On Converting Logic Programs Into Matrices

Tuan Nguyen Quoc[a] and Katsumi Inoue[b]
National Institute of Informatics, Tokyo, Japan

Keywords: Logic Programming, Linear Algebra, Matrix Representation.

Abstract: Recently it has been demonstrated that deductive and abductive reasoning can be performed by exploiting the linear algebraic characterization of logic programs. Those experimental results reported so far on both forms of reasoning have proved that the linear algebraic approach can reach higher scalability than symbol manipulations. The main idea behind these proposed algorithms is based on linear algebra matrix multiplication. However, it has not been discussed in detail yet how to generate the matrix representation from a logic program in an efficient way. As conversion time and resulting matrix dimension are important factors that affect algebraic methods, it is worth investigating in standardization and matrix constructing steps. With the goal to strengthen the foundation of linear algebraic computation of logic programs, in this paper, we will clarify these steps and propose an efficient algorithm with empirical verification.

1 INTRODUCTION

Recently, Logic Programming (LP), which provides languages for declarative problem solving and symbolic reasoning (Lloyd, 2012), has started gaining more attention in terms of building explainable learning models (Shakerin and Gupta, 2020; D'Asaro et al., 2020; Garcez and Lamb, 2020; Hitzler, 2022). For decades, LP representation has been considered mainly in the form of symbolic logic (Kowalski, 1979), which is useful for declarative problem-solving and symbolic reasoning. Since then, researchers have developed various dedicated solvers and efficient tools for Answer Set Programming (ASP) - LP that are based on the stable model semantics (Schaub and Woltran, 2018). On the other hand, many researchers attempt to translate logical inference into numerical computation in the context of mixed integer programming (Bell et al., 1994; Gordon et al., 2009; Hooker, 1988; Liu et al., 2012). They exploit connections between logical inference and mathematical computation that open a new way for efficient implementations.

Lately, several studies have been done on embedding logic programs to numerical spaces and exploiting algebraic characteristics (Sakama et al., 2017; Sato, 2017; Aspis et al., 2020). There are multiple reasons for considering linear algebraic computation of LP. First, linear algebra is at the heart of a myriad of applications of scientific computation, and integrating linear algebraic computation and symbolic computation is considered a challenging topic in Artificial Intelligence (AI) (Saraswat, 2016). In particular, transforming symbolic representations into vector spaces and reasoning through matrix computation are considered one of the most promising approaches in neural-symbolic integration (Garcez and Lamb, 2020). Second, linear algebraic computation has the potential to develop scalable techniques dealing with huge relational knowledge bases that have been reported in several studies (Nickel et al., 2015; Rocktäschel et al., 2014; Yang et al., 2015). Since relational KBs consist of ground atoms, the next challenge is applying linear algebraic techniques to LP and deductive Knowledge Base (KB)s. Third, it would enable us to use efficient (parallel) algorithms of numerical linear algebra for computing LP, and further simplify the core method so that one can exploit great computing resources ranging from multi-threaded CPU to GPU. The promising efficiency has been reported in GraphBLAS where various graph algorithms are redefined in the language of linear algebra (Davis, 2019).

(Sakama et al., 2017) has coined the linear characteristics of the logic program by realizing deductive reasoning using matrix multiplication. The main idea is to transform a logic program into a matrix and then perform the T_P-operator (van Emden and Kowal-

[a] https://orcid.org/0000-0002-1754-9329
[b] https://orcid.org/0000-0002-2717-9122

Quoc, T. and Inoue, K.
On Converting Logic Programs Into Matrices.
DOI: 10.5220/0011802400003393
In *Proceedings of the 15th International Conference on Agents and Artificial Intelligence (ICAART 2023) - Volume 2*, pages 405-415
ISBN: 978-989-758-623-1; ISSN: 2184-433X

ski, 1976) in vector spaces by matrix multiplication (Sakama et al., 2017). Further extensions to disjunctive logic programs and normal logic programs are also discussed later in (Sakama et al., 2021). Based on these works, linear algebraic approaches have been developed for deduction (Nguyen et al., 2022) and abduction (Nguyen et al., 2021b) with great potential for scalability. Both the methods for deduction and abduction share the same way to convert a logic program into a matrix. This step is very important because it affects the execution time remarkably and also the dimension of the standardized matrix for further reasoning steps. However, the conversion step is not yet discussed in detail, therefore, in this paper, we conduct a detailed analysis of it and also propose an efficient algorithm to convert a logic program into a matrix with the goal to strengthen the foundation of linear algebraic computation of logic programs.

The rest of this paper is organized as follows: Section 2 describes the preliminaries and the definitions of program matrix for definite and normal logic programs; Section 3 analyzes the method in detail focusing on time complexity and also proposes a linear complexity algorithm to deal with converting a logic program into a matrix; Section 4 demonstrates the proposed algorithm on the graph coloring problem and verifies it has linear complexity; Section 5 discusses different matrix representation using by other linear algebraic methods in LP; Section 6 gives conclusion and discusses future work.

2 MATRIX REPRESENTATION OF LOGIC PROGRAM

The linear algebraic characteristics of LP were first analyzed by (Sakama et al., 2017; Sakama et al., 2021). We hereafter mention again the matrix representation of the logic program.

We consider a language $\mathscr{L}$ that contains a finite set of propositional variables. A logic program P is a finite set of clauses defined over a number of alphabets and the logical connectives $\neg$, $\wedge$, $\vee$, and $\leftarrow$. The Herbrand base B_P is the set of all propositional variables in a logic program P. A rule in P can be in one of the forms (1), (2), or (3).

$$h \leftarrow b_1 \wedge b_2 \wedge ... \wedge b_l \quad (l \geq 0) \tag{1}$$

$$h \leftarrow b_1 \vee b_2 \vee ... \vee b_l \quad (l \geq 0) \tag{2}$$

$$h \leftarrow b_1 \wedge b_2 \wedge ... \wedge b_l \wedge \neg b_{l+1} \wedge ... \wedge \neg b_{l+k} \quad (l+k \geq l \geq 0) \tag{3}$$

where h and b_i are propositional variables in $\mathscr{L}$.
For convenience, we refer to (1) as an *And*-rule, (2) as an *Or*-rule, and (2) as a *normal rule*.

For each rule r of the form (1) or (2) we define $head(r) = h$ and $body(r) = \{b_1, ..., b_l\}$. For each *normal rule* r (3), we define $head(r) = h$ and $body(r) = \{b_1, ..., b_l, \neg b_{l+1}, ..., \neg b_{l+k}\}$. Its $body(r)$ is partitioned into $body^+(r) = \{b_1, b_2, ..., b_l\}$ and $body^-(r) = \{\neg b_{l+1}, \neg b_{l+2}, ..., \neg b_{l+k}\}$ which refers to the positive and negative occurrences of atoms in $body(r)$, respectively. A rule r is a *fact* if $body(r) = \emptyset$. A rule r is a *constraint* if $head(r) = \emptyset$. A rule r is invalid if both $head(r)$ and $body(r)$ are $\emptyset$.

