is to resolve the local cycles in $\mathbf{G}_{\Pi^{\wedge}}$ and $\mathbf{G}_{\Pi^{\vee}}$ such as $\{d, e\}$ in $\mathbf{G}_{\Pi_1^{\wedge}}$; and $\{a, f\}$, $\{a, f, g\}$ in $\mathbf{G}_{\Pi_1^{\vee}}$. We can easily identify the local cycles in $\mathbf{G}_{\Pi^{\wedge}}$ and $\mathbf{G}_{\Pi^{\vee}}$ by identifying every Strongly Connected Component (SCC) in $\mathbf{G}_{\Pi^{\wedge}}$ and $\mathbf{G}_{\Pi^{\vee}}$ respectively. This can be done in polynomial time by applying the *Tarjan's algorithm* [24] or using the algorithm in [25] which can be implemented in linear algebraic way using GraphBLAS[1] [26].

After identifying the local cycles, let us consider how to resolve them. The main idea is to translate a cycle into a set of rules preserving the same logical meaning but does not create cyclic computation in Definition 5 and Definition 6. This should be done differently for And-rules and Or-rules. For a cycle L in $\mathbf{G}_{\Pi^{\wedge}}$, obviously, there is no way to make an And-node in L become **True** other than the cycle L itself. On the other hand, for a cycle L in $\mathbf{G}_{\Pi^{\vee}}$, we can make an Or-node in L become **True** if there is any body literal (outside from the cycle L) of that rule which is **True**. Accordingly, we propose the following cycle-resolving techniques for And-rules and Or-rules respectively.

Algorithm 1: Cycle-resolving for And-rules

```
1: Identify all SCCs in $\mathbf{G}_{\Pi^{\wedge}}$.
2: for each SCC $L$ in $\mathbf{G}_{\Pi^{\wedge}}$ do
3:     for each rule $r \in \Pi^{\wedge}$ such that $head(r) \in L$ do
4:         Remove $r$ (by setting the corresponding entries of $r$ in $\mathbf{M}_{\Pi^{\wedge}}$ to 0).
```

Algorithm 2: Cycle-resolving for Or-rules

```
1: Identify all SCCs in $\mathbf{G}_{\Pi^{\vee}}$.
2: for each SCC $L$ in $\mathbf{G}_{\Pi^{\vee}}$ do
3:     Let $E = \emptyset$
4:     for each rule $r \in \Pi^{\vee}$ such that $head(r) \in L$ do
5:         $E = E \cup (body(r) \setminus L)$
6:     for each rule $r \in \Pi^{\vee}$ such that $head(r) \in L$ do
7:         Replace $r$ by $head(r) \leftarrow \bigvee_{q \in E} q$.
```

After resolving the cycles, we can apply the linear algebraic PE of And-rules and Or-rules as described in Definition 5 and Definition 6 respectively. Now we can prove that the computation has a fixed point.

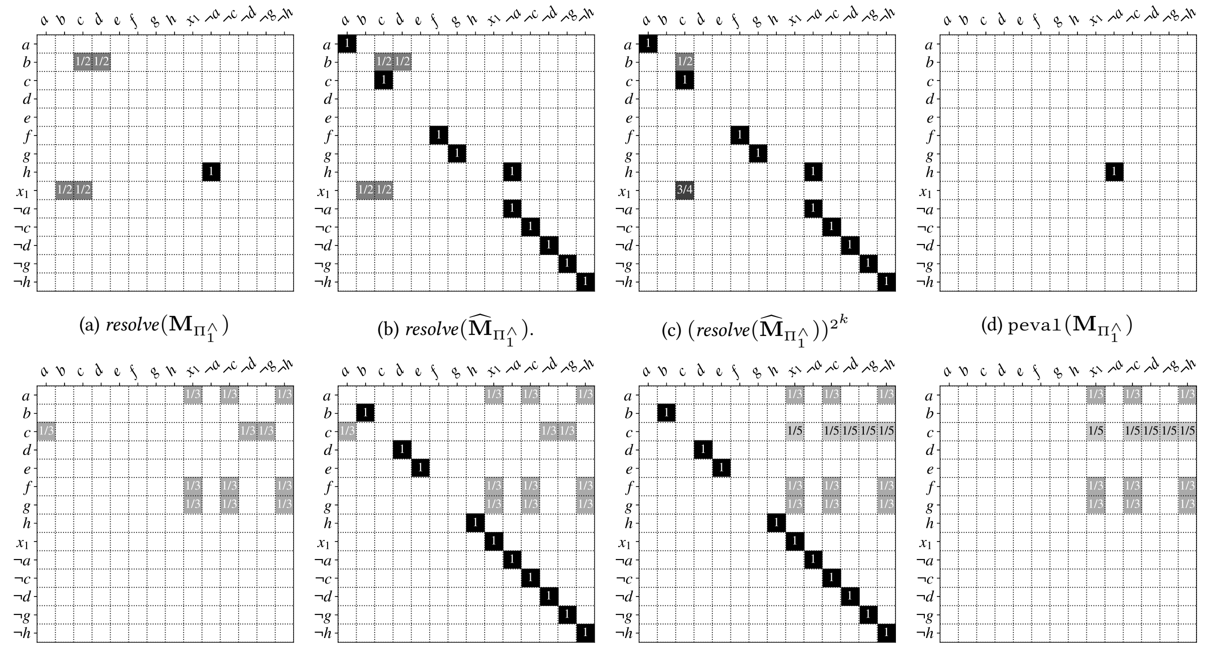
Proposition 2. *Given a resolved matrix resolve*$(\mathbf{M}_{\Pi^{\wedge}})$ *(or resolve*$(\mathbf{M}_{\Pi^{\vee}})$*) as input for the linear algebraic PE of And-rules (or Or-rules), the fixed point is guaranteed to be reached after a finite number of iterations.*

