New Generation Computing (2022) 40:225–254
https://doi.org/10.1007/s00354-021-00142-2

Enhancing Linear Algebraic Computation of Logic Programs Using Sparse Representation

Nguyen Tuan Quoc[1,2] · **Katsumi Inoue**[1,2] · **Chiaki Sakama**[3]

Received: 9 February 2021 / Accepted: 29 October 2021 / Published online: 2 December 2021

Abstract
Algebraic characterization of logic programs has received increasing attention in recent years. Researchers attempt to exploit connections between linear algebraic computation and symbolic computation to perform logical inference in large-scale knowledge bases. In this paper, we analyze the complexity of the linear algebraic methods for logic programs and propose further improvement by using sparse matrices to embed logic programs in vector spaces. We show its great power of computation in reaching the fixed point of the immediate consequence operator. In particular, performance for computing the least models of definite programs is dramatically improved using the sparse matrix representation. We also apply the method to the computation of stable models of normal programs, in which the guesses are associated with initial matrices, and verify its effect when there are small numbers of negation. These results show good enhancement in terms of performance for computing consequences of programs and depict the potential power of tensorized logic programs.

Keywords Logic program · Fixed-point computation · Linear algebra · Sparse representation

✉ Nguyen Tuan Quoc
tuannq@nii.ac.jp

Katsumi Inoue
inoue@nii.ac.jp

Chiaki Sakama
sakama@wakayama-u.ac.jp

[1] National Institute of Informatics, 2 Chome-1-2 Hitotsubashi, Chiyoda City, Tokyo, Japan

[2] Department of Informatics, The Graduate University for Advanced Studies, Sokendai, Tokyo, Japan

[3] Wakayama University, 930 Sakaedani, Wakayama, Japan

Introduction

For decades, logic programming (LP) representation has been considered mainly in the form of symbolic logic [14], which is useful for declarative problem solving and symbolic reasoning. Logic programming starts gaining more attention recently to build explainable learning models [8, 27], whereas it still has some limitations in terms of computation. In other words, symbolic computation is not an efficient way when we need to combine it with other numerical learning models such as artificial neural network (ANN). Recently, several studies have been done on embedding logic programs to numerical spaces so that we can exploit great computing resources ranging from multi-threaded CPU to GPU. The linear algebraic approach is a robust way to manipulate logic programs in numerical spaces. Because linear algebra is at the heart of many applications of scientific computation, this approach is promising to develop scalable techniques to process huge relational knowledge base (KB) [20, 29]. In addition, it enables the ability to use efficient parallel algorithms of numerical linear algebra for computing LP.

In [7], Cohen described a probabilistic deductive database system in which reasoning is performed by a differentiable process. With this achievement, they can enable novel gradient-based learning algorithms. In [23], Sato presented the use of first-order logic in vector spaces for Tarskian semantics, which demonstrates how tensorization realizes efficient computation of Datalog. In [24], Sato proposed a linear algebraic approach to datalog evaluation. In this work, the least Herbrand model of DB is computed via adjacency matrices. He also provided theoretical proofs for translating a program into a system of linear matrix equations. This approach achieves $O(N^3)$ time complexity where N is the number of variables in a clause. Continuing to this direction, Sato, Inoue, and Sakama developed linear algebraic abduction to abductive inference in Datalog [25]. They did empirical experiments on linear and recursive cases and indicated that the approach can successfully abduce base relations.

In [13], Hitzler et al. theoretically proved that first-order normal logic programs can be approximated by feedforward connectionist networks based on the well-known theorem of Funahashi [9] that every feedforward neural network with at least 3 layers can uniformly approximate any continuous function. Hitzler et al. realized the use of neural networks to compute the immediate consequence operator T_P and further extended it to first-order logic. However, the main open question is how to find the appropriate structure of the network (how many layers, how many neurons per layer) for a given logic program. In this regard, Serafini and Garcez show how real logic can be implemented in deep ANN [26] then propose logic tensor networks (LTN). The framework is built upon a learning task with both knowledge and data being mapped onto real-valued vectors that the authors follow an inference-as-learning approach.

Using a linear algebraic method, Sakama, Inoue, and Sato define relations between LP and multi-dimensional array (tensor) then propose algorithms for computation of LP models [21, 22]. The representation is done by defining a series of conversions from logical rules to vectors and then the computation is

done by applying matrix multiplication. Later, elimination techniques are applied to reduce the matrix size [16] and gain impressive performance. In [3], a similar idea using 3D tensor was employed to compute solutions of abductive Horn propositional tasks. In addition, Aspis built upon previous works on matrix characterization of Horn propositional logic programs to explore how inference from logic programs can be done by linear algebraic algorithms [2]. He also proposed a new algorithm for the non-monotonic deduction, based on linear algebraic reducts and differentiable deduction. These works show that the linear algebraic methods are promising for logic inference on large scales. However, such methods have not yet been proved to be really efficient, since they have not yet been done adequate experiments, to the best of our knowledge.

In this paper, we continue Sakama et al.'s idea of representing logic programs by tensors [16, 21, 22]. Although the method is well-defined, there are some problems, which limit the performance of the approach and have not been solved. First, the obtained matrix after conversion is sparse but sparsity analysis has never been considered yet. Second, the experiments were limited to small-size logic programs that are not sufficient to prove the robustness of matrix representation. In this research, we further raise the bar of computing performance using sparse representation for logic programs in order to reach the fixed point of the immediate consequence operator (T_P-operator). We are able to do experiments on large sizes of logic programs to demonstrate the performance for computing least models of definite programs. Note that computation of the fixed point of the T_P-operator frequently appears in logic programming, not only in obtaining the least model of a definite program but also in any model construction, e.g., computing the minimal models of the reduct of a normal or disjunctive logic program with negation. In this regard, we also conduct experiments on the computation of stable models of normal programs with a small number of negations.

Accordingly, the rest of this paper is organized as follows:[1] Sect. 2 reviews and summaries some definitions and computation algorithms for definite and normal programs, Sect. 3 discusses sparsity problem in tensorized logic programs and proposes a method to represent LP, Sect. 4 investigates space and time complexity of the methods, Sect. 5 demonstrates experimental results with definite and normal programs, and Sect. 6 gives final conclusions and future works.

Preliminaries

Definite Programs

We consider a language $\mathscr{L}$ that contains a finite set of propositional variables. A *definite (logic) program* is a finite set of *rules* of the form:

[1] A preliminary version of this paper was presented as a Technical Communication paper at The 36th International Conference on Logic Programming (ICLP 2020) [18]. This paper has much extended the contents of [18] by considering several sparse methods for logic programs and comparing them both analytically with complexity results presented and experimentally with more datasets.

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

where h and b_i are propositional variables (atoms) in $\mathscr{L}$.

Given a logic program P, the set of all propositional variables appearing in P is called the *Herbrand base* of P (written B_P). For each rule r of the form (1), define $head(r) = h$ and $body(r) = \{b_1, \ldots, b_m\}$. A rule is called a fact if $body(r) = \emptyset$. A definite program P is called an singly-defined (SD) program if there are no two rules that have the same head in it, that is $head(r_1) \neq head(r_2)$ for any two rules r_1 and r_2 ($r_1 \neq r_2$) in P.

When a definite program P contains more than one rule (of the form (1)) having the same head:

$$h \leftarrow \mathscr{B}_1$$
$$\cdots$$
$$h \leftarrow \mathscr{B}_n,$$

where $\mathscr{B}_i$, $(1 \leq i \leq 1)$ is a conjunction (possibly empty) of atoms, we can replace them with a set of new rules:

$$h \leftarrow b_1 \vee \cdots \vee b_n \quad (n \geq 0), \tag{2}$$

$$b_i \leftarrow \mathscr{B}_i \ (i = 1, \ldots, n), \tag{3}$$

where b_i $(i = 1, \ldots, n)$ are newly introduced atoms ($b_i \notin B_P$) such that $b_i \neq b_j$ if $i \neq j$. Then the set of rules of (3) is an SD program. Each rule of form (2) is called an OR-rule. Every definite program P is transformed to a program $P' = Q \cup D$ such that Q is an SD program and D is a set of OR-rules. The resulting program P' is called a *standardized program*. A definite program P coincides with its standardized form P' iff P is an SD program. By introducing the OR-rule (2) which is a shorthand of n rules: $h \leftarrow b_1, \ldots, h \leftarrow b_n$ including new atoms, the Herbrand base of P' (written $B_{P'}$) is usually larger that B_P. In this paper, a program means a standardized program unless stated otherwise.