A *definite program* is a finite set of *And*-rules (1) or *Or*-rules (2). Note that an *Or*-rule is a shorthand of l *And*-rule $h \leftarrow b_1$, $h \leftarrow b_2$, ... $h \leftarrow b_l$. A *definite program* P is called an Singly-Defined (SD)-program if there are no two rules with the same head in it. Every *definite program* can be converted into an SD-program in the following manner. If there is more than one rule with the same head $\{h \leftarrow body(r_1), ..., h \leftarrow body(r_j)\}$, where $j > 1$ and $body(r_1), ..., body(r_j) \neq \emptyset$ (or h is not a $fact$). One can replace them with a set of rules $\{h \leftarrow b_1 \vee ... \vee b_j, b_1 \leftarrow body(r_1), ..., b_j \leftarrow body(r_j)\}$ including an *Or*-rule and j *And*-rules for j newly introduced atoms $\{b_1, ..., b_j\}$. We refer to the converted SD-program of a definite program P as a *standardized program* P^δ.

A *normal program* is a finite set of rules of the form (3). Note that a *normal program* P is a *definite program* if $body^-(r) = \emptyset$ for every rule $r \in P$. Normal logic programs can be transformed into definite programs as mentioned in (Alferes et al., 2000). Accordingly, one can obtain a *definite program* from a *normal program* P by replacing the negated literals in every rule of the form (3) and rewriting as follows:

$$h \leftarrow b_1 \wedge b_2 \wedge ... \wedge b_l \wedge \overline{b}_{l+1} \wedge ... \wedge \overline{b}_{l+k} \quad (l+k \geq l \geq 0) \tag{4}$$

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, denote as P^+. In terms of model computation, the stable models of P can be computed from the least model of P^+ and interpretation vectors as proposed in (Sakama et al., 2017; Sakama et al., 2021; Takemura and Inoue, 2022).

For convenience, we also adopt the definition of the same head logic program (SHLP) in (Gao et al., 2022) to the propositional language. In our paper, an SHLP is a normal logic program such that it has *at least two rules of the same head*. A normal program P may contain multiple SHLPs in it. This definition will be used in the next section for complexity analysis.

The main idea of representing a logic program using a matrix is to create a mapping from the set of all atoms to row/column indices of the matrix. It is essential for a matrix representation of a logic program

to contain exactly one row and one column for each atom in a program matrix. That is why we need to consider a program P in the *standardized format*. One can find a bijection from its set of atoms to indices of a matrix to represent every rule of P as a row in the matrix as follows:

Definition 1. Matrix Representation of Standardized Programs: (Sakama et al., 2021) Let P be a standardized program and $B_P = \{p_1, \ldots, p_n\}$. Then P is represented by a matrix $M_P \in \mathbb{R}^{n\times n}$ such that for each element a_{ij} $(1 \leq i, j \leq n)$ in M_P,

1. $a_{ij_k} = \dfrac{1}{l}$ $(1 \leq k \leq l; 1 \leq i, j_k \leq n)$ if $p_i \leftarrow p_{j_1} \wedge \cdots \wedge p_{j_l}$ is in P;
2. $a_{ij_k} = 1$ $(1 \leq k \leq c; 1 \leq i, j_k \leq n)$ if $p_i \leftarrow p_{j_1} \vee \cdots \vee p_{j_c}$ is in P;
3. $a_{ii} = 1$ if $p_i \leftarrow$ is in P;
4. $a_{ij} = 0$, otherwise.

M_P is called a *program matrix* of P. Let us consider a concrete example for a better understanding of Definition 1:

Example 1. Consider a definite program $P = \{p \leftarrow q \wedge r,\ p \leftarrow q \wedge s,\ q \leftarrow t,\ q \leftarrow s \wedge u,\ r \leftarrow,\ s \leftarrow\}$.
P is not an SD program because $p \leftarrow q \wedge r$, $p \leftarrow q \wedge s$ have the same head p and $q \leftarrow t$, $q \leftarrow s \wedge u$ have the same head q. Then P is converted into the standardized program P^{δ} by introducing new atoms $x_1 = q \wedge r$, $x_2 = q \wedge s$ and $x_3 = s \wedge u$ as follows: $P^{\delta} = \{p \leftarrow x_1 \vee x_2,\ q \leftarrow t \vee x_3,\ r \leftarrow,\ s \leftarrow,\ x_1 \leftarrow q \wedge r,\ x_2 \leftarrow q \wedge s,\ x_3 \leftarrow s \wedge u\}$. Then by applying Definition 1, one can obtain[1]:

	p	q	r	s	t	u	x_1	x_2	x_3
p							1.00	1.00	
q					1.00				1.00
r			1.00						
s				1.00					
t									
u									
x_1		0.50	0.50						
x_2		0.50		0.50					
x_3				0.50		0.50			

Note that it is not needed to introduce a new atom for the single-length body rule $q \leftarrow t$. Additionally, an introduced body can also be reused immediately without re-introducing it twice.

As mentioned, we have a way to transform normal logic programs into standardized programs, therefore representing a normal program P is made easy by first transforming it into a positive form P^+ and then converting P^+ into a standardized program $\mathbb{P}^+$.

[1]We omit all zero elements in matrices for better readability.

Definition 2. Matrix Representation of Normal Programs: (Sakama et al., 2021) Let P be a normal program, and $\mathbb{P}^+$ its transformed positive form as a standardized program. Suppose that the Herbrand base of $\mathbb{P}^+$ is given by $B_{\mathbb{P}^+} = \{p_1, \ldots, p_n, \overline{q}_{n+1}, \ldots, \overline{q}_{n+k}\}$ where p_i are positive literals and q_j $(n+1 \leq j \leq n+k)$ are negative literals appearing in $\mathbb{P}^+$. Then $\mathbb{P}^+$ is represented by a matrix $M_{\mathbb{P}^+} \in \mathbb{R}^{(n+k)\times(n+k)}$ such that for each element a_{ij} $(1 \leq i, j \leq n+k)$ in $\mathbb{P}^+$:

1. $a_{ii} = 1$ for $n+1 \leq i \leq n+k$;
2. $a_{ij} = 0$ for $n+1 \leq i \leq n+k$ and $1 \leq j \leq n+k$ such that $i \neq j$;
3. Otherwise, a_{ij} $(1 \leq i \leq n;\ 1 \leq j \leq n+k)$ is encoded as in Definition 1.

Let us consider an example.