Proof. There are two cases:
For the case of And-rules, all cycles in $\mathbf{G}_{\Pi^{\wedge}}$ are removed. Hence, this case is identical to the case of acyclic programs in that the computation in Definition 5 reaches a fixed point after a finite number of iterations.
For the case of Or-rules, cycles still exist in $\mathbf{G}_{\Pi^{\vee}}$ but all Or-rules such that their head nodes are in the cycle are updated in a way that they have incoming edges from all body literals related to a cycle but excluding the cycle itself. This ensures that all possible ways to make an Or-node in a cycle become **True** are considered, so no new cases are created during the computation in Definition 6. Thus, a fixed point is guaranteed. □

Figure 3 demonstrates the linear algebraic PE of $\Pi_1^{\wedge}$ and $\Pi_1^{\vee}$ after resolving the cycles. We denote *resolve*$(\mathbf{M}_{\Pi_1^{\wedge}})$ and *resolve*$(\mathbf{M}_{\Pi_1^{\vee}})$ as the matrix representation of $\Pi_1^{\wedge}$ and $\Pi_1^{\vee}$ after applying Algorithm 1 and Algorithm 2 respectively.
For the case of And-rules, there is a cycle $\{d, e\}$ corresponding to two And-rules $d \leftarrow e$ and $e \leftarrow d$. To resolve the cycle, we simply remove it as illustrated in Figure 3a following Algorithm 1. After all cycles are resolved, it is guaranteed that the iteration method can reach a fixed point when computing $(resolve(\widehat{\mathbf{M}}_{\Pi_1^{\wedge}}))^{2^k}$ to obtain $\texttt{peval}(\Pi_1^{\wedge})$. Figure 4a visualizes the dependency graph of $\texttt{peval}(\Pi_1^{\wedge})$.
For the case of Or-rules, there are 2 cycles $\{a, f\}$ and $\{a, f, g\}$. They all belong to a single SCC. Hence, we only need to resolve $\{a, f, g\}$ corresponding to three Or-rules: $a \leftarrow x_1 \vee f \vee \neg h$, $f \leftarrow a \vee g$, $g \leftarrow a \vee \neg c$. Following Algorithm 2, we find $E = \{\neg c, \neg h, x_1\}$. Next, we reset all Or-rules corresponding to a, f, and g with the new body $\{\neg c, \neg h, x_1\}$. The resulting matrix *resolve*$(\mathbf{M}_{\Pi_1^{\vee}})$ is illustrated in Figure 3e. Unlike the case of And-rules where we remove the cycle, here we find all possibilities to make the cycle become **True** then update the rules accordingly. After all cycles are resolved, we can apply the iteration method described in Definition 6 to compute $\texttt{peval}(\Pi_1^{\vee})$. Figure 4b visualizes the dependency graph of $\texttt{peval}(\Pi_1^{\vee})$.

Combining $\texttt{peval}(\mathbf{M}_{\Pi^{\wedge}})$ and $\texttt{peval}(\mathbf{M}_{\Pi^{\vee}})$ To sum up, we have presented the basic idea of linear algebraic PE. We have also introduced cycle-resolving techniques to ensure that this method also works effectively even with cycles in $\mathbf{G}_{\Pi^{\wedge}}$ and $\mathbf{G}_{\Pi^{\vee}}$. Finally, we can construct the partially

[1]GraphBLAS is an open-source API specification which defines standard building blocks for graph algorithms in the language of linear algebra.



(a) $\mathit{resolve}(\mathbf{M}_{\Pi_1^\wedge})$ (b) $\mathit{resolve}(\widehat{\mathbf{M}}_{\Pi_1^\wedge})$. (c) $(\mathit{resolve}(\widehat{\mathbf{M}}_{\Pi_1^\wedge}))^{2^k}$ (d) $\mathtt{peval}(\mathbf{M}_{\Pi_1^\wedge})$

(e) $\mathit{resolve}(\mathbf{M}_{\Pi_1^\vee})$ (f) $\mathit{resolve}(\widehat{\mathbf{M}}_{\Pi_1^\vee})$. (g) $(\mathit{resolve}(\widehat{\mathbf{M}}_{\Pi_1^\vee}))^{2^k}$ (h) $\mathtt{peval}(\mathbf{M}_{\Pi_1^\vee})$

Figure 3: Visualization of the linear algebraic PE of $\Pi_1^\wedge$ (upper) and $\Pi_1^\vee$ (lower).

evaluated program matrix for logic inferencing in vector spaces by combining $\mathtt{peval}(\mathbf{M}_{\Pi^\wedge})$ and $\mathtt{peval}(\mathbf{M}_{\Pi^\vee})$:

$$\begin{aligned}\mathtt{peval}(\mathbf{M}_\Pi) = \quad & \mathtt{peval}(\Pi^\wedge) + \theta^{\Uparrow}\big(\mathtt{peval}(\Pi^\vee)\big) \\ & + \mathtt{diag}\big(\mathbf{v}_{\Pi^F} \oplus_{\theta\Downarrow} \mathbf{v}_{\mathrm{neg}(\Pi)}\big) \qquad (8)\end{aligned}$$

$\mathtt{peval}(\mathbf{M}_\Pi)$ can be used for the fixpoint computation in the same way as $\mathbf{M}_\Pi$. A few modifications may be needed to apply the idea to Horn abduction in [13], however, the main idea remains the same. $\mathtt{peval}(\mathbf{M}_\Pi)$ is expected to be more efficient than $\mathbf{M}_\Pi$ in case it helps to reduce the number of deduction steps to reach a fixpoint. Figure 4 illustrates the visualization of $\mathtt{peval}(\mathbf{M}_{\Pi_1})$ after combining $\mathtt{peval}(\mathbf{M}_{\Pi_1^\wedge})$ and $\mathtt{peval}(\mathbf{M}_{\Pi_1^\vee})$ with dependency graphs and matrix representations.