A set $I \subseteq B_P$ is an *interpretation* of P. An interpretation I is a *model* of a standardized program P if $\{b_1, \ldots, b_m\} \subseteq I$ implies $h \in I$ for every rule (1) in P, and $\{b_1, \ldots, b_m\} \cap I \neq \emptyset$ implies $h \in I$ for every rule (2) in P. A model I is the *least model* of P if $I \subseteq J$ for any model J of P. A mapping $T_P : 2^{B_P} \rightarrow 2^{B_P}$ (called a T_P-*operator*) is defined as: $T_P(I) = \{\, h \mid h \leftarrow b_1 \wedge \cdots \wedge b_m \in P \text{ and } \{b_1, \ldots, b_m\} \subseteq I \,\} \cup \{\, h \mid h \leftarrow b_1 \vee \cdots \vee b_n \in P \text{ and } \{b_1, \ldots, b_n\} \cap I \neq \emptyset \,\}$.

The *powers* of T_P are defined as: $T_P^{k+1}(I) = T_P(T_P^k(I))$ $(k \geq 0)$ and $T_P^0(I) = I$. Given $I \subseteq B_P$, there is a fixed-point $T_P^{n+1}(I) = T_P^n(I)$ $(n \geq 0)$. For a definite program P, the fixed-point $T_P^n(\emptyset)$ coincides with the least model of P [28].

Definition 1 (*Matrix representation of standardized programs* [21])

Let P be a standardized program and $B_P = \{p_1, \cdots, p_n\}$. 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. $a_{ij_k} = \frac{1}{m}$ $(1 \leq k \leq m; 1 \leq i, j_k \leq n)$ if $p_i \leftarrow p_{j_1} \wedge \cdots \wedge p_{j_m}$ is in P;
2. $a_{ij_k} = 1$ $(1 \leq k \leq l; 1 \leq i, j_k \leq n)$ if $p_i \leftarrow p_{j_1} \vee \cdots \vee p_{j_l}$ 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*. We write $\mathsf{row}_i(M_P) = p_i$ and $\mathsf{col}_j(M_P) = p_j$ $(1 \leq i, j \leq n)$.

To better understand Definition 1, let us consider a concrete example.

Example 1 Consider the definite program $P = \{p \leftarrow q \wedge r,\ p \leftarrow s \wedge t,\ r \leftarrow s,\ q \leftarrow t,\ s \leftarrow,\ t \leftarrow\}$.

P is not an SD program because there are two rules $p \leftarrow q \wedge r$ and $p \leftarrow s \wedge t$ having the same head, then P is transformed to the standardized program P' by introducing new atoms u and v as follows: $P' = \{u \leftarrow q \wedge r,\ v \leftarrow s \wedge t,\ p \leftarrow u \vee v,\ r \leftarrow s,\ q \leftarrow t,\ s \leftarrow,\ t \leftarrow\}$. Then by applying Definition 1, we obtain:

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

Sakama et al. further define representation of interpretation using *interpretation vectors* (Definition 2). This vector is used to store the truth value of all propositions in P. The starting point of *interpretation vector* is defined as the *initial vector* (Definition 3).

Definition 2 (*Interpretation vector* [21])

Let P be a program and $B_P = \{p_1, \ldots, p_n\}$. Then an interpretation $I \subseteq B_P$ is represented by a vector $v = (a_1, \ldots, a_n)^{\mathsf{T}}$, where each element a_i $(1 \leq i \leq n)$ represents the truth value of the proposition p_i such that $a_i = 1$ if $p_i \in I$; otherwise, $a_i = 0$. We write $\mathsf{row}_i(v) = p_i$.

Definition 3 (*Initial vector*) Let P be a program and $B_P = \{p_1, \ldots, p_n\}$. Then, the *initial vector* of P is an interpretation vector $v_0 = (a_1, \ldots, a_n)^{\mathsf{T}}$ such that $a_i = 1$ $(1 \leq i \leq n)$ if $\mathsf{row}_i(v_0) = p_i$ and a fact $p_i \leftarrow$ is in P; otherwise, $a_i = 0$.

To compute the least model in vector space, Sakama et al. proposed an algorithm that is equivalent to the result of computing least models by the T_P-operator. This algorithm is presented in Algorithm 1.

Definition 4 (θ-*thresholding*) Given a value x, define $\theta(x) = x'$, where $x' = 1$ if $x \geq 1$; otherwise, $x' = 0$.

Similarly, the *θ-thresholding* is extended in an element-wise way to vectors and matrices.

Algorithm 1 Matrix computation of least model

Input: a *definite program* P and its Herbrand base $B_P = \{p_1, p_2, ..., p_n\}$
Output: a vector v representing the least model

1: transform P to a *standardized program* $P^s = Q \cup D$ with $B_{P^s} = \{p_1, p_2, ..., p_n, p_{n+1}, ..., p_m\}$ where Q is an SD program and D is a set of OR-rules.
2: create matrix $M_{P^s} \in \mathbb{R}^{m \times m}$ representing P^s
3: create initial vector $v_0 = (v_1, v_2, ..., v_m)^T$ of P^s
4: $v = v_0$
5: $u = \theta(M_{P^s} v)$
6: **while** $u \neq v$ **do**
7: $\quad v = u$
8: $\quad u = \theta(M_{P^s} v)$
9: **end while**
10: **return** v

Normal Programs

Normal programs can be transformed to definite programs as introduced in [1]. Therefore, we transform normal programs to definite programs before encoding them in matrices.

Definition 5 (*Normal program*) A *normal program* is a finite set of normal rules:

$$h \leftarrow b_1 \wedge b_2 \wedge \cdots \wedge b_l \wedge \neg b_{l+1} \wedge \cdots \wedge \neg b_m \ (m \geq l \geq 0), \tag{4}$$

where h and $b_i (1 \leq i \leq m)$ are propositional variables (atoms) in $\mathscr{L}$.

P is transformed to a definite program by rewriting the above rule into the following form:

$$h \leftarrow b_1 \wedge b_2 \wedge \cdots \wedge b_l \wedge \overline{b}_{l+1} \wedge \cdots \wedge \overline{b}_m \ (m \geq l \geq 0), \tag{5}$$

where $\overline{b}_i$ is a new proposition associated with b_i.

In this part, we denote P as a normal program with an interpretation $I \subseteq B_P$. The *positive form* P^+ of P is obtained by applying the above transformation. Since a definite program P^+ is transformed to its standardized program, then we can apply Algorithm 1 to compute the least model. [1] proved that if P is a normal program, I is a stable model of P iff I^+ is the least model of $P^+ \cup \bar{I}$, where $\bar{I} = \{\bar{p} \mid p \in B_P \backslash I\}$, then $I^+ = I \cup \bar{I}$. We should note that I^+ is an interpretation of P^+ which is a definite program. We can obtain I^+ by applying Algorithm 1 to the transformed program P^+.

Definition 6 (*Matrix representation of normal programs* [16])

Let P be a normal program with $B_P = \{p_1, \ldots, p_n\}$ and its *positive form* P^+ with $B_{P^+} = \{p_1, \ldots, p_n, \overline{q}_{n+1}, \ldots, \overline{q}_m\}$.

Then, P^+ is represented by a matrix $M_P \in \mathbb{R}^{m \times m}$ such that for each element a_{ij} $(1 \leq i, j \leq m)$:

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

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

Example 2 Consider a program $P = \{p \leftarrow q \wedge s,\ q \leftarrow p \wedge t,\ s \leftarrow \neg t,\ t \leftarrow,\ u \leftarrow v\}$.

First, transform P to P^+ such that $P^+ = \{p \leftarrow q \wedge s,\ q \leftarrow p \wedge t,\ s \leftarrow \bar{t},\ t \leftarrow,\ u \leftarrow v\}$. Then applying Definition 6, we obtain:

$$\begin{array}{c} \\ p \\ q \\ s \\ t \\ u \\ v \\ \bar{t} \end{array} \begin{array}{c} \begin{array}{ccccccc} p & q & s & t & u & v & \bar{t} \end{array} \\ \begin{pmatrix} 0 & 1/2 & 1/2 & 0 & 0 & 0 & 0 \\ 1/2 & 0 & 0 & 1/2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

Instead of the initial vector in the case of definite programs, the notion of an initial matrix is introduced to encode multiple interpretations containing positive and negative facts in a program.