Example 2. Consider a normal program $P = \{p \leftarrow q \wedge \neg r,\ q \leftarrow r \wedge \neg p,\ r \leftarrow p \wedge \neg q\}$.
First, transform P to P^+ such that $P^+ = \{p \leftarrow q \wedge \overline{r},\ q \leftarrow r \wedge \overline{p},\ r \leftarrow p \wedge \overline{q}\}$. P^+ satisfies the SD condition, therefore, its transformed positive form as a standardized program $\mathbb{P}^+ = P^+$. Then by applying Definition 2, one can obtain:

	p	q	r	$\overline{p}$	$\overline{q}$	$\overline{r}$
p		0.50				0.50
q			0.50	0.50		
r	0.50				0.50	
$\overline{p}$				1.00		
$\overline{q}$					1.00	
$\overline{r}$						1.00

3 LOGIC PROGRAM TO MATRIX CONVERSION

According to the previous section, we can have a quick summary of the logic program to matrix conversion method as described in Figure 1. In this section, we focus on the time complexity of the algebraic method. First, let us assume a logic program P with $|B_P| = n$. Let m be the number of rules in P, $\bar{l}$ be the average length of rules in P, k $(k \leq n)$ the number of negations appearing in P, g be the number of SHLPs in P, and $\overline{l_g}$ is the average number of rules in each SHLP of P.

Converting to Positive Form.
First, one has to scan the whole program to identify all negations in P. Thus, the complexity of this step is $\Theta(m\bar{l})$. Additionally, the resulting program P^+ of

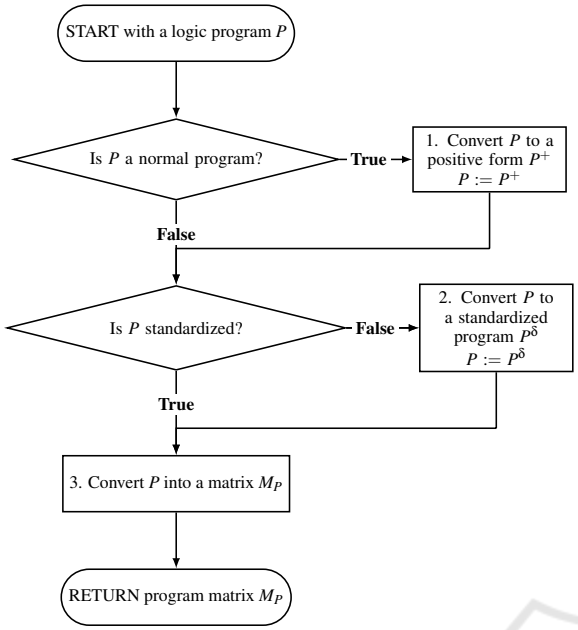

Figure 1: The flow of the logic program to matrix conversion method.

this step has more atoms than P has. In fact, we introduce new positive forms for all negations appearing in P, therefore $|B_{P^+}| = n+k$. This is also the number of atoms in the positive form program P^+. One can take the normal program in Example 2 to verify the calculation.

Converting to Standardized Program.
In the standardization step, we have to introduce a new *Or*-rule for each SHLP as mentioned in the previous section. In order to do that, first, we need a way to identify all SHLPs in P. Next, for each SHLP, we introduce a new atom for all the rule bodies with more than 1 atom in those. Then for that SHLP, we introduce a new *Or*-rule which includes all newly introduced atoms and all rule bodies of the length 1.

Because the number of rules of P is m, which does not change through the step converting to the positive form, the maximum number of new atoms we need to introduce is m. The worst case happens when all rule bodies are unique with length > 1 and they all belong to one of the SHLPs.

Let P' be the obtained program of P after the standardization step. Accordingly, n' is the number of atoms, m' is the number of rules, and $\overline{l}'$ is the average rule length. The following proposition holds:

Proposition 1. The number of atoms of P', n', is bounded by $n+m+k$.

Proof. According to the mentioned explanations above for each conversion step, we have to introduce exactly k new positive forms for all negations and at most m new atoms for all SHLPs. Hence, $n' \leq n+m+k$. □

The number of rules m' of P' is also increased as we have to introduce new rules for each SHLP. The following proposition holds:

Proposition 2. The number of rules of P', m', is bounded by $\left\lfloor \frac{3m}{2} \right\rfloor$.

Proof. As defined, an SHLP must have at least two rules, so the largest number of SHLPs in P is $\left\lfloor \frac{m}{2} \right\rfloor$. It happens when we have $\left\lfloor \frac{m}{2} \right\rfloor$ pairs of rules with the same head. In this case, we need to introduce $\left\lfloor \frac{m}{2} \right\rfloor$ new *Or*-rules for each pair. Additionally, if we assume all the rule bodies are unique and have a length larger than 1, we have to introduce m new atoms for those bodies to create new corresponding *And*-rules. These new *And*-rules will replace the old same head rules, so the number of *And*-rules remains the same. Therefore, m' is bounded by $m + \left\lfloor \frac{m}{2} \right\rfloor = \left\lfloor \frac{3m}{2} \right\rfloor$. □

Proposition 3. The average rule length of P' is $\overline{l}' = \frac{m\overline{l} + g\overline{l_g}}{m+g}$, where $g > 0$ is the number of SHLPs and $\overline{l_g}$ is the average number of rules in each SHLP.

Proof. Obviously, $m\overline{l}$ is the total length of all rules in P. Similarly, the total length of all rules in P' is $m'\overline{l'}$. Now consider the case there is an SHLP having two rules: $h \leftarrow body(r_1)$ and $h \leftarrow body(r_2)$. In this case, the number of SHLPs $g = 1$. The standardization step will replace these two rules by $\{h \leftarrow x_1 \vee x_2,\ x_1 \leftarrow body(r_1),\ x_2 \leftarrow body(r_2)\}$. As one can see, only the new disjunction $h \leftarrow x_1 \vee x_2$ affects the average rule length of P'. A similar manner can be extended to different SHLPs in P. Accordingly, we only need to determine how many of these rules will be introduced and the total length of them in P'.

Because we need to introduce a new disjunction rule for each SHLP, we have the number of rules $m' = m+g$. Additionally, the body length of a disjunction for each SHLP is also equal to the number of rules in that SHLP. So the increasing length of P' is $g\overline{l_g}$. Therefore, the total length of every rule in P' is $m'\overline{l'} = m\overline{l} + g\overline{l_g}$. Thus, $\overline{l'} = \frac{m\overline{l} + g\overline{l_g}}{m'} = \frac{m\overline{l} + g\overline{l_g}}{m+g}$. □

Corollary 1. The average rule length l' is bounded: $\bar{l}' \leq \frac{m\bar{l}+m}{m+1}$

Proof. According to the proof of Proposition 3, the increasing length of P' after introducing new disjunctions is $m'\bar{l}' = m\bar{l} + g\overline{l_g}$. Because P has maximum m rules, the total body length of newly introduced disjunctions cannot exceed m. Consequently, $g\overline{l_g} \leq m \Rightarrow \bar{l}' \leq \frac{m\bar{l}+m}{m+g} \leq \frac{m\bar{l}+m}{m+1}$. Equality happens when all rule bodies of P belong to an SHLP. □

Converting to Program Matrix.
Combining Proposition 1 and Definition 1, we have the following proposition:

Proposition 4. The lower bound of the dimension of the program matrix M_P is $n \times n$ while the upper bound is $(n+m+k) \times (n+m+k)$.