5. Partial evaluation using matrix decomposition

5.1. Eigendecomposition

As we have seen in the previous sections, the main idea of PE is to compute the powers of a program matrix. While in linear algebra, it is known that powers of a matrix $\mathbf{M}$ can be computed efficiently using its eigendecomposition $\mathbf{M} = \mathbf{Q} \cdot \mathbf{\Lambda} \cdot \mathbf{Q}^{-1}$, where $\mathbf{\Lambda}$ is a diagonal matrix of eigenvalues and $\mathbf{Q}$ is a matrix of eigenvectors [27]. Then we can compute $\mathbf{M}^n = \mathbf{Q} \cdot \mathbf{\Lambda}^n \cdot \mathbf{Q}^{-1}$ that is computationally more efficient than computing $\mathbf{M}^n$ directly, because $\mathbf{\Lambda}^n$ is just the element-wise power of the diagonal matrix $\mathbf{\Lambda}$.

In this section, we will show how to apply eigendecomposition to realize PE in LP. Let us consider an example:

Example 2. *Given a logic program $P_3 = \{p \leftarrow p \wedge q,\ q \leftarrow q \wedge r,\ r \leftarrow q\}$. Standardized logic program (no change): $\Pi_3 = P_3$.*

There is no Or-rule in Π_3, so we only need to consider $\mathbf{M}_{\Pi_3^\wedge} = \begin{array}{c|ccc} & p & q & r \\ \hline p & 1/2 & 1/2 & \\ q & & 1/2 & 1/2 \\ r & & 1 & \end{array}$. For computing the eigenvalues, it is more numerically stable to use the adjacency matrix $\theta^{\Uparrow}(\mathbf{M}_{\Pi_3^\wedge}) = \begin{array}{c|ccc} & p & q & r \\ \hline p & 1 & 1 & \\ q & & 1 & 1 \\ r & & 1 & \end{array}$ instead of $\mathbf{M}_{\Pi_3^\wedge}$. Next, we append needed information to the diagonal to obtain $\theta^{\Uparrow}(\widehat{\mathbf{M}}_{\Pi_3^\wedge})$, here they are identical. Let us compute the eigenvalues of $\theta^{\Uparrow}(\widehat{\mathbf{M}}_{\Pi_3^\wedge})$.

$$\begin{aligned} & det(\theta^{\Uparrow}(\widehat{\mathbf{M}}_{\Pi_3^\wedge}) - \lambda \mathbf{I}) && = 0 \\ \Leftrightarrow \quad & \begin{array}{c|ccc} & p & q & r \\ \hline p & 1-\lambda & 1 & \\ q & & 1-\lambda & 1 \\ r & & 1 & \end{array} && = 0 \\ \Leftrightarrow \quad & -\lambda^3 + 2\lambda^2 - 1 && = 0 \\ \Leftrightarrow \quad & (\lambda - 1)(\lambda^2 - \lambda - 1) && = 0 \end{aligned}$$

Eigenvalues:

$$\lambda_1 = 1$$

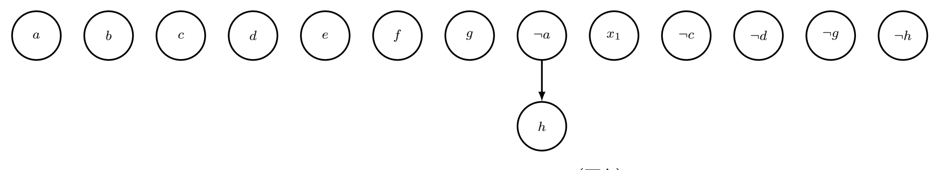
$$\lambda_2 = \frac{1}{2}(1 + \sqrt{5})$$

$$\lambda_3 = \frac{1}{2}(1 - \sqrt{5})$$

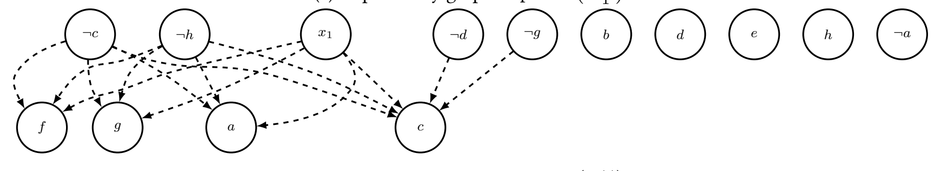
Eigenvectors:

$$v_1 = (\frac{1}{2}(3 + \sqrt{5}), \frac{1}{2}(1 + \sqrt{5}), 1)$$

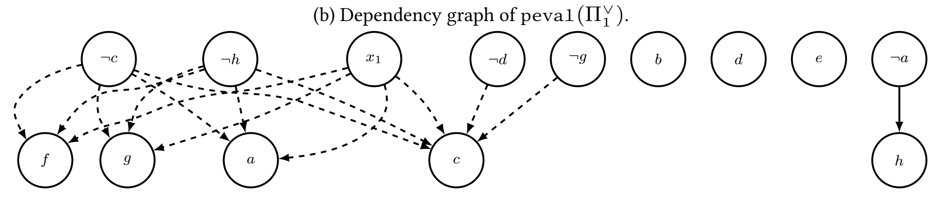
$$v_2 = (1, 0, 0)$$



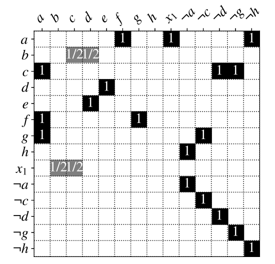
(a) Dependency graph of $\texttt{peval}(\Pi_1^{\wedge})$.



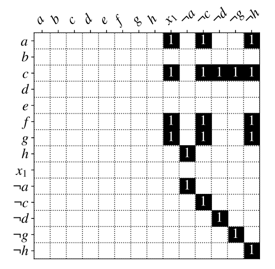
(b) Dependency graph of $\texttt{peval}(\Pi_1^{\vee})$.



(c) *And-Or*-dependency graph of $\texttt{peval}(\Pi_1)$.



(d) $\mathbf{M}_{\Pi_1}$



(e) $\texttt{peval}(\mathbf{M}_{\Pi_1})$

Figure 4: Visualization of partial evaluated dependency graphs of Π_1, the program matrix $\mathbf{M}_{\Pi_1}$ and the partially evaluated program matrix $\texttt{peval}(\mathbf{M}_{\Pi_1})$.

$$v_3 = (\frac{1}{2}(3-\sqrt{5}), \frac{1}{2}(1-\sqrt{5}), 1)$$

Eigendecomposition:

$$\theta^{\Uparrow}(\widehat{\mathbf{M}}_{\Pi_3^{\wedge}}) = \begin{array}{c|ccc} & p & q & r \\ \hline p & 1 & 1 & \\ q & & 1 & 1 \\ r & & 1 & \end{array} = \mathbf{Q}\cdot\mathbf{A}\cdot\mathbf{Q}^{-1}$$

where:

$$\mathbf{A} = \begin{array}{c|ccc} & p & q & r \\ \hline p & 1 & & \\ q & & \frac{1}{2}(1+\sqrt{5}) & \\ r & & & \frac{1}{2}(1-\sqrt{5}) \end{array}$$

$$\mathbf{Q} = \begin{array}{c|ccc} & p & q & r \\ \hline p & \frac{1}{2}(3+\sqrt{5}) & \frac{1}{2}(1+\sqrt{5}) & 1 \\ q & 1 & 0 & 0 \\ r & \frac{1}{2}(3-\sqrt{5}) & \frac{1}{2}(1-\sqrt{5}) & 1 \end{array}$$

When we obtain the eigendecomposition of $\theta^{\Uparrow}(\widehat{\mathbf{M}}_{\Pi_3^{\wedge}})$, we can compute powers of $\theta^{\Uparrow}(\widehat{\mathbf{M}}_{\Pi_3^{\wedge}})$ efficiently. However, unlike the iteration method in which we let the method determine a fixpoint condition, here we need to determine the power n in advance. Fortunately, we can set a sufficiently large n to ensure that the fixpoint is reached following Proposition 1. In this example, as $n = 3$, we have a sufficient number of iterations to reach the fixpoint $k = \lceil log_2(3) - 1\rceil = 1$, then we just need to raise $\mathbf{A}$ to the power of $k+1 = 2$. Accordingly, the partial evaluated matrix is:

$$\mathbf{Q}\cdot\mathbf{A}^2\cdot\mathbf{Q}^{-1} = \begin{array}{c|ccc} & p & q & r \\ \hline p & 1 & 2 & 1 \\ q & 0 & 2 & 1 \\ r & 0 & 1 & 1 \end{array}$$

This matrix can be translated into a logic program: $\texttt{peval}(\mathbf{M}_{\Pi_3^{\wedge}}) = \{p \leftarrow p \wedge q \wedge r,\ q \leftarrow q \wedge r,\ r \leftarrow q \wedge r\}$ which is the partial evaluated program of $\Pi_3^{\wedge}$. Because there is no *Or*-rule in this case, so $\texttt{peval}(\mathbf{M}_{\Pi_3^{\wedge}})$ is also the partially evaluated program of Π_3.

Using eigendecomposition for partial evaluation is computationally more efficient than the iteration method, especially when the number of iterations is large. However, the eigendecomposition method requires the matrix to be diagonalizable [27], which is not always the case. Unfortunately for the case of program matrices, we usually see that the determinant of the matrix is 0, which means that the matrix is not diagonalizable. In such cases, we can use the Jordan Normal Form (JNF) to compute the powers of a matrix that we will discuss in the next section.

5.2. Jordan normal form

In linear algebra, the JNF, also known as the Jordan canonical form, is a specific type of upper triangular matrix called a Jordan matrix.