Definition 7 (*Initial matrix* [16])

Let P be a normal program and $B_P = \{p_1, \ldots, p_n\}$ and its positive form P^+ with $B_{P^+} = \{p_1, \ldots, p_n, \overline{q}_{n+1}, \ldots, \overline{q}_m\}$. The initial matrix $M_0 \in \mathbb{R}^{m \times h} (1 \leq h \leq 2^{m-n})$ is defined as follows:

1. each row of M_0 corresponds to each element of B_P in a way that $row_i(M_0) = p_i$ for $1 \leq i \leq n$ and $row_i(M_0) = \overline{q}_i$ for $n+1 \leq i \leq m$;
2. $a_{ij} = 1 (1 \leq i \leq n, 1 \leq j \leq h)$ iff a fact $q_i \leftarrow$ is in P; otherwise $a_{ij} = 0$;
3. $a_{ij} = 0$ $(n+1 \leq i \leq m, 1 \leq j \leq h)$ iff a fact $p_k \leftarrow$ (with $1 \leq k \leq n$) is in P and $q_i = \overline{p_k}$; otherwise, a_{ij} takes the value 0 or 1 in a way that every combination in 2^{m-n} (except the deterministic case of $a_{ij} = 0$) is enumerated.

Each column of M_0 is a potential stable model in the first stage. We update M_0 by applying matrix multiplication with the matrix representation obtained by Definition 6 as $M_{k+1} = \theta(M_P M_k)$. The resulting matrices are called interpretation matrices that each of which includes multiple interpretations of the corresponding program. Then, the algorithm for computing the stable models is presented in Algorithm 2.

Algorithm 2 Matrix computation of stable models

Input: a *normal program* P and its Herbrand base $B_P = \{p_1, p_2, ..., p_n\}$
Output: a set of vectors V representing the stable models or P

1: transform P to a *standardized program* P^+ with $B_{P^+} = \{p_1, \ldots, p_n, \overline{q}_{n+1}, \ldots, \overline{q}_m\}$.
2: create the matrix $M_P \in \mathbb{R}^{m \times m}$ representing P^+
3: create the initial matrix $M_0 \in \mathbb{R}^{m \times h}$
4: $M = M_0, U = \theta(M_P M)$ {refer to Definition 4}
5: **while** $U \neq M$ **do**
6: $M = U, U = \theta(M_P M)$ {refer to Definition 4}
7: **end while**
8: $V =$ find stable models of P {refer to Algorithm 3}
9: **return** V

Algorithm 3 Find stable models of P

Input: interpretation matrix M
Output: a set of vectors V representing the stable models of P

1: $V = \emptyset$
2: **for** i from 1 to h **do**
3: $v = (a_1, \ldots a_n, a_{n+1}, \ldots, a_m)^T$ (i^{th}-column of M)
4: **for** j from $n+1$ to m **do**
5: $\overline{q}_j = row_j(M)$
6: **for** l from 1 to n **do**
7: **if** $row_l(M) = q_j$ **then**
8: **if** $a_l + a_j \neq 1$ **then** break;
9: **end if**
10: **end for**
11: **if** $l \leq n$ **then** break;
12: **end for**
13: **if** $j \leq m$ **then** break;
14: **else** $V = V \cup \{v\}$
15: **end for**
16: **return** V

This method requires extra steps on transforming and finding stable models of a program that is represented in Algorithm 3. As we can see, Algorithm 3 loops over each interpretation vector of the fixed point of M which we obtain by applying matrix multiplication and thresholding. The main idea behinds this algorithm is to verify the consistency of each interpretation $I^+ (= I \cup \bar{I})$ that does not contain 1s for both positive and negative forms of an atom. This is done by the condition in line 8 of Algorithm 3 that tests whether the sum of values (corresponding to positive and negative forms of an atom in P) is 1 or not.

In addition, the initial matrix size grows exponentially by the number of negations $m - n$. Therefore, this representation requires a lot of memory and the algorithm performs considerably slower than the method for definite programs if there are many negations appearing in the program. Nevertheless, we will later show that this method still has the advantage when there are a small number of negations.

Sparse Representation of Logic Programs

The idea of representing logic programs in vector spaces could minimize the work with symbolic computation and utilize better computing performance. Besides that, this method copes with the curse of dimension when a matrix representing logic programs becomes very large. Previous works on this topic only consider dense

matrices for their implementation and it seems not very impressive in terms of performance even on small datasets [16]. To solve this problem, this paper focuses on analyzing the sparsity of logic programs in vector spaces and proposes improvement using sparse representation for logic programs. Additionally, we analyze and verify different sparse representations to conclude which format is efficient for logic programs in terms of memory cost.

Sparsity of Logic Programs in Vector Spaces

A sparse matrix is a matrix in which most of the elements are zero. The level of sparseness is measured by sparsity which equals the number of zero-valued elements divided by the total number of elements [6]. Because there are a large number of zero elements in sparse matrices, we can save the computation by ignoring these zero values [12]. According to the conversion method of linear algebraic approach, we can calculate the sparsity of a program P.[2] This calculation is done by counting the number of non-zero-valued elements of each rule in P, then let 1 minus the fraction of the number of non-zero-valued elements and the matrix size.

By definition, the sparsity of a program P is computed by the following equation:

$$\text{sparsity}(P) = 1 - \frac{\sum_{r \in P} |body(r)|}{n^2}, \tag{6}$$

where n is the number of elements in B_P and $|body(r)|$ is the length of body of rule r.

Accordingly, the representation matrix becomes a high level of sparsity if the matrix size becomes larger, while the length of the body rule is insignificant. In fact, a rule r in a logic program rarely has a body length approx n, therefore, $|body(r)| \ll n$. In short, we can say that the matrix representation of a logic program according to the linear algebraic approach is sparse in most cases.

Converting Logic Programs to Sparse Matrices

Sparse matrix computation is very important due to the large number of zero elements in real-world matrix data; therefore, compaction techniques are used to reduce the amount of storage, memory accesses, and computation [6]. Among several sparse storage formats, we select the three formats coordinate, compressed sparse row (CSR) and block compressed sparse row (BSR) [5] which are the most general, efficient, robust, and widely adopted by many programming libraries.

Because the matrix representation of a logic program P is sparse, applying Algorithm 1 and Algorithm 2 on sparse representation is remarkably faster than the dense matrix. Moreover, sparse representation saves the memory space as well, therefore enabling the ability to deal with a large scale KBs.

[2] We only consider the programs in Definitions 1 and 6.

The Coordinate Format

The COO format is the most simple idea of sparse matrix format which represents each non-zero element by a tuple of a row index, a column index, and the value of the element. That means the COO format uses 2 arrays of coordinates and 1 array of values. The length of these arrays is equal to the number of non-zero elements. The first array stores the row index of each value, and the second array stores the row and column indices of each value, while the third array stores the values in the original matrix. We can imagine that the *i*th non-zero element in a matrix is represented by a 3-tuple extracted from these 3 arrays at index *i*.

Example 3 illustrates sparse representation in the COO format for the program *P* in Example 1. We should note that in Example 3, zero-based indexing[3] is used and we follow row-major order.[4]

Example 3 The COO representation for *P* in Example 1 becomes:

Row index	0	0	1	2	3	4	5	5	6	6
Col index	5	6	4	3	3	4	1	2	3	4
Value	1.0	1.0	1.0	1.0	1.0	1.0	0.5	0.5	0.5	0.5

This format is the most simple and flexible for general-purpose usage. The storage requirement for this format is $O(3 \times \eta_z)$ where η_z is the number of non-zero elements. Because of the generality, we often use the COO format as the baseline to evaluate other sparse representations.

The Compressed Sparse Row Format

The CSR format is an improvement of the COO format. Noticeably, in the row index array of the COO format, a value can be repeated consecutively because the non-zero elements may appear in the same row many times. We may reduce the size of the row index array by considering the CSR format. In this format, while the column index and the value arrays remain the same, we compress the row index array by storing the index of the row only where non-zero elements appear. That means we do not need to store two consecutive 0s and two consecutive 5s as in Example 3. Instead, we store the index of the next row, then finally point the last index to the end of the row (which equals the number of non-zero elements). Concretely in the row index array, the first element is the starting index which is 0. The last element is an extra element to indicate the end of this array which is equal to the number of non-zero elements. We need two consecutive values in the row index array to extract the non-zero elements in this row. To be specific, we need to interpret

[3] The initial element of a sequence is assigned the index 0.