Proof. The lower bound is explained in the first step converting to the positive form, therefore, we only need to prove the upper bound. According to Proposition 1, the upper bound of the dimension of the program matrix is $(n+m+k) \times (n+m+k)$. Hence, the dimension of the program matrix is bounded by $(n+m+k) \times (n+m+k)$. □

On the other hand, the number of negations is bounded by the number of atoms in P or $k \leq n$, so we can also write that the dimension of the program matrix is bounded by $(2n+m) \times (2n+m)$.

The next step is converting P' into a matrix. In this step, we first init a zero matrix of the size $n' \times n'$. Then we go through all rules in P' to assign a value for each atom that appears in the rule body by a value following Definition 1. The following proposition holds:

Proposition 5. The number of needed assignments to convert P' into a matrix is bounded by $m\bar{l}+m$

Proof. Based on Definition 1, the number of needed assignments is $m'\bar{l}'$, where m' is the number of rules and $\bar{l}'$ is the average rule length of P'. According to Proposition 3, $\bar{l}' = \frac{m\bar{l}+g\overline{l_g}}{m+g}$ and $m' = m+g$, therefore, $m'\bar{l}' = m\bar{l} + g\overline{l_g}$. As already mentioned in the proof of Corollary 1, we have $g\overline{l_g} \leq m$. Accordingly, $m'\bar{l}' \leq m\bar{l}+m$. □

In summary, we need at most $m\bar{l}+m$ assignments in converting a logic program P into a matrix of the size which is bounded by $(n+m+k) \times (n+m+k)$.

Now, we will discuss how to implement the method efficiently. Through what we have analyzed above, the key is to avoid introducing new atoms if not needed and optimize the number of assignments as much as possible. An efficient way to implement this method is by exploiting a hashtable data structure for storing rules. For each *item* in the hashtable, *key* is the head of a rule and *value* is an array of rule bodies. A fact is an *item* with the *value* as an empty array. The hashtable ensures the complexity of look up and insertion of a *key-value* pair is $O(1)$ in average. The detailed method of converting a logic program into a matrix is described in Algorithm 1.

Algorithm 1: Converting a standardized program into a matrix.

Input: A logic program P

Output: Matrix representation $M_{P'}$ of P' where P' is the standardized program obtained from P

Init $\mathcal{H} := \{\emptyset\}$. ▷ Hashtable
for each rule r in P **do**
  for each atom b in body of r **do**
    if b is negation **then**
      Introduce positive form $\bar{b}$ of b.
      Insert $\bar{b}$ into $\mathcal{H}$. ▷ empty-body rule
  if head of r is not in $\mathcal{H}$ **then**
    Insert r into $\mathcal{H}$.
  else
    Append body of r to *value* of the existing item of r in $\mathcal{H}$.
for each r in $\mathcal{H}$ that has more than 1 body **do**
  for each body b of the rule r **do**
    if $|b| > 1$ **then**
      Introduce a new atom for b.
      Insert this new *And*-rule into $\mathcal{H}$.
  Clear the all rule bodies in the *value* of r and update r to *Or*-rule with newly introduced atoms.
$n' = \textbf{sizeof}(\mathcal{H})$.
Create a zero matrix $M_{P'}$ of the size $\mathbb{R}^{n' \times n'}$.
for each rule r in $\mathcal{H}$ **do**
  Assigning $M_{P'}$ according to Definition 1.
return assigned matrix $M_{P'}$.

Some explanations are in order:

- Step 2-10: Scan through all rules in the program and convert all negations to positive forms. Update the hashtable $\mathcal{H}$ by either inserting a new rule or updating the rule body of an existing rule. The complexity of this step is $O(m\bar{l})$ because we need to go through all the rules in the logic program P and for each rule, we also need to verify whether it contains negation or not.
- Step 11-16: Loop over only SHLPs and do the standardization step. In case all rules of P belong to an SHLP, we have to scan all these rules in P to introduce new atoms for each rule body as in the standardization step. Accordingly, the complexity of this step is $O(m)$.
- Step 19-20: Assigning values to the matrix according to Definition 1. According to Proposition 5, the number of needed assignments to convert P' into a matrix is bounded by $m\bar{l}+m$. This is also the number of non-zero elements in the output matrix. Therefore, the complexity of this step is $O(m\bar{l}+m)$.

As a result, the time complexity of Algorithm 1 is $O(m\bar{l}+m+m(\bar{l}+1)) = O(2m\bar{l}+2m) = O(m\bar{l})$, in other words, linear to the number of rules in P if we consider $\bar{l}$ as a constant. A similar result can be extended to the abductive matrix as it is defined based on the program matrix (Nguyen et al., 2021b).

The time complexity of Algorithm 1 is independent of the matrix representation format. It is worth noting that in the standardization step, introducing a new atom for a body of length 1 is not needed.

Proposition 6. Any algorithm converting a logic program into a matrix has the lower bound $\Omega(m\bar{l})$.

Proof. Any algorithm converting a logic program into a matrix has to go through all the rules of P at least once to read these rules into memory. Therefore, no algorithm can do better than $\Omega(m\bar{l})$. □

To sum up, we have proposed Algorithm 1, which has linear complexity, for the problem of converting a logic program into a matrix. In Section 4, we will provide an experiment to verify this conclusion.

4 BASELINE EXPERIMENT

For the purpose of this work, we provide a baseline experiment to verify the time complexity of the method of converting logic programs into matrices.

Experiment Setup.
In this experiment, we use the graph coloring problem - a problem in graph theory in that we search for possible a way of coloring all the vertices of a graph such that no pair of adjacent vertices are of the same color. The graph coloring problem can be formulated in first-order logic as follows:

Listing 1: Modeling of graph coloring in ASP language.

```
color(1..3).                                          1
1 { node_color(N,C): color(C) } 1 :-                  2
    node(N).
:- node_color(X,C), node_color(Y,C),                  3
    edge(X,Y), color(C).
```

This example is written following the syntax of an ASP language specified by `clingo` solver (`clasp`) (Gebser et al., 2009). In Listing 1, the first line defines there are 3 colors we can use to label nodes (3-Coloring). The second line is a choice rule that defines a node that can be labeled by only one color. And finally, the third rule is a constraint that declares there are no two adjacent nodes that can be labeled with the same color. Listing 2 illustrates an instance of the graph coloring problem with 6 nodes.

Listing 2: A problem instance of the graph coloring problem.

```
node(n1).                                             1
node(n2).                                             2
node(n3).                                             3
node(n4).                                             4
node(n5).                                             5
node(n6).                                             6
edge(n1,n2).                                          7
edge(n1,n3).                                          8
edge(n2,n3).                                          9
edge(n1,n4).                                          10
edge(n2,n4).                                          11
edge(n1,n5).                                          12
edge(n2,n5).                                          13
edge(n1,n6).                                          14
edge(n2,n6).                                          15
```

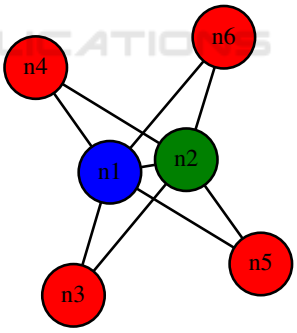

Figure 2.