Definition 8 (Jordan normal form [28]). *Let J_i be a square $k \times k$ matrix* $\begin{pmatrix} \lambda_i & 1 & & & \\ & \lambda_i & 1 & & \\ & & \ddots & \ddots & \\ & & & \lambda_i & 1 \\ & & & & \lambda_i \end{pmatrix}$ *such that λ_i is identical on the diagonal and there are 1s just above the diagonal. We call each such matrix a Jordan λ_i-block. A matrix $\mathbf{M}$ is in JNF if*

$$\mathbf{M} = \begin{pmatrix} J_1 & & & \\ & J_2 & & \\ & & \ddots & \\ & & & J_p \end{pmatrix}$$

It is proved that every square matrix in $\mathbb{R}^{n\times n}$ can be decomposed into a matrix in JNF [29] according to Jordan's theorem. Additionally, computing powers of a Jordan matrix $\mathbf{M}$ is straightforward:

$$\mathbf{M}^n = \begin{pmatrix} J_1 & & & \\ & J_2 & & \\ & & \ddots & \\ & & & J_p \end{pmatrix}^n = \begin{pmatrix} (J_1)^n & & & \\ & (J_2)^n & & \\ & & \ddots & \\ & & & (J_p)^n \end{pmatrix}$$

that can be simplified by computing powers of each Jordan block. The power of a Jordan block J_i of the size $k \times k$ can be computed by:

$$(J_i)^n = \begin{pmatrix} \lambda_i^n & \binom{n}{1}\lambda_i^{n-1} & \binom{n}{2}\lambda_i^{n-2} & \dots & \dots & \binom{n}{k-1}\lambda_i^{n-k+1} \\ & \lambda_i^n & \binom{n}{1}\lambda_i^{n-1} & \dots & \dots & \binom{n}{k-2}\lambda_i^{n-k+2} \\ & & \ddots & \dots & \dots & \vdots \\ & & & \ddots & \dots & \vdots \\ & & & & \lambda_i^n & \binom{n}{1}\lambda_i^{n-1} \\ & & & & & \lambda_i^n \end{pmatrix}$$

where $\binom{n}{b}$ is the binomial coefficient describing the algebraic expansion of powers of a binomial.

Now let us consider an example to illustrate the idea of using JNF for partial evaluation.

Example 3. *Given a normal logic program:* $P_4 = \{p \leftarrow q,\ p \leftarrow r,\ q \leftarrow s,\ q \leftarrow t,\ r \leftarrow \neg t,\ r \leftarrow \neg s,\ s \leftarrow \neg t,\ s \leftarrow \neg r,\ t \leftarrow \neg r,\ t \leftarrow \neg s\}$.
Standardized logic program: $\Pi_4 = \{p \leftarrow q \vee r,\ q \leftarrow s \vee t,\ r \leftarrow \neg t \vee \neg s,\ s \leftarrow \neg t \vee \neg r,\ t \leftarrow \neg s \vee \neg r\}$.

There is no And-rules, so we only need to consider $\widehat{\mathbf{M}}_{\Pi_4^\vee}$. The matrix is not diagonalizable as $det(\widehat{\mathbf{M}}_{\Pi_4^\vee}) = 0$, so we will use the JNF to compute powers of $\widehat{\mathbf{M}}_{\Pi_4^\vee}$. First, let us compute the eigenvalues of $\theta^{\Uparrow}(\widehat{\mathbf{M}}_{\Pi_4^\vee})$:

$$\theta^{\Uparrow}(\widehat{\mathbf{M}}_{\Pi_4^\vee}) = \begin{array}{c|cccccccc} & p & q & r & s & t & \neg r & \neg s & \neg t \\ \hline p & & 1 & 1 & & & & & \\ q & & & & 1 & 1 & & & \\ r & & & & & & & 1 & 1 \\ s & & & & & & 1 & & 1 \\ t & & & & & & 1 & 1 & \\ \neg r & & & & & & 1 & & \\ \neg s & & & & & & & 1 & \\ \neg t & & & & & & & & 1 \end{array}$$

$$\begin{aligned} & det(\theta^{\Uparrow}(\widehat{\mathbf{M}}_{\Pi_4^\vee}) - \lambda \mathbf{I}) && = 0 \\ \Leftrightarrow\quad & \lambda^8 - 3\lambda^7 + 3\lambda^6 - \lambda^5 && = 0 \\ \Leftrightarrow\quad & \lambda^5(\lambda - 1)^3 && = 0 \end{aligned}$$

1. $\lambda_1 = 0$, algebraic multiplicity[2] 5, eigenvectors:

$$\begin{aligned} v_1 &= (1, 0, 0, 0, 0, 0, 0, 0)^\top \\ v_2 &= (0, -1, 1, 0, 0, 0, 0, 0)^\top \\ v_3 &= (0, 0, 0, -1, 1, 0, 0, 0)^\top \end{aligned}$$

For v_1, solve $(\theta^{\Uparrow}(\widehat{\mathbf{M}}_{\Pi_4^\vee}) - \lambda_1 \mathbf{I})^r \cdot v_1 = 0$:

2. $\lambda_2 = 1$, algebraic multiplicity 3, eigenvectors:

$$\begin{aligned} v_4 &= (2, 2, 0, 1, 1, 1, 0, 0)^\top \\ v_5 &= (2, 1, 1, 0, 1, 0, 1, 0)^\top \\ v_6 &= (2, 1, 1, 1, 0, 0, 0, 1)^\top \end{aligned}$$