[4] In row-major order, the consecutive elements of a row reside next to each other.

row_start and *row_end* of the *i*th row from the compressed value in *row_index* array: $row\_start_i = row\_index[i], row\_end_i = row\_index[i+1]$.

Example 4 The CSR representation for P in Example 1 becomes:

Row index	0	2	3	4	5	6	8	10		
Col index	5	6	4	3	3	4	1	2	3	4
Value	1.0	1.0	1.0	1.0	1.0	1.0	0.5	0.5	0.5	0.5

Example 4 illustrates this method. For the first row ($i = 0$), we have $row\_start_0 = 0, row\_end_i = 2$, then we extract two values 0 and 1 for the non-zero element in the first row. These *start* and *end* will be used to extract column index and value of non-zero elements. Similarly, the second row ($i = 1$), we have $row\_start_1 = 2, row\_end_1 = 3$ then we have only one non-zero element at index 2. Continue this interpretation until we reach the final row ($i = 6$), we have $row\_start_6 = 8, row\_end_6 = 10$ then we extract last two non-zero elements at index 8 and 9 for the final row.

For a sparse matrix of the size $m \times n$, the CSR format saves on memory compared to the dense format only when $\eta_z < (m(n-1)-1)/2$ (where η_z is number of non-zero elements). Compared to the COO format, the CSR format uses less numbers in the row index array only when $m + 1 < \eta_z$. This is because the actual size of the row index array is $m + 1$. Therefore, the space complexity of the CSR format is $O(2 \times \eta_z + m + 1)$.

There is another format compressed sparse column (CSC) which is similar to the CSR. The only difference is that the CSC enumerates non-zero elements following the column-major order[5] and compress the column index array. Hence, the space complexity of the CSC is $O(2 \times \eta_z + n + 1)$. In the case of logic programs, the matrices are square so that these two formats are identical.

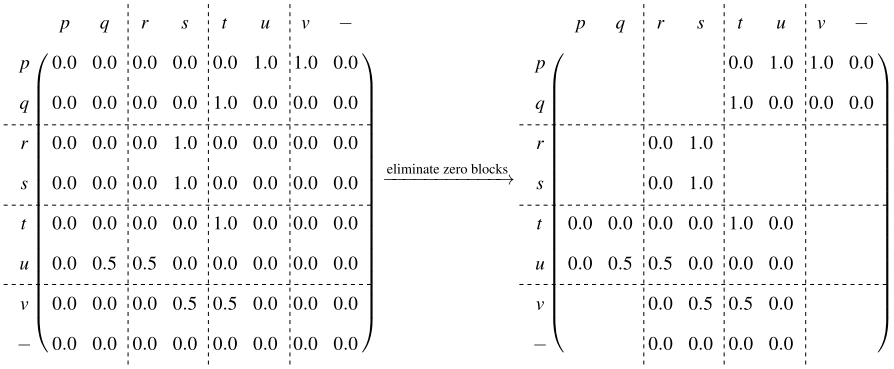
The Block Compressed Sparse Row Format

There is another sparse representation BSR which stores a two-dimensional square block of primitive data types instead of storing a single value. The dimension of the square block is d_b then the matrix is divided into multiple blocks of the size $d_b \times d_b$. In case that the dimension of the matrix is not a multiple of the d_b, we need to add a zero column or row to the matrix. For example, the matrix program in Example 1 has the dimension 7×7 and the d_b is 2, we need to pad the matrix to the dimension 8×8. Then, we divide the padded matrix into 16 blocks of the dimension $d_b \times d_b$. In the BSR, the format only stores non-zero blocks and uses the same way to index each block as in the CSR. Let us consider the BSR format for the logic program P in

[5] In the column-major order, the consecutive elements of a column reside next to each other, in contrast to row-major order.

Example 1, we can identify 8 non-zero blocks in the matrix. The illustration of these steps and the BSR representation of P are presented in Example 5.

Example 5 Illustration of block representation and the BSR representation for P in Example 1 are following:



Row index	0	2	3	6	8			
Col index	2	3	1	0	1	2	1	2
Block	B_{13}	B_{14}	B_{22}	B_{31}	B_{32}	B_{33}	B_{42}	B_{43}
Block value	0 1 1 0	1 0 0 0	0 1 0 1	0 0 0 1/2	0 0 0 1/2	1 0 0 0	0 1/2 0 0	1/2 0 0 0

Note that in each block, we store all the numbers following an exact order, row-major order in this example. If we follow the column-major order, the block value vector may be different, for example, the block B_{22} in the column-major order is 0 0 1 1.

Noticeably, this format is not efficient in this example because it stores many blocks with only 1 or 2 non-zero elements. In fact, this format only shows its advantages in case the matrix is highly concentrated in a few blocks. In other words, if the matrix has η_z non-zero elements and η_b non-zero blocks of the size $d_b \times d_b$ then the BSR performs the best in case $\eta_z \approx \eta_b \times {d_b}^2$.

Assume we have a sparse matrix of the size $m \times n$. In the matrix, there are η_z non-zero elements and η_b non-zero blocks of the size $d_b \times d_b$. Note that in the BSR format, we only need to store the indices of non-zero blocks and all values in those blocks. So, we can consider it as a CSR matrix where each non-zero block (in the BSR format) is a single non-zero element (in the CSR format) that the matrix size is $\left\lceil \frac{m}{d_b} \right\rceil \times \left\lceil \frac{n}{d_b} \right\rceil$, where $\lceil\ \rceil$ is the ceiling function. Accordingly, the space complexity of the BSR format is $O(\left\lceil \frac{m}{d_b} \right\rceil + 1 + \eta_b + \eta_b \times {d_b}^2)$.

Which Format is the Best for Logic Programs?

As we can see in Example 4, the row index array now has only 8 indices rather than 10 in Example 3. We save storing repeatedly indices in the row index array by storing only the position where it starts and ends. Accordingly, the CSR can be considered more economical than the COO but it comes with the cost that non-zero elements must follow row-major order while a strict order is not necessary for the COO format. Fortunately, in the case of linear algebraic methods for fixed-point computation, we do not need to update the program matrix frequently. Then the CSR format will be a better choice over the COO format. In fact, we can save up to 25% of the size of the row index array using the CSR format as will be illustrated in the experiments. The BSR format takes advantage over the CSR format when the program matrix is concentrated in a few non-zero blocks. Unfortunately, it is not very often in the case of program matrices. The experiments section will reveal which kind of logic programs will be beneficial from this sparse format. Accordingly, we propose the CSR format is the ideal sparse representation for linear algebraic computation methods.

Complexity Analysis

In this section, we analyze the time and space complexity of the linear algebraic methods for computing fixed points as defined in Algorithm 1 and Algorithm 2.

Linear Algebraic Method for Definite Programs

Assume that a definite program P has a matrix representation $M_P \in \mathbb{R}^{n\times n}$ and the matrix has η_z non-zero elements.[6]

Proposition 1 *The space complexity of linear algebraic method for definite programs is*

1. $O(n^2 + n)$ *for dense format*,
2. $O(\eta_z + n)$ *for sparse format*.

Proof Obviously, we have to store the program matrix and the interpretation vector. As defined, the program matrix size is $n \times n$ and the interpretation vector size is $n \times 1$. Note that only the program matrix can be stored in the sparse format while the interpretation vector must be stored in dense format. □

Proposition 2 *The time complexity of linear algebraic method for definite programs is*

[6] The matrix size depends on the number of literals linearly.

1. $O(n^3)$ *for dense format*,
2. $O(\eta_z \times n)$ *for sparse format*.

Proof Similar to the T_P-operator the main loop of Algorithm 1 repeats n times in the worst case. In addition, the complexity of each loop depends on the matrix multiplication between a matrix of the size $n \times n$ and a vector of the size $n \times 1$, so the multiplication takes $O(n^2)$ for dense format and $O(\eta_z)$ for sparse format.