We exploit `clingo` grounder (`gringo`) (Gebser et al., 2007) to deal with ASP format. First, `clingo` converts an ASP program into an ASP intermediate format - *aspif* (Gebser et al., 2016). In short, a program in an ASP intermediate format serves internally as the output of the grounder and the input of the solver. We then convert the obtained *aspif* format into our *internal format* which is equivalent to a propositional language as described in Section 2. The resulting program now can be treated as an input for Algorithm 1 to obtain the matrix representation. It is reported that the program matrix is highly sparse if the average rule length is small compared to the size

Table 1: Baseline results on the graph coloring problem (the unit of time is *millisecond*).

No.	Data instance (70 instances)	m	n	k	m'	n'	$\bar{l}$	η_z	t_{clingo}	t_{int}	t_{alg}	$t_{asg}^{(d)}$	$t_{asg}^{(s)}$
1	01_octahedral.txt	105	63	30	106	136	1.981	267	0.330	0.027	0.273	**0.068**	0.147
2	02_33fan.txt	101	62	30	102	132	1.971	257	0.312	0.025	0.281	**0.065**	0.145
3	03_42fan.txt	93	60	30	94	124	1.947	237	0.244	0.023	0.212	**0.035**	0.135
4	04_K133.txt	126	74	35	127	162	2.000	321	0.266	0.026	0.277	**0.067**	0.153
5	05_43cone.txt	130	75	35	131	166	2.008	331	0.260	0.027	0.288	**0.091**	0.156
6	06_43fan.txt	122	73	35	123	158	1.992	311	0.254	0.026	0.267	**0.045**	0.158
7	07_52fan.txt	110	70	35	111	146	1.964	281	0.249	0.025	0.247	**0.041**	0.146
8	08_K134.txt	151	86	40	152	192	2.020	385	0.287	0.033	0.338	**0.087**	0.174
9	g500a.txt	30,983	10,623	2,500	30,984	33,484	2.197	78,203	164.252	12.064	75.842	46.138	**21.731**
10	g500b.txt	31,215	10,681	2,500	31,216	33,716	2.198	78,783	161.577	12.641	79.654	46.807	**21.747**
11	g500c.txt	31,159	10,667	2,500	31,160	33,660	2.198	78,643	167.144	12.839	94.754	46.899	**22.856**
12	g500d.txt	30,631	10,535	2,500	30,632	33,132	2.197	77,323	162.624	12.772	98.012	48.870	**22.276**
13	g500e.txt	31,247	10,689	2,500	31,248	33,748	2.198	78,863	166.513	13.261	117.535	47.481	**22.746**
14	g1000a.txt	62,259	21,317	5,000	62,260	67,260	2.198	157,143	603.443	11.494	192.126	NA	**39.980**
15	g1000b.txt	61,523	21,133	5,000	61,524	66,524	2.197	155,303	519.572	10.429	184.788	96.685	**39.284**
16	g1000c.txt	63,059	21,517	5,000	63,060	68,060	2.198	159,143	510.697	10.153	209.855	NA	**40.232**
17	g1000d.txt	61,411	21,105	5,000	61,412	66,412	2.197	155,023	505.066	9.622	186.825	97.017	**39.611**
18	g1000e.txt	62,811	21,455	5,000	62,812	67,812	2.198	158,523	517.933	9.911	213.331	NA	**40.649**
19	g1500a.txt	132,823	41,833	7,500	132,824	140,324	2.213	334,303	1,464.692	24.899	487.264	NA	**85.393**
20	g1500b.txt	132,119	41,657	7,500	132,120	139,620	2.213	332,543	1,369.828	23.139	464.427	NA	**101.596**
21	g1500c.txt	134,519	42,257	7,500	134,520	142,020	2.214	338,543	1,365.100	23.323	491.663	NA	**87.459**
22	g1500d.txt	132,255	41,691	7,500	132,256	139,756	2.213	332,883	1,380.001	22.866	449.122	NA	**89.254**
23	g1500e.txt	133,887	42,099	7,500	133,888	141,388	2.214	336,963	1,369.460	23.022	479.502	NA	**87.569**
24	g2000a.txt	229,491	68,875	10,000	229,492	239,492	2.222	576,723	2,831.258	43.639	856.753	NA	**152.884**
25	g2000b.txt	229,763	68,943	10,000	229,764	239,764	2.222	577,403	2,565.067	41.061	887.015	NA	**152.745**
26	g2000c.txt	231,147	69,289	10,000	231,148	241,148	2.222	580,863	2,579.886	42.584	819.382	NA	**153.804**
27	g2000d.txt	231,707	69,429	10,000	231,708	241,708	2.222	582,263	2,572.219	40.758	908.208	NA	**156.532**
28	g2000e.txt	230,411	69,105	10,000	230,412	240,412	2.222	579,023	2,517.911	41.468	840.034	NA	**157.877**
29	g2500a.txt	353,639	102,787	12,500	353,640	366,140	2.227	887,843	4,477.236	71.675	1,433.340	NA	**242.757**
30	g2500b.txt	354,639	103,037	12,500	354,640	367,140	2.227	890,343	4,427.957	68.898	1,375.676	NA	**247.986**
31	g2500c.txt	353,999	102,877	12,500	354,000	366,500	2.227	888,743	4,478.155	65.696	1,362.169	NA	**244.847**
32	g2500d.txt	356,127	103,409	12,500	356,128	368,628	2.227	894,063	4,400.563	67.971	1,331.527	NA	**248.394**
33	g2500e.txt	355,415	103,231	12,500	355,416	367,916	2.227	892,283	4,455.153	70.469	1,410.733	NA	**245.272**
34	g3000a.txt	384,979	113,497	15,000	384,980	399,980	2.225	966,943	6,649.247	79.231	1,525.205	NA	**265.820**
35	g3000b.txt	386,771	113,945	15,000	386,772	401,772	2.225	971,423	6,125.425	75.262	1,478.556	NA	**266.498**
36	g3000c.txt	385,899	113,727	15,000	385,900	400,900	2.225	969,243	6,078.847	75.971	1,554.599	NA	**267.996**
37	g3000d.txt	386,843	113,963	15,000	386,844	401,844	2.225	971,603	6,030.022	73.548	1,450.245	NA	**266.121**
38	g3000e.txt	387,595	114,151	15,000	387,596	402,596	2.225	973,483	6,039.618	77.444	1,560.791	NA	**268.629**
39	g3500a.txt	422,263	125,693	17,500	422,264	439,764	2.223	1,060,903	8,833.187	87.178	1,530.809	NA	**292.929**
40	g3500b.txt	422,487	125,749	17,500	422,488	439,988	2.223	1,061,463	8,812.472	91.360	1,758.592	NA	**301.646**
41	g3500c.txt	422,111	125,655	17,500	422,112	439,612	2.223	1,060,523	9,003.760	88.826	1,719.789	NA	**297.962**
42	g3500d.txt	425,415	126,481	17,500	425,416	442,916	2.223	1,068,783	8,440.211	82.093	1,625.366	NA	**295.812**
43	g3500e.txt	424,951	126,365	17,500	424,952	442,452	2.223	1,067,623	8,506.824	81.822	1,668.312	NA	**296.487**
44	g4000a.txt	545,851	159,465	20,000	545,852	565,852	2.226	1,370,623	11,829.531	114.623	2,149.171	NA	**379.328**
45	g4000b.txt	547,731	159,935	20,000	547,732	567,732	2.226	1,375,323	10,600.458	118.387	2,185.071	NA	**381.556**
46	g4000c.txt	546,939	159,737	20,000	546,940	566,940	2.226	1,373,343	10,575.754	108.357	2,225.634	NA	**378.562**
47	g4000d.txt	550,507	160,629	20,000	550,508	570,508	2.226	1,382,263	10,460.095	107.292	2,128.184	NA	**381.873**
48	g4000e.txt	550,307	160,579	20,000	550,308	570,308	2.226	1,381,763	10,656.230	108.578	2,083.164	NA	**381.546**
49	g4500a.txt	580,591	171,025	22,500	580,592	603,092	2.225	1,458,223	14,352.234	124.617	2,290.720	NA	**409.873**
50	g4500b.txt	579,287	170,699	22,500	579,288	601,788	2.225	1,454,963	13,937.675	124.642	2,321.257	NA	**404.875**
51	g4500c.txt	580,343	170,963	22,500	580,344	602,844	2.225	1,457,603	13,930.808	126.757	2,271.633	NA	**403.584**
52	g4500d.txt	581,143	171,163	22,500	581,144	603,644	2.225	1,459,603	13,842.388	127.051	2,237.434	NA	**406.271**
53	g4500e.txt	584,407	171,979	22,500	584,408	606,908	2.225	1,467,763	14,125.477	129.132	2,290.216	NA	**409.394**
54	g5000a.txt	711,547	206,639	25,000	711,548	736,548	2.227	1,786,363	18,390.131	152.617	2,828.521	NA	**496.992**
55	g5000b.txt	712,211	206,805	25,000	712,212	737,212	2.227	1,788,023	18,301.363	158.893	2,785.979	NA	**510.082**
56	g5000c.txt	713,467	207,119	25,000	713,468	738,468	2.227	1,791,163	18,218.742	155.108	2,890.195	NA	**509.008**
57	g5000d.txt	712,851	206,965	25,000	712,852	737,852	2.227	1,789,623	18,273.827	143.145	2,828.261	NA	**503.874**
58	g5000e.txt	716,187	207,799	25,000	716,188	741,188	2.227	1,797,963	18,455.484	149.504	2,725.695	NA	**502.417**
59	structured-type1-100nodes.txt	2,991	1,325	500	2,992	3,492	2.138	7,623	9.821	1.090	6.781	3.928	**1.957**
60	structured-type1-900nodes.txt	28,751	12,365	4,500	28,752	33,252	2.148	73,223	606.712	13.673	72.614	43.158	**20.259**
61	structured-type1-1600nodes.txt	51,531	22,085	8,000	51,532	59,532	2.149	131,223	1,693.839	19.363	160.946	80.609	**37.406**
62	structured-type1-2500nodes.txt	80,911	34,605	12,500	80,912	93,412	2.149	206,023	3,951.911	26.385	289.428	NA	**59.276**
63	structured-type1-3600nodes.txt	116,891	49,925	18,000	116,892	134,892	2.150	297,623	8,187.798	33.079	467.196	NA	**83.547**
64	structured-type1-400nodes.txt	12,571	5,445	2,000	12,572	14,572	2.146	32,023	123.387	5.826	30.777	18.862	**8.261**
65	structured-type1-4900nodes.txt	159,471	68,045	24,500	159,472	183,972	2.150	406,023	15,266.063	41.605	607.520	NA	**113.451**
66	structured-type1-6400nodes.txt	208,651	88,965	32,000	208,652	240,652	2.150	531,223	28,546.341	53.976	847.902	NA	**147.779**
67	structured-type1-8100nodes.txt	264,431	112,685	40,500	264,432	304,932	2.150	673,223	49,308.250	66.401	1,063.075	NA	**190.640**
68	structured-type1-10000nodes.txt	326,811	139,205	50,000	326,812	376,812	2.151	832,023	88,130.737	76.450	1,303.904	NA	**231.742**
69	structured-type1-12100nodes.txt	395,791	168,525	60,500	395,792	456,292	2.151	1,007,623	130,363.785	91.751	1,559.279	NA	**279.478**
70	structured-type1-14400nodes.txt	471,371	200,645	72,000	471,372	543,372	2.151	1,200,023	193,854.156	113.170	1,885.119	NA	**338.243**