Following the algorithm described in [30], one can find the JNF of $\theta^{\Uparrow}(\widehat{\mathbf{M}}_{\Pi_4^\vee}) = \mathbf{P} \cdot \mathbf{J} \cdot \mathbf{P}^{-1}$ where:

$$\mathbf{J} = \begin{array}{c|ccc:c:c:c:c:c} & p & q & r & s & t & \neg r & \neg s & \neg t \\ \hline p & \boxed{0} & \boxed{1} & & & & & & \\ q & & \boxed{0} & \boxed{1} & & & & & \\ r & & & \boxed{0} & & & & & \\ \hdashline s & & & & \boxed{0} & & & & \\ \hdashline t & & & & & \boxed{0} & & & \\ \hdashline \neg r & & & & & & \boxed{1} & & \\ \hdashline \neg s & & & & & & & \boxed{1} & \\ \hdashline \neg t & & & & & & & & \boxed{1} \end{array}$$

$$\mathbf{P} = \begin{array}{c|ccc:c:c:c:c:c} & p & q & r & s & t & \neg r & \neg s & \neg t \\ \hline p & \boxed{1} & & & & & \boxed{2} & \boxed{2} & \boxed{2} \\ q & & \boxed{1} & & \boxed{-1} & & \boxed{2} & \boxed{1} & \boxed{1} \\ r & & & & \boxed{1} & & & \boxed{1} & \boxed{1} \\ s & & & \boxed{1} & & \boxed{-1} & \boxed{1} & & \boxed{1} \\ t & & & & & \boxed{1} & \boxed{1} & \boxed{1} & \\ \neg r & & & & & & \boxed{1} & & \\ \neg s & & & & & & & \boxed{1} & \\ \neg t & & & & & & & & \boxed{1} \end{array}$$

For visualization purpose, we highlight total 6 Jordan blocks of $\mathbf{J}$ in different colors corresponding their eigenvectors v_1, v_2, v_3, v_4, v_5, v_6.

Similar to the eigendecomposition, we can compute the partial evaluated matrix $\texttt{peval}(\mathbf{M}_{\Pi_4^\vee})$ by computing $\mathbf{P} \cdot \mathbf{J}^k \cdot \mathbf{P}^{-1}$. For this example, $k = 4$ is sufficient to reach the fixpoint according to Proposition 1.

$$\mathbf{P} \cdot \mathbf{J}^4 \cdot \mathbf{P}^{-1} = \begin{array}{c|cccccccc} & p & q & r & s & t & \neg r & \neg s & \neg t \\ \hline p & & & & & & 2 & 2 & 2 \\ q & & & & & & 2 & 1 & 1 \\ r & & & & & & & 1 & 1 \\ s & & & & & & 1 & & 1 \\ t & & & & & & 1 & 1 & \\ \neg r & & & & & & 1 & & \\ \neg s & & & & & & & 1 & \\ \neg t & & & & & & & & 1 \end{array}$$

This matrix can be translated to: $\texttt{peval}(\mathbf{M}_{\Pi_4^\vee}) = \{p \leftarrow \neg r \vee \neg s \vee \neg t,\ q \leftarrow \neg r \vee \neg s \vee \neg t,\ r \leftarrow \neg s \vee \neg t,\ s \leftarrow \neg r \vee \neg t,\ t \leftarrow \neg r \vee \neg s\}$ is the partial evaluated program of $\Pi_4^\vee$. $\texttt{peval}(\mathbf{M}_{\Pi_4^\vee})$ is also identical to the partial evaluated program of Π_4 as there is no And-rule in this case.

General approach using matrix decomposition To summarize this section, we have shown how to use eigendecomposition and JNF to compute the powers of a matrix in the context of PE of logic programs. A baseline step-by-step is as follows:

[2]The algebraic multiplicity of an eigenvalue is the number of times it appears as a root of the characteristic polynomial.

Table 1
Statistical data of the datasets and detailed comparison of execution time (in *ms*) of the linear algebraic PE methods on the datasets. (green - best, red - worst)

	Artificial samples I (166 instances)		**Artificial samples II** (118 instances)		**FMEA samples** (213 instances)	
Parameters	mean / std	[min, max]	mean / std	[min, max]	mean / std	[min, max]
Matrix size	2088.32 / 1584.48	[11, 6601]	321.86 / 252.64	[18, 1110]	27.58 / 19.32	[6, 84]
No. *And*-rules	1188.63 / 1349.59	[8, 6375]	201.86 / 186.64	[9, 1007]	16.10 / 9.23	[1, 43]
No. *Or*-rules	899.69 / 839.58	[3, 3345]	119.99 / 107.40	[4, 437]	11.48 / 11.01	[1, 41]
Sparsity (of $\mathbf{M}_\Pi$)	0.99 / 0.02	[0.90, 1.00]	0.99 / 0.01	[0.90, 1.00]	0.95 / 0.04	[0.73, 0.99]
Longest path	4.63 / 5.36	[2, 65]	6.56 / 8.56	[2, 58]	1.94 / 0.24	[1, 2]
`peval` steps	3.78 / 0.95	[2, 5]	3.71 / 0.81	[2, 6]	2.00 / 0.00	[2, 2]
Algorithms	mean / std	Timeout?	mean / std	Timeout?	mean / std	Timeout?
(I) Iteration + dense	799 965 / 58 500	0 / 166	4483 / 688	0 / 118	103 / 10	0 / 213
(II) Decomposition + dense	9 292 159 / 34 274	152 / 166	6 041 323 / 28 710	96 / 118	1 607 397 / 19 170	18 / 213
(I) Iteration + sparse	545 / 15	0 / 166	138 / 4	0 / 118	157 / 5	0 / 213

```
Algorithm 3: Partial evaluation using matrix decomposition
1: Find the standardized program and its matrix
   representation $\mathbf{M}_{\Pi^\wedge}$ and $\mathbf{M}_{\Pi^\vee}$.
2: Resolve cycles in these matrices.
3: For each matrix $\widehat{\mathbf{M}}_{\Pi^\wedge}$ and $\widehat{\mathbf{M}}_{\Pi^\vee}$, compute the
   eigenvalues and eigenvectors.
4: if the matrix is diagonalizable then
5:     find the eigendecomposition of the matrix.
6: else
7:     find the Jordan normal form of the matrix.
8: Compute the power using the decomposition.
9: Translate resulting matrices back to a logic program.
```