Theoretically, if the program matrix is sparse, methods using sparse format outperform methods using the dense format in both time and space complexity. □

Linear Algebraic Method for Normal Programs

Let us consider a normal program P which has k negations. Assume that P has a matrix representation $M_P \in \mathbb{R}^{n \times n}$ and the matrix has η_z non-zero elements.[7]

Proposition 3 *The space complexity of the linear algebraic method for normal programs is*

1. $O(n^2 + n \times 2^k)$ *for dense format*,
2. $O(\eta_z + n \times 2^k)$ *for sparse format*.

Proof Similar to the methods for definite programs, the size of the program matrices is the same. The cost for storing the interpretation matrix exponentially depends on the number of negations because we have to consider all the combinations according to the Algorithm 2. Therefore, it is the limitation of the method that we can handle programs with a limited number of negations. □

Proposition 4 *The time complexity of linear algebraic method for normal programs is*

1. $O(n^3 \times 2^k + n^2 \times (2^k - 1))$ *for dense format*,
2. $O(\eta_z \times n \times 2^k + n^2 \times (2^k - 1))$ *for sparse format*.

Proof Similar to previous proof, the main loop of Algorithm 2 repeats n times in the worst case. Each loop involves the multiplication between a matrix of the size $n \times n$ and a matrix of the size $n \times 2^k$. Hence, the complexity of Algorithm 2 is $O(n^3 \times 2^k)$ if we use dense format and $O(\eta_z \times n \times 2^k)$ if we use sparse format. Then, we have to apply the Algorithm 3 to find the stable model. This algorithm loops over all 2^k combinations to verify the model in case $k > 0$. If $k = 0$ the loop is not executed. Each verification takes 2 nested loops over n times. Therefore, the complexity of this algorithm is $O(n^2 \times (2^k - 1))$. □

[7] Usually n is larger than the number of literals in P because we have to do several standardized steps. To simplify, we can assume that n linearly depends on the number of literals in P.

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

Body length	0	1	2	3	4	5	6	7	8
Allocated proportion	$< n/3$	4%	4%	10%	40%	35%	4%	2%	1%

Obviously, if k is small, then we obtain the same complexity as the method for definite programs. If k is considerably large, then both the space and time complexity are infeasible, so that is the limitation of the method. Although both formats are exponential in terms of time and space complexity, sparse representation improves a lot in general cases.

Experimental Results

In this section, we report the results of two experiments on finding the least models of definite programs and computing stable models of normal programs.To evaluate the performance of linear algebraic methods, we compared the implementations of Algorithm 1 and Algorithm 2 with (i) the T_P-operator and (ii) Clasp (Clingo v5.4.1 running with flag –mode=clasp). Our implementations are done with (*iii*) dense matrices and (iv) sparse matrices. Except Clasp, all implementations (i), (iii) and (iv) are implemented on C++ with CPU x64 as a targeted device. In (i), we implement the operator using hashset instead of list for better set operations performance. To avoid ambiguity with the original definition of the T_P-operator, we will call (i) as Hashset method from now on in this section. (ii) is the solver of Clingo which is a powerful Answer Set Programming (ASP) solver developed at the University of Potsdam [10]. In terms of matrix representations and operators for (iii) and (iv), we use Eigen 3 library [11] with the default backend. The computer running experiments has the following configurations: CPU: Intel Core i7-4770 (4 cores, 8 threads) @3.4 GHz; RAM: 16 GB DDR3 @1333 MHz; GPU: NVIDIA GTX 1080; Operating system: Ubuntu 18.04 LTS 64 bit.

Focusing on analyzing the performance of sparse representation, we first evaluate our method by conducting experiments on randomized logic programs. We use the same method of LP generation conducted in [16] that the size of a logic program is defined by the size $n = |B_P|$ of the Herband base B_P and the number of rules $m = |P|$ in P. The number of facts (rules with the body length is 0) of the logic program is limited by $n/3$. The other rules are uniformly generated based on the length of their rule body (maximum length is 8) according to Table 1.

According to Algorithms 1 and 2, we have to transform logic programs to standardized programs to encode them as matrices. Hence, in the experiments, we also track the size of the Herbrand base of a standardized program which is equal to the actual square matrix size and denote it by n'.

We further generate denser matrices in order to analyze the efficacy of the sparse method. While keeping the same proportion of facts and rules with the body length of 1 and 2, we generate the rest $70 \sim 80\%$ rules such that their body length is around

Table 2 Details of experimental results on definite programs of Hashset method, Clasp and linear algebraic methods (with dense and sparse representation)

n	m	n'	Sparsity	Hashset method	Clasp	Dense matrix	Sparse matrix
1000	5000	5788	0.99	0.04	0.17	2.06	**0.01**
1000	10,000	10,799	0.99	0.12	0.29	17.99	**0.01**
1600	24,000	25,198	0.99	0.39	1.85	73.35	**0.04**
1600	30,000	31,285	0.99	0.48	2.54	116.12	**0.06**
2000	36,000	37,596	0.99	0.75	3.17	155.43	**0.07**
2000	40,000	41,936	0.99	0.98	5.16	187.65	**0.07**
10,000	120,000	127,119	0.99	18.56	9.07	–	**0.38**
10,000	160,000	167,504	0.99	25.65	15.77	–	**0.48**
16,000	200,000	211,039	0.99	57.02	19.97	–	**0.86**
16,000	220,000	231,439	0.99	60.44	24.78	–	**0.94**
20,000	280,000	297,293	0.99	104.99	30.57	–	**0.90**
20,000	320,000	337,056	0.99	108.59	34.40	–	**1.06**

n' is the actual matrix size after transformation. Time unit is second

The best results are shown in bold

5% of the number of propositions. This method leads to the lower sparsity level of generated matrices with approximate 0.95.

Also based on the generation method for definite programs, we generate normal programs by randomly changing literals to negations and limit the number of negations, denoted by k, such that $4 \leq k \leq 8$. The important difference from [16] is that we do experiments on much larger n and m, because our method, which is implemented on C++, is dramatically more efficient than Nguyen et al.'s implementation using Maple. The largest size of the logic program in this experiment reaches thousands of propositions and hundreds of thousands of rules. Further, we also compare our method with one of the best ASP solvers—Clasp [10] running in the same environment. All methods are conducted 30 times on each LP to obtain mean values of execution time.

In addition, we also conduct a further experiment using non-random problems with definite programs using the transitive closure problem. The graph we use is selected from the Koblenz network collection [15]. This dataset contains binary tuples and we compute the transitive closure of them using the following rules:

- $path(X, Y) \leftarrow edge(X, Y)$
- $path(X, Y) \leftarrow edge(X, Z) \wedge path(Z, Y)$

Definite programs

The final results on definite programs are illustrated in Table 2 and Fig. 1.

We can see in the results that the dense matrix method is slowest and being unable to run with very large programs that is why the data for this method is not displayed if the number of rules is larger or equal to 120,000. We should mention that

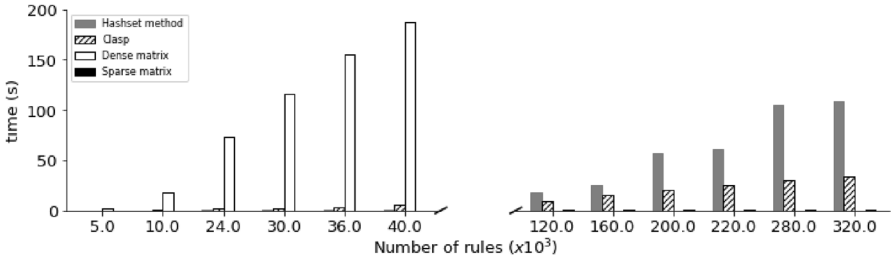


Fig. 1 Comparison of execution time between Hashset method, Clasp and linear algebraic methods (with dense and sparse representation) on definite programs