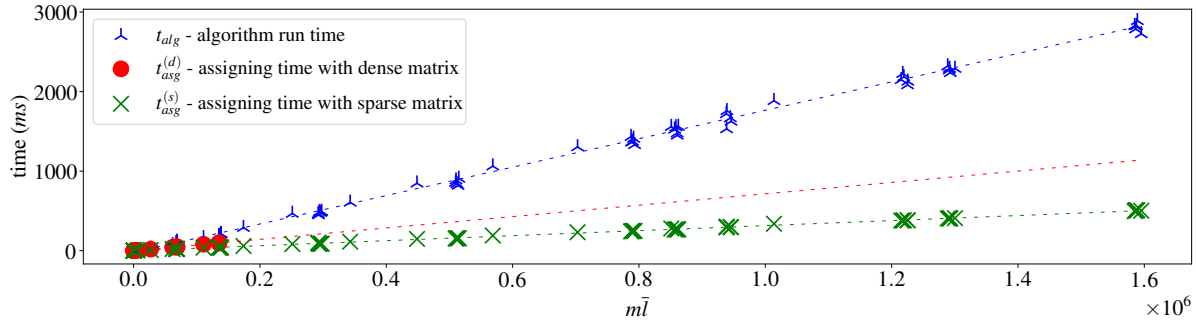

Figure 3: Run time trend lines based on $m\bar{l}$ for Algorithm 1 and assigning methods with dense and sparse formats.

of Herbrand base (Nguyen et al., 2022). Furthermore, sparseness is exploited successfully in both deduction (Nguyen et al., 2022) and abduction (Nguyen et al., 2021b) to reach higher scalability. Therefore, in this step, we also consider a comparison between *dense* and *sparse* matrix formats in terms of execution time assigning matrix values. Accordingly, we measure the execution time of different steps:

- t_{clingo} : execution time call to `clingo` grounder.
- t_{int} : time to convert *aspif* to our internal format.
- t_{alg} : time for Algorithm 1 excluding line 19, 20.
- $t^{(d)}_{asg}$: assigning time using dense matrix format.
- $t^{(s)}_{asg}$: assigning time using sparse matrix format [2].