6. Experimental Results

We focus on evaluating the performance of the proposed linear algebraic PE with iteration method (I) and the matrix decomposition method (II) using the logic programs in Failure Modes and Effects Analysis (FMEA) benchmarks [31] that also has been reported in [13]. Note that we only measure the time for partial evaluation computation (`peval` for short) *not including the time for solving the abduction problem*. The dataset consists of three problem sets: **Artificial samples I** (166 instances), **Artificial samples II** (118 instances), and **FMEA samples** (213 instances). All programs in the dataset are acyclic. We also augment the FMEA benchmarks with cycles to evaluate the performance in the cyclic case. The augmented benchmarks are generated by adding randomly 1-5 cycles of the length 2-5 to each $\mathbf{G}_{\Pi^\wedge}$ and $\mathbf{G}_{\Pi^\vee}$ of a program P. Algorithms to be compared are: (I) in dense matrix format, (I) in sparse (Compressed Sparse Row (CSR)) matrix format, and (II) in dense matrix format. Our code is implemented in Python 3.7 using `numpy`, `scipy`, and `sympy` for matrices representation and computation. We set a time out of $20s$ for PE computation, if a method takes longer than that, we report it as a timeout and its execution is set to 60s for comparison. System environment: Intel(R) Xeon(R) Bronze 3106 @1.70GHz; 64GB DDR3 @1333MHz; Ubuntu 22.04 LTS 64bit.

Table 1 reports the statistical data of the datasets and a detailed comparison of the execution time of the proposed algorithms. It can be seen that (I) is the fastest on all datasets while (II) is significantly slower. Table 2 reports the comparison for the cyclic case. In this case, we also report the execution time for the cycle-resolving step (`resolve` for short). `peval` + `resolve` is the total run time for this case. Augmented cycles do not change much the structure of the dataset, so the comparison is similar to the acyclic case.

The reason for (II) being slow in our benchmark is that all program matrices in the benchmarks are not diagonalizable, and Algorithm 3 must call `sympy` for JNF decomposition. As `sympy` is meant for symbolic computation, it can only handle matrices of up to 100 atoms in a reasonable time. For program matrices of this size, according to Proposition 1, (I) can reach a fixed point in a few iterations, and then it dominates (II). JNF is also known to be numerically unstable that is a small perturbation in the input matrix can lead to a large change in the Jordan form [32]. This leads to the low adoption of JNF in API libraries for numerical computation that we cannot find an available one in sparse format.

In general, (I) is the best choice for linear algebraic PE in practice because it is simple, fast, and stable.

7. Conclusion

To wrap things up, we have proposed several techniques to extend the linear algebraic PE to accommodate *Or*-rules and cycles in logic programs. First, the matrix representation of a standardized program Π is separated into $\mathbf{M}_{\Pi^\wedge}$ and $\mathbf{M}_{\Pi^\vee}$, then PE for *And*-rules and *Or*-rules can be handled similarly. Next, we introduced cycle-resolving techniques to ensure that linear algebraic PE works effectively even with local cycles in the $\Pi^\wedge$ or $\Pi^\vee$. Moreover, by seeing the PE as computing the power of the matrix representation of the program, we can further leverage eigenvalues and eigenvectors or program matrices to perform PE in vector spaces. To the best of our knowledge, this is the first attempt to incorporate matrix decomposition techniques into linear algebraic PE for LP. Although the decomposition method does not perform really well in practice, it opens up a new direction for future research where we focus on leveraging eigenvalues and eigenvectors of program matrices for reasoning with LP. It is also important to connect LP to *spectral graph theory* [33] in which researchers have also studied the connection between the eigenvalues of the adjacency matrix of a graph and its properties. Future work also includes investigating to extend linear algebraic PE to globally handle both *And*-rules and *Or*-rules in a logic program even with *global cycles*.

Table 2
Detailed comparison of execution time (in *ms*) of the linear algebraic PE methods on the *augmented* datasets with cycles. (green - best, red - worst)

	Artificial samples I (166 instances)		**Artificial samples II** (118 instances)		**FMEA samples** (213 instances)	
Parameters	mean / std	[min, max]	mean / std	[min, max]	mean / std	[min, max]
No. cycles And-rules	3.72 / 0.25	[1, 5]	3.68 / 0.30	[1, 5]	1.00 / 0.00	[1, 1]
No. cycles Or-rules	3.89 / 0.37	[1, 5]	3.75 / 0.42	[1, 5]	1.00 / 0.00	[1, 1]
Algorithms	`peval` (mean / std)	`resolve` (mean / std)	`peval` (mean / std)	`resolve` (mean / std)	`peval` (mean / std)	`resolve` (mean / std)
(I) Iteration + dense	821 780 / 62 340	573 / 27	4501 / 793	407 / 19	90 / 7	52 / 6
(II) Decomposition + dense	9 251 534 / 33 491	554 / 24	5 970 126 / 27 104	398 / 18	1 271 842 / 18 510	56 / 6
(I) Iteration + sparse	579 / 17	76 / 14	151 / 4	68 / 12	112 / 4	17 / 3

Acknowledgments

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

References

[1] C. Sakama, K. Inoue, T. Sato, Linear algebraic characterization of logic programs, in: International Conference on Knowledge Science, Engineering and Management, Springer, 2017, pp. 520–533.