Table 3 Details of experimental results on definite programs (with lower sparsity level) of Hashset method, Clasp and linear algebraic methods (with dense and sparse representation)

n	m	n'	Sparsity	Hashset method	Clasp	Dense matrix	Sparse matrix
1000	5000	5876	0.95	0.10	0.39	2.31	**0.04**
1000	10,000	10,243	0.95	0.36	0.92	17.59	**0.05**
1600	24,000	25,712	0.95	0.95	2.25	70.09	**0.16**
1600	30,000	31,430	0.95	1.18	3.01	120.52	**0.38**
2000	36,000	36,612	0.95	1.73	4.78	152.91	**0.55**
2000	40,000	41,509	0.95	2.04	6.33	192.36	**0.63**
10,000	120,000	125,692	0.95	27.80	10.89	–	**1.08**
10,000	160,000	166,741	0.95	47.24	18.60	–	**2.29**
16,000	200,000	210,526	0.95	89.55	21.71	–	**3.79**
16,000	220,000	230,178	0.95	108.13	28.54	–	**4.86**
20,000	280,000	298,582	0.95	144.80	35.09	–	**5.34**
20,000	320,000	335,918	0.95	183.53	42.84	–	**5.92**

n' is the actual matrix size after transformation. Time unit is second

The best results are shown in bold

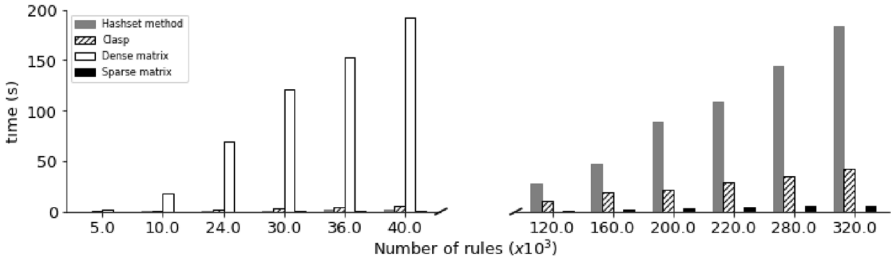


Fig. 2 Comparison of execution time between Hashset method, Clasp and linear algebraic methods (with dense and sparse representation) on definite programs with lower sparsity level

the number of rules m is used as horizontal axis in the Fig. 1 similar to the experiments in [16]. The reason for choosing n and m is to generate actual matrix size n' increasing linearly with two different levels: smaller scale ($n < 10,000$) and larger scale ($n > 10,000$). The same parameters are used for other experiments using the

Table 4 Details of experimental results on the transitive closure problem of Hashset method, Clasp and sparse representation approach

Data name (|V|, |E|)	n	m	n'	Sparsity	Hashset method	Clasp	Sparse matrix
Club membership (65, 95)	1200	14,492	15,600	0.99	0.84	0.34	**0.02**
Cattle (28, 217)	1512	20,629	21,924	0.99	0.95	0.51	**0.04**
Windsurfers (43, 336)	4324	99,788	103,776	0.99	3.65	3.37	**0.18**
Contiguous USA (49, 107)	4704	113,003	117,600	0.99	4.29	3.88	**0.18**
Dolphins (62, 159)	7564	230,861	238,266	0.99	12.31	9.38	**0.40**
Train bombing (64, 243)	8064	254,259	262,080	0.99	15.23	10.63	**0.45**
Highschool (70, 366)	9660	333,636	342,930	0.99	19.96	15.80	**0.66**
Les Miserables (77, 254)	11,704	445,006	456,456	0.99	27.79	21.96	**0.83**

n' is the actual matrix size after transformation. Time unit is second

The best results are shown in bold

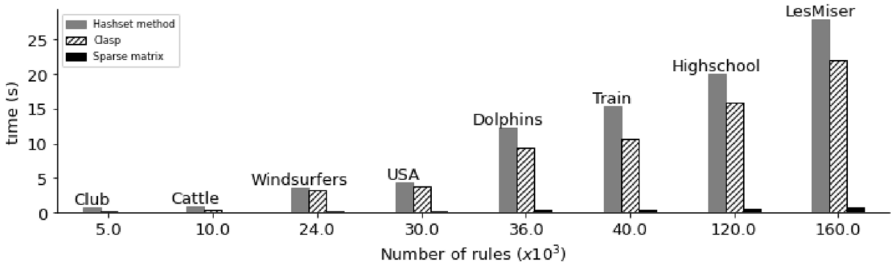


Fig. 3 Comparison of execution time between Hashset method, Clasp and linear algebraic methods (with dense and sparse representation) on definite programs with Transitive closure problem using Koblenz network datasets

random generated method. Overall, the sparse matrix method is very efficient which is 10–15 times faster than Clasp.

The benchmark results on denser matrix are presented in Table 3 and Fig. 2. As can be seen in the results, denser matrices require more computation for the sparse matrix method, while they do not affect the same scale on other competitors. Despite that fact, the sparse matrix method still holds first place in this benchmark. In terms of analyzing the sparseness level of logic programs, we hardly find a program in which the sparsity is less than 0.97. This observation strongly encourages the use of sparse representation for logic programs.

In the next experiment, we show the comparison for computing transitive closure. We assume that a dataset contains *edges* (tuples of *nodes*), then first perform grounding two rules of defining *path*. The obtained results are demonstrated in Table 4 and Fig. 3. In this non-randomized problem, we can see that the matrix representations are very sparse. Therefore, it is no doubt that the sparse matrix method outperforms

Table 5 Details of experimental results on normal programs of Hashset method, Clasp and linear algebraic methods (with dense and sparse representation)

n	m	n'	k	Sparsity	Hashset method	Clasp	Dense matrix	**Sparse matrix**
1000	5000	6379	8	0.99	0.07	0.31	3.96	**0.01**
1000	10,000	12,745	8	0.99	0.18	1.09	28.18	**0.02**
1600	24,000	30,061	8	0.99	0.55	3.27	105.49	**0.05**
1600	30,000	36,402	7	0.99	0.68	4.31	168.80	**0.08**
2000	36,000	42,039	5	0.99	1.24	6.72	203.27	**0.09**
2000	40,000	48,187	8	0.99	1.54	7.18	256.97	**0.10**
10,000	120,000	171,967	6	0.99	27.31	7.68	–	**0.71**
10,000	160,000	207,432	7	0.99	32.55	24.70	–	**0.84**
16,000	200,000	250,194	5	0.99	70.31	30.72	–	**1.56**
16,000	220,000	278,190	6	0.99	86.52	35.40	–	**1.83**
20,000	280,000	357,001	4	0.99	133.79	50.19	–	**1.92**
20,000	320,000	396,128	4	0.99	150.34	58.61	–	**2.11**

n' is the actual matrix size after transformation. k is the number of negations. Time unit is second

The best results are shown in bold

the dense matrix method. Accordingly, we only highlight the efficiency of sparse representation and omit the dense matrix approach. Surprisingly, the sparse matrix method surpasses Clasp once again in this experiment by a large margin.

As can be witnessed in the results, the dense matrix method is the slowest, even slower than the hashset method, in terms of computation time due to wasting computation on a huge amount of zero elements. This could be explained by the high level of sparsity of logic programs provided in Tables 2, 3 and 4. Moreover, large dense matrices consume a huge amount of memory, therefore the method is unable to run with a large scale matrix size. Overall, the sparse matrix method is effective in computing the fixed points of definite programs. On the other hand, the performance would be improved if we use GPU accelerated code and exploit parallel computing power. The results indicate that using sparse representation for logic programs opens the gate to deal with large-scale logic programs.

Normal Programs

The goal of this experiment is to highlight the enhancement of the sparse representation in terms of computing the stable models in normal logic programs. To generate normal programs for this benchmark, we use the same method to generate definite programs, then randomly select some rules and set some atoms in the rule body to negations. In our current method, since the number of columns in the initial matrix (Definition 7) grows exponentially by the number of negations, we limit the number

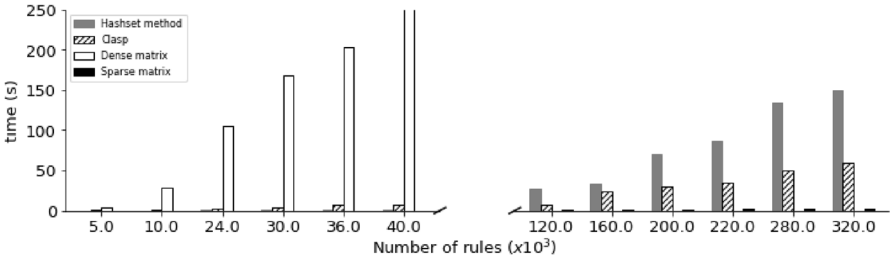


Fig. 4 Comparison of execution time between Hashset method, Clasp and linear algebraic methods (with dense and sparse representation) on normal programs