All these steps are reported in Table 1 by taking the **mean** of 10 runs. Further trend lines are illustrated in Figure 3 for t_{alg}, $t^{(d)}_{asg}$ and $t^{(s)}_{asg}$. Table 1 also records m - number of rules, n - number of atoms, k - number of negations, m' - number of rules in the standardized program, n' - number of atoms in the standardized program or the matrix size, $\bar{l}$ - average rule length, η_z - number of non-zero elements.

The source code and dataset[3] are available at https://github.com/nqtuan0192/lpmatrixgrounder. This experiment is conducted on a machine having the following configurations: CPU: Intel Core i7-11800H @2.30GHz; RAM: 32GB; OS: Ubuntu 20.04 x86_64.

Experimental Results.

Overall, Algorithm 1 runs very fast in a matter of seconds while most of the time is spent on `clingo` grounder. The average length of the rule body is just above 2 for this dataset which has an influence on the sparsity of the program matrix. Additionally, η_z is also relatively small to the size of the program matrix $n' \times n'$. As a consequence, the sparsity of the matrix representations of all instances is 0.99 approximately.

`clingo` does many things under the hood that is out of our control, therefore it is too slow as can be seen in the column t_{clingo} of Table 1. In fact, neither dealing with first-order language nor benchmarking `clingo` grounder is the purpose of this paper. We provide the execution time t_{clingo} here as a reference for future improvement when we consider a method combining the grounding step and converting a logic program into a matrix. The time dealing with *aspif* is almost instant even though the matrix size is huge.

The main focus of our experiment is the execution time of Algorithm 1 and assigning time $t^{(d)}_{asg}$, $t^{(s)}_{asg}$. As reported Algorithm 1 can handle every instance in less than 3 seconds and is usually minor compared to t_{clingo}, especially in large-size instances. Assigning time with dense format $t^{(d)}_{asg}$ is faster than that with sparse format $t^{(s)}_{asg}$ in some very small size instances and also fast enough in general, however, the method is unable to allocate very large size matrix in which most of its elements are zero. The similar method using sparse matrix format, on the other hand, works well in every situation, especially more than 2 times faster in large-scale instances as can be witnessed in the column $t^{(s)}_{asg}$ of Table 1. Figure 3 further reports and verifies the execution time of the algorithm and the assigning step that they are linear complexity in general. The assigning step with the sparse format has the same complexity as that with the dense format in theory but the practical experiment shows the one using the sparse format performs better.

To wrap up, we have demonstrated and verified the theory complexity of the method of converting logic programs into matrices using the experiment with graph coloring problem.

[2]We use Coordinate (COO) format for the experiment.

[3]The original set of problem instances is obtained from http://www.info.deis.unical.it/npdatalog/experiments/experiments.htm, however, the link is no longer accessible.

5 RELATED WORK

There are several approaches to representing logic programs in the language of linear algebra. They share a similar idea but may differ in terms of how they assign values and how they organize the matrix to store these values. We can classify them mainly as square matrix and non-square matrix representations.

The matrix representation of logic programs was first coined by (Sakama et al., 2017) and then followed up by other researchers in (Sakama et al., 2018; Nguyen et al., 2018; Nguyen et al., 2021a). These works follow the same encoding scheme that we have summarized in this paper. Although they have not reported in detail how they perform the conversion method, their methods should have a very similar boundary to what we have analyzed. Later, the non-differentiable computation of 3-valued models of a program's completion in vector spaces was considered the first step towards computing supported models (Sato et al., 2020). His method prefers to use integers rather than real numbers to assign values to the matrix. However, the general idea is similar to the complexity of matrix conversion in his method is also bounded by linear complexity. (Aspis et al., 2020) have introduced a gradient-based search method for the computation of stable and supported models of normal logic programs in continuous vector spaces. The program matrix representation (Aspis et al., 2020) was influenced by (Sakama et al., 2017)'s idea so its complexity should be similar to our work.

With a different perspective, (Sato and Kojima, 2019) has proposed a differentiable framework for logic program inference as a step toward realizing flexible and scalable logical inference. They have further extended the idea to develop MatSat - a matrix-based differentiable SAT solver - and presented that this method outperforms all the Conflict-Driven Clause Learning (CDCL) type solvers using a random benchmark set from SAT 2018 competition (Sato and Kojima, 2021). (Sato and Kojima, 2019) represent the program matrix of a logic program by two halves, one half for the positive atoms and one half for the negations. By doing so, their method allows a non-square matrix in his method. The number of rows in the matrix is the same after the standardization step, however, (Sato and Kojima, 2019) has to include all the positive forms of atoms in the program. A similar encoding scheme can be seen in (Takemura and Inoue, 2022) that the authors have extended the idea of (Aspis et al., 2020) to present another gradient-based approach to compute supported models approximately. The matrix representation in (Takemura and Inoue, 2022)'s method is similar to the idea of (Sato and Kojima, 2019). We can follow the same method of analyzing complexity as we have presented to analyze the complexity of non-square matrix representation. Accordingly, the number of rules in non-square matrix representation may grow to $\left(n+\left\lfloor\frac{m}{2}\right\rfloor\right)$. However, in their encoding method, they need to keep all pairs of positive and negative forms of all atoms, so the number of columns is double the number of rows. Therefore, the maximum matrix size in the worst case is $\left(n+\left\lfloor\frac{m}{2}\right\rfloor\right)\times(2n+m)$. In terms of complexity, based on what we have analyzed in Section 3, we can estimate the complexity of their converting method is $O(2m\bar{l}+m)$. In Table 2, we illustrate a quick summary of the methods using square matrix and non-square matrix representations.

Table 2: A comparison of square matrix representation and non-square matrix representation.

	Square matrix	**Non-square matrix**
Papers	(Sakama et al., 2017), (Sakama et al., 2018), (Nguyen et al., 2018), (Nguyen et al., 2021a), (Nguyen et al., 2021b), (Nguyen et al., 2022)	(Sato and Kojima, 2019), (Sato and Kojima, 2021), (Aspis et al., 2020), (Takemura and Inoue, 2022)
Matrix size	$(n+m+k)\times(n+m+k)$ in the worst case	$\left(n+\left\lfloor\frac{m}{2}\right\rfloor\right)\times(2n+m)$ in the worst case
Complexity	$O(2m\bar{l}+2m)$	$O(2m\bar{l}+m)$

At a glance, representation using a non-square matrix seems better in terms of space and time. On the other hand, a square matrix is compatible to apply the power of matrix to itself that realizes partial evaluation with speed-up benefit (Nguyen et al., 2021a). The performance gain usually outweighs the cost of constructing matrices as demonstrated in their paper.

6 CONCLUSION

In this paper, we have reported a detailed analysis of converting logic programs into matrices with an efficient algorithm. It is proved that the complexity of the method is $O(m\bar{l})$, linear to the number of rules and is close to the lower bound $\Omega(m\bar{l})$. The experiment supports us to verify the theory in practice using a graph coloring problem. This work aims to contribute to the theory of linear algebraic characteristics of logic programs (Sakama et al., 2017; Sakama et al., 2021) in addition to build up the baseline for different matrix representations employed in embedding logic programs into vector spaces.