[2] T. Sato, K. Inoue, C. Sakama, Abducing relations in continuous spaces., in: IJCAI, 2018, pp. 1956–1962.

[3] Y. Aspis, K. Broda, A. Russo, 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, 2020, pp. 59–68.

[4] A. Takemura, K. Inoue, Gradient-based supported model computation in vector spaces, in: Logic Programming and Nonmonotonic Reasoning, Springer International Publishing, LNAI, volume 13416, 2022, pp. 336–349.

[5] M. H. van Emden, R. A. Kowalski, The semantics of predicate logic as a programming language, J. ACM 23 (1976) 733–742.

[6] C. Sakama, K. Inoue, T. Sato, Logic programming in tensor spaces, Annals of Mathematics and Artificial Intelligence 89 (2021) 1133–1153.

[7] T. Sato, C. Sakama, K. Inoue, From 3-valued semantics to supported model computation for logic programs in vector spaces., in: ICAART (2), 2020, pp. 758–765.

[8] K. Inoue, Algebraic connection between logic programming and machine learning, in: Proceedings of the 17th International Symposium on Functional and Logic Programming (FLOPS 2024), volume 14659 of *Lecture Notes in Computer Science*, Springer, 2024.

[9] T. Sato, R. Kojima, Boolean network learning in vector spaces for genome-wide network analysis., in: KR, 2021, pp. 560–569.

[10] K. Gao, H. Wang, Y. Cao, K. Inoue, Learning from interpretation transition using differentiable logic programming semantics, Machine Learning 111 (2022) 123–145.

[11] K. Gao, K. Inoue, Y. Cao, H. Wang, A differentiable first-order rule learner for inductive logic programming, Artificial Intelligence 331 (2024) 104108.

[12] H. D. Nguyen, C. Sakama, T. Sato, K. Inoue, An efficient reasoning method on logic programming using partial evaluation in vector spaces, Journal of Logic and Computation 31 (2021) 1298–1316.

[13] T. Q. Nguyen, K. Inoue, C. Sakama, Linear algebraic abduction with partial evaluation, in: Practical Aspects of Declarative Languages: 25th International Symposium, PADL 2023, Boston, MA, USA, January 16–17, 2023, Proceedings, Springer, 2023, pp. 197–215.

[14] J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinska, T. C. Przymusinski, Dynamic updates of nonmonotonic knowledge bases, The journal of logic programming 45 (2000) 43–70.

[15] T. Q. Nguyen, K. Inoue, On converting logic programs into matrices., in: ICAART (2), 2023, pp. 405–415.

[16] T. Q. Nguyen, K. Inoue, C. Sakama, Enhancing linear algebraic computation of logic programs using sparse representation, New Generation Computing 40 (2022) 225–254.

[17] K. L. Clark, Negation as failure, Logic and data bases (1978) 293–322.

[18] M. Gelfond, V. Lifschitz, The stable model semantics for logic programming., in: ICLP/SLP, volume 88, 1988, pp. 1070–1080.

[19] F. Fages, Consistency of clark's completion and existence of stable models, Methods Log. Comput. Sci. 1 (1994) 51–60.

[20] P. Ferraris, J. Lee, V. Lifschitz, A generalization of the lin-zhao theorem, Ann. Math. Artif. Intell. 47 (2006) 79–101.

[21] E. Erdem, V. Lifschitz, Tight logic programs, Theory Pract. Log. Program. 3 (2003) 499–518.

[22] C. Sakama, H. D. Nguyen, T. Sato, K. Inoue, Partial evaluation of logic programs in vector spaces, in: International Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP 2018), Oxford UK, 2018.

[23] T. Q. Nguyen, K. Inoue, C. Sakama, Linear algebraic computation of propositional horn abduction, in: 2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI), IEEE, 2021, pp. 240–247.

[24] R. Tarjan, Depth-first search and linear graph algorithms, SIAM journal on computing 1 (1972) 146–160.

[25] Y. S. U. Vishkin, Y. Shiloach, An O(log n) parallel connectivity algorithm, J. algorithms 3 (1982) 57–67.

[26] Y. Zhang, A. Azad, A. Buluç, Parallel algorithms for finding connected components using linear algebra, Journal of Parallel and Distributed Computing 144 (2020) 14–27.

[27] G. Strang, Introduction to linear algebra 4th edition, SIAM, 2009.

[28] G. Sewell, Computational methods of linear algebra 2nd edition, World Scientific Publishing Company, 2014.

[29] R. Piziak, P. L. Odell, Matrix theory: from generalized inverses to Jordan form, Chapman and Hall/CRC, 2007.

[30] S. H. Weintraub, Jordan canonical form: theory and practice, Springer Nature, 2009.
[31] R. Koitz-Hristov, F. Wotawa, Faster horn diagnosis-a performance comparison of abductive reasoning algorithms, Applied Intelligence 50 (2020) 1558–1572.
[32] C. Moler, C. Van Loan, Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later, SIAM review 45 (2003) 3–49.
[33] R. A. Brualdi, Spectra of digraphs, Linear Algebra and its Applications 432 (2010) 2181–2213.