Table 6 Details of experimental results on normal programs (with lower sparsity level) of Hashset method, Clasp and linear algebraic methods (with dense and sparse representation)

n	m	n'	k	Sparsity	Hashset method	Clasp	Dense matrix	Sparse matrix
1000	5000	6385	7	0.95	0.17	0.37	3.78	**0.11**
1000	10,000	12,294	8	0.95	0.24	1.49	30.06	**0.19**
1600	24,000	33,172	7	0.95	0.68	3.78	102.54	**0.22**
1600	30,000	35,091	8	0.95	0.77	5.91	174.52	**0.35**
2000	36,000	44,145	8	0.95	2.32	7.10	197.30	**0.41**
2000	40,000	49,080	7	0.95	3.27	8.67	250.09	**0.49**
10,000	120,000	181,550	8	0.95	36.95	10.45	–	**3.25**
10,000	160,000	203,576	6	0.95	54.11	33.19	–	**4.02**
16,000	200,000	246,159	4	0.95	86.36	48.19	–	**7.22**
16,000	220,000	282,734	5	0.95	106.03	56.91	–	**8.31**
20,000	280,000	365,190	4	0.95	163.06	78.18	–	**9.02**
20,000	320,000	387,094	4	0.95	202.55	84.33	–	**11.52**

n' is the actual matrix size after transformation. k is the number of negations. Time unit is second

The best results are shown in bold

of negations in this benchmark by 8[8] as specified in the experiment setup. The experiment results show that the sparse method can be applied to normal logic programs with a small number of negations. The performance gain from this improvement is potential for further developing more efficient algorithms.

First, we perform benchmarks on normal programs which has 0.99 sparsity level. Table 5 and Fig. 4 illustrate the execution time in detail. As can be witnessed in the results, the sparse matrix method is still faster than Clasp but with a smaller scale than it did in definite programs. It is needed to mention that the initial matrix size is remarkably larger due to the occurrence of negations. We have to initialize all possible combinations of atoms that appear with their negation form in the program. There is no doubt that with a larger number of negations, the space complexity of the

[8] The dimension of the initial matrix depends on k and grows exponentially if k increases. At this moment, we are able to handle k up to 16 and in some specific cases depending on the matrix size and memory capacity, k could be larger (up to 24).

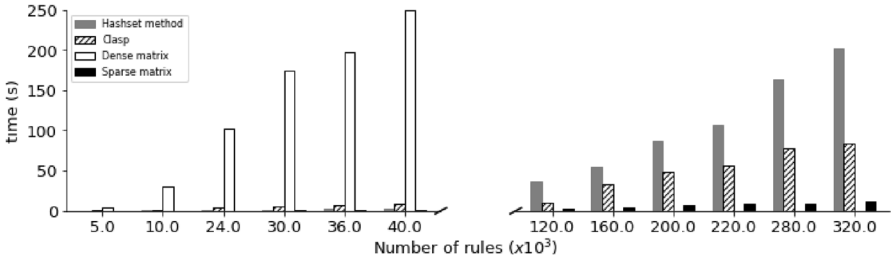


Fig. 5 Comparison of execution time between Hashset method, Clasp and linear algebraic methods (with dense and sparse representation) on normal programs with lower sparsity level

linear algebraic method is exponential. Accordingly, the performance of the sparse matrix method is better than Clasp when there are a small number of negations.

In the next experiments, we compare different methods on denser matrices. Table 6 and Fig. 5 present the data for this benchmark. Once again, with a limited number of negations, the sparse matrix method holds the winner position.

Noticeably, execution time on normal programs is generally greater than that on definite programs. This is obvious because we have a larger size of initial matrices as well as the need for extra computation on transforming and finding the least models as described in Algorithm 2. Then, the weakness of the linear algebraic method is that we have to deal with all combinations of truth assignments to compute the stable model. Accordingly, the column size of the initial matrix exponentially increases by the number of negations. Thus, in the benchmark on randomized programs, we limit the number of negations for all benchmarks so that the matrix can fit in memory. This limitation will become clearer in real problems which have many negations. This is a major problem that we are investigating to do further research.

Sparse Representations Comparison

In this experiment, we focus on space complexity of different sparse representations for logic programs. The benchmark is done on the same datasets in the previous results. To highlight the efficiency of sparse formats, we compare the memory space in Bytes to store the program matrices using the three mentioned methods in Sect. 3 including: COO, CSR and BSR. The BSR will be analyzed with two different d_b: 2×2 and 4×4. The figures for the COO format will be considered as the baseline to compare these other spare formats (Fig. 6).

The experimental results for definite programs, definite programs for the transitive closure problem and normal programs are illustrated in Tables 7, 8 and 9 respectively. As can be witnessed in the data, the CSR format is better than the baseline COO 20–30% in terms of storage usage. It is a remarkable saving because we only need to store fewer numbers in the row index array as explained in Sect. 3. On the other hand, the data for the BSR format show an increase in memory usage by a large margin. This is due to the program matrices are not concentrated and we have to store many blocks with zero included (Figs. 7, 8).

Table 7 Comparison of different sparse representations in terms of the memory size on definite programs in Table 2

n	m	n'	η_z	COO	CSR/CSC	BSR 2×2	BSR 4×4
1000	5000	5788	24,848	298,176 (100.00%)	221,940 (74.43%)	45,9876 (154.23%)	1,414,952 (474.54%)
1000	10,000	10,799	50,805	609,660 (100.00%)	449,640 (73.75%)	935,720 (153.48%)	2,879,924 (472.38%)
1600	24,000	25,198	122,466	1,469,592 (100.00%)	1,080,524 (73.53%)	2,258,796 (153.70%)	7,023,964 (477.95%)
1600	30,000	31,285	154,395	1,852,740 (100.00%)	1,360,304 (73.42%)	2,850,912 (153.88%)	8,870,200 (478.76%)
2000	36,000	37,596	185,092	2,221,104 (100.00%)	1,631,124 (73.44%)	3,418,412 (153.91%)	10,668,648 (480.33%)
2000	40,000	41,936	208,352	2,500,224 (100.00%)	1,834,564 (73.38%)	3,851,612 (154.05%)	12,019,048 (480.72%)
10,000	120,000	127,119	606,233	7,274,796 (100.00%)	5,358,344 (73.66%)	11,201,120 (153.97%)	35,233,888 (484.33%)
10,000	160,000	167,504	817,728	9,812,736 (100.00%)	7,211,844 (73.49%)	15,130,228 (154.19%)	47,646,464 (485.56%)
16,000	200,000	211,039	1,009,279	12,111,348 (100.00%)	8,918,392 (73.64%)	18,647,660 (153.97%)	58,712,052 (484.77%)
16,000	220,000	231,439	1,116,473	13,397,676 (100.00%)	9,857,544 (73.58%)	20,645,480 (154.10%)	65,034,080 (485.41%)
20,000	280,000	297,293	1,442,651	17,311,812 (100.00%)	12,730,384 (73.54%)	26,730,648 (154.41%)	84,262,608 (486.73%)
20,000	320,000	337,056	1,649,792	19,797,504 (100.00%)	14,546,564 (73.48%)	30,563,432 (154.38%)	96,370,464 (486.78%)

η_z is the number of non-zero elements in the program matrix. Memory unit is Bytes

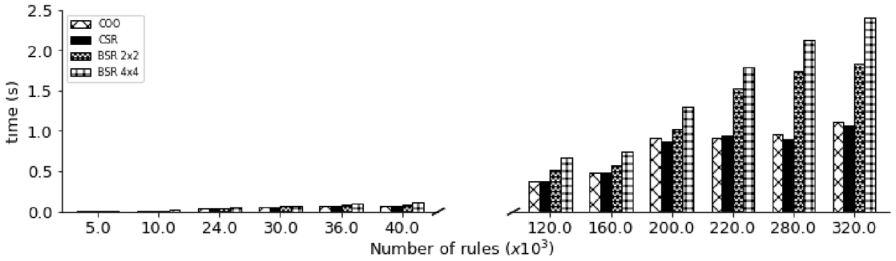


Fig. 6 Comparison of execution time between different sparse representations on definite programs