In the future, working on handling first-order logic is an important topic to extend the potential of ma-

trix representation to a wider range of applications. Another interesting direction could involve optimizing the number of newly introduced rules in the standardization step. Moreover, as we keep improving the method for the standardization step, we also need to do further analysis for a better approximation of the algorithm complexity.

ACKNOWLEDGEMENTS

This work has been supported by JSPS, KAKENHI Grant Numbers JP18H03288 and JP21H04905, and by JST, CREST Grant Number JPMJCR22D3, Japan.

REFERENCES

Alferes, J. J., Leite, J. A., Pereira, L. M., Przymusinska, H., and Przymusinski, T. C. (2000). Dynamic updates of non-monotonic knowledge bases. *The journal of logic programming*, 45(1-3):43–70.

Aspis, Y., Broda, K., Russo, A., and Lobo, J. (2020). 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*, pages 59–68.

Bell, C., Nerode, A., Ng, R. T., and Subrahmanian, V. (1994). Mixed integer programming methods for computing nonmonotonic deductive databases. *Journal of the ACM (JACM)*, 41(6):1178–1215.

D'Asaro, F. A., Spezialetti, M., Raggioli, L., and Rossi, S. (2020). Towards an inductive logic programming approach for explaining black-box preference learning systems. In *Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning*, pages 855–859.

Davis, T. A. (2019). Algorithm 1000: Suitesparse: GraphBLAS: Graph algorithms in the language of sparse linear algebra. *ACM Transactions on Mathematical Software (TOMS)*, 45(4):1–25.

Gao, K., Inoue, K., Cao, Y., and Wang, H. (2022). Learning first-order rules with differentiable logic program semantics. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, pages 3008–3014.

Garcez, A. d. and Lamb, L. C. (2020). Neurosymbolic AI: the 3rd wave. *arXiv preprint arXiv:2012.05876*.

Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., and Wanko, P. (2016). Theory solving made easy with clingo 5 (extended version).

Gebser, M., Kaufmann, B., and Schaub, T. (2009). The conflict-driven answer set solver clasp: Progress report. In *LPNMR*, volume 5753 of *Lecture Notes in Computer Science*, pages 509–514. Springer.

Gebser, M., Schaub, T., and Thiele, S. (2007). Gringo : A new grounder for answer set programming. In *LPNMR*, volume 4483 of *Lecture Notes in Computer Science*, pages 266–271. Springer.

Gordon, G. J., Hong, S. A., and Dudík, M. (2009). First-order mixed integer linear programming. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*, UAI '09, page 213–222, Arlington, Virginia, USA. AUAI Press.

Hitzler, P. (2022). *Neuro-Symbolic Artificial Intelligence: The State of the Art*. IOS Press.

Hooker, J. N. (1988). A quantitative approach to logical inference. *Decision Support Systems*, 4(1):45–69.

Kowalski, R. (1979). *Logic for problem solving*. North Holland, Elsevier.

Liu, G., Janhunen, T., and Niemela, I. (2012). Answer set programming via mixed integer programming. In *Thirteenth International Conference on the Principles of Knowledge Representation and Reasoning*.

Lloyd, J. W. (2012). *Foundations of logic programming*. Springer Science & Business Media.

Nguyen, H. D., Sakama, C., Sato, T., and Inoue, K. (2018). Computing logic programming semantics in linear algebra. In *International Conference on Multidisciplinary Trends in Artificial Intelligence*, pages 32–48. Springer.

Nguyen, H. D., Sakama, C., Sato, T., and Inoue, K. (2021a). An efficient reasoning method on logic programming using partial evaluation in vector spaces. *Journal of Logic and Computation*, 31(5):1298–1316.

Nguyen, T. Q., Inoue, K., and Sakama, C. (2021b). Linear algebraic computation of propositional horn abduction. In *2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 240–247. IEEE.

Nguyen, T. Q., Inoue, K., and Sakama, C. (2022). Enhancing linear algebraic computation of logic programs using sparse representation. *New Generation Computing*, 40(1):225–254. A preliminary version is in: Online Proceedings of ICLP 2020, EPTCS vol. 325, pp. 192-205 (2020).

Nickel, M., Murphy, K., Tresp, V., and Gabrilovich, E. (2015). A review of relational machine learning for knowledge graphs. *Proceedings of the IEEE*, 104(1):11–33.

Rocktäschel, T., Bošnjak, M., Singh, S., and Riedel, S. (2014). Low-dimensional embeddings of logic. In *Proceedings of the ACL 2014 Workshop on Semantic Parsing*, pages 45–49.

Sakama, C., Inoue, K., and Sato, T. (2017). Linear algebraic characterization of logic programs. In *International Conference on Knowledge Science, Engineering and Management*, pages 520–533. Springer.

Sakama, C., Inoue, K., and Sato, T. (2021). Logic programming in tensor spaces. *Annals of Mathematics and Artificial Intelligence*, 89(12):1133–1153.

Sakama, C., Nguyen, H. D., Sato, T., and Inoue, K. (2018). Partial evaluation of logic programs in vector spaces. *Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP'18@FLoC)*.

Saraswat, V. (2016). Reasoning 2.0 or machine learning and logic–the beginnings of a new computer science. *Data Science Day, Kista Sweden*.

Sato, T. (2017). A linear algebraic approach to datalog evaluation. *Theory and Practice of Logic Programming*, 17(3):244–265.

Sato, T. and Kojima, R. (2019). Logical inference as cost minimization in vector spaces. In *International Joint Conference on Artificial Intelligence*, pages 239–255. Springer.

Sato, T. and Kojima, R. (2021). Matsat: a matrix-based differentiable sat solver. *Pragmatics of SAT - a workshop of the 24nd International Conference on Theory and Applications of Satisfiability Testing*.

Sato, T., Sakama, C., and Inoue, K. (2020). From 3-valued semantics to supported model computation for logic programs in vector spaces. In *ICAART (2)*, pages 758–765.

Schaub, T. and Woltran, S. (2018). Special issue on answer set programming. *KI-Künstliche Intelligenz*, 32(2):101–103.

Shakerin, F. and Gupta, G. (2020). White-box induction from svm models: Explainable ai with logic programming. *Theory and Practice of Logic Programming*, 20(5):656–670.

Takemura, A. and Inoue, K. (2022). Gradient-based supported model computation in vector spaces. In *Logic Programming and Nonmonotonic Reasoning*, pages 336–349, Cham. Springer International Publishing.

van Emden, M. H. and Kowalski, R. A. (1976). The semantics of predicate logic as a programming language. *J. ACM*, 23(4):733–742.

Yang, B., Yih, S. W.-t., He, X., Gao, J., and Deng, L. (2015). Embedding entities and relations for learning and inference in knowledge bases. In *Proceedings of ICLR 2015*.