Accordingly, in general cases, the CSR format is the best option in terms of space efficiency. We also understand that the BSR format is efficient when the matrix is highly concentrated in a way that non-zero elements are stored in as small number of blocks as possible. In this experiment, we also conduct the comparison on special logic programs. For example, consider the program P and its matrix representation that contains the following rules:

$$
\begin{aligned}
&r_1 \leftarrow r_3 \wedge r_4,\\
&r_2 \leftarrow r_3 \wedge r_4,\\
&\ldots,\\
&r_{n-3} \leftarrow r_{n-1} \wedge r_n,\\
&r_{n-2} \leftarrow r_{n-1} \wedge r_n
\end{aligned}
\qquad
\begin{array}{c}
\\
r_1\\ r_2\\ r_3\\ r_4\\ r_5\\ r_6\\ \ldots
\end{array}
\begin{array}{c}
\begin{array}{ccccccc} r_1 & r_2 & r_3 & r_4 & r_5 & r_6 & \ldots \end{array}\\
\begin{pmatrix}
0 & 0 & 1/2 & 1/2 & 0 & 0 & \ldots\\
0 & 0 & 1/2 & 1/2 & 0 & 0 & \ldots\\
0 & 0 & 0 & 0 & 1/2 & 1/2 & \ldots\\
0 & 0 & 0 & 0 & 1/2 & 1/2 & \ldots\\
0 & 0 & 0 & 0 & 0 & 0 & \ldots\\
0 & 0 & 0 & 0 & 0 & 0 & \ldots\\
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots
\end{pmatrix}
\end{array}
$$

We can easily see that in this case, the program matrix contains only 2×2 blocks that will be ideal for the BSR 2×2 format. In this case, the block value matrix does not need to store zero elements while the indexing arrays for non-zero blocks are much less than the indexing arrays for non-zero elements. The data for this experiment is illustrated in Table 10. In the perfect case, the BSR can save up to 50% compared to the baseline COO format and is much more efficient than the CSR format (Fig. 9).

Scalability of Sparse Matrix on GPU

In this experiment, we compare the execution time of the sparse matrix implementation on CPU and GPU using definite programs. We use the same method for generating definite programs as presented. Additionally, we increase the body length of generated rules to obtain large-scale programs. The implementation on GPU is done using cuSPARSE.[9]

[9] CUDA version 10.0.130.

Table 8 Comparison of different sparse representations in terms of the memory size on definite programs for the transitive closure problem in Table 4

Data name (|V|, |E|)	n	m	n'	η_z	COO	CSR/CSC	BSR 2×2	BSR 4×4
Club membership (65, 95)	1200	14,492	15,600	42,692	512,304 (100.00%)	403,940 (78.85%)	673,840 (131.53%)	1,584,020 (309.20%)
Cattle (28, 217)	1512	20,629	21,924	60,697	728,364 (100.00%)	573,276 (78.71%)	960,928 (131.93%)	2,376,560 (326.29%)
Windsurfers (43, 336)	4324	99,788	103,776	296,530	3,558,360 (100.00%)	2,787,348 (78.33%)	4,664,212 (131.08%)	11,527,504 (323.96%)
Contiguous USA (49, 107)	4704	113,003	117,600	336,443	4,037,316 (100.00%)	3,161,948 (78.32%)	5,291,840 (131.07%)	12,504,344 (309.72%)
Dolphins (62, 159)	7564	230,861	238,266	688,483	8,261,796 (100.00%)	6,460,932 (78.20%)	10,840,292 (131.21%)	26,812,056 (324.53%)
Train bombing (64, 243)	8064	254,259	262,080	758,259	9,099,108 (100.00%)	7,114,396 (78.19%)	11,936,120 (131.18%)	29,479,572 (323.98%)
Highschool (70, 366)	9660	333,636	342,930	995,346	11,944,152 (100.00%)	9,334,492 (78.15%)	15,662,880 (131.13%)	38,723,832 (324.21%)
Les Miserables (77, 254)	11,704	445,006	456,456	1,328,658	15,943,896 (100.00%)	12,455,092 (78.12%)	20,868,772 (130.89%)	49,390,820 (309.78%)

η_z is the number of non-zero elements in the program matrix. Memory unit is Bytes

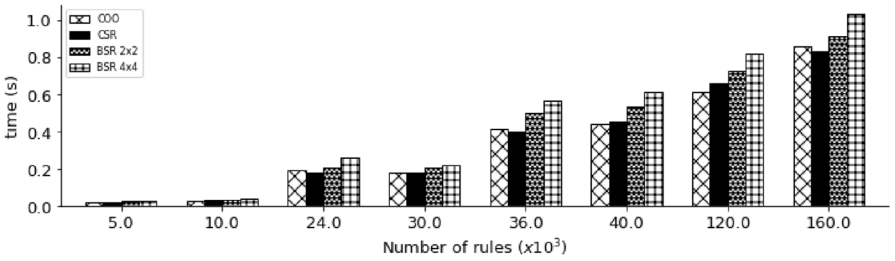


Fig. 7 Comparison of execution time between different sparse representations on definite programs for the transitive closure problem

As we can see in Fig. 10, the implementation on GPU is faster than that on CPU approximately 3–4 times. That is because sparse matrix computation usually does not reach maximum throughput on GPU. Thus, it is less scalable than dense computation. However, the sparse matrix computation is faster than the dense counterpart. We should note that we generate very large matrices which can not be fit in GPU memory if we store them in dense format. Accordingly, although sparse matrix computation is more difficult to scale up, using the sparse matrix is the ideal solution for large-scale logic programs in terms of both time and space complexity (Tables 11, 12).

Conclusion

In this paper, we analyze the sparsity of matrix representation for LP and then propose an improved implementation for logic programming in vector space using sparse matrix representation. The experimental results on computing the least models of definite programs demonstrate a very significant enhancement in terms of computation performance even when compared to Clasp. This improvement remarkably reduced the burden of computation in previous linear algebraic approaches for representing LP. The T_P-operator plays an important role in model construction for computation of definite and normal logic programs. Thus, improving the efficiency of fixed-point computation is the key to develop algorithms dealing with large-scale datasets. Although the current method requires a huge amount of memory to store all possible combinations of negated atoms, we witnessed considerable improvement when there are small numbers of negations. Moreover, matrix computation could be more accelerated using GPU. We have tested our implementation in this way, and obtained expected results too.

In addition to the improvement using sparse representation, we conducted experiments on different general-purpose sparse matrix representations and demonstrated the merits and demerits of each format. Accordingly, we propose to use the CSR in the linear algebraic methods of logic programs for both efficiency and generality.

Table 9 Comparison of different sparse representations in terms of the memory size on normal programs in Table 5

n	m	n'	k	η_z	COO	CSR/CSC	BSR 2×2	BSR 4×4
1000	5000	6379	8	26,147	313,764 (100.00%)	249,176 (79.42%)	487,336 (155.32%)	1514,852 (482.80%)
1000	10,000	12,745	8	54,814	657,768 (100.00%)	478,512 (72.75%)	1,019,432 (154.98%)	3,163,488 (480.94%)
1600	24,000	30,061	8	135,661	1,627,932 (100.00%)	1,149,288 (70.60%)	2,528,412 (155.31%)	7,840,124 (481.60%)
1600	30,000	36,402	7	183,703	2,204,436 (100.00%)	1,533,624 (69.57%)	3,402,460 (154.35%)	10,528,348 (477.60%)
2000	36,000	42,039	5	205,800	2,469,600 (100.00%)	1,726,400 (69.91%)	3,824,712 (154.87%)	11,829,348 (479.00%)
2000	40,000	48,187	8	238,597	2,863,164 (100.00%)	1,988,776 (69.46%)	4,437,184 (154.97%)	13,764,920 (480.76%)
10,000	120,000	171,967	6	716,115	8,593,380 (100.00%)	6,128,920 (71.32%)	13,523,496 (157.37%)	42,851,300 (498.65%)
10,000	160,000	207,432	7	917,746	11,012,952 (100.00%)	7,741,968 (70.30%)	17,043,484 (154.76%)	52,633,948 (477.93%)
16,000	200,000	250,194	5	1,129,348	13,552,176 (100.00%)	9,674,784 (71.39%)	21,203,960 (156.46%)	66,035,684 (487.27%)
16,000	220,000	278,190	6	1,547,360	18,568,320 (100.00%)	13,018,880 (70.11%)	29,028,448 (156.33%)	90,012,932 (484.77%)
20,000	280,000	357,001	4	1,841,749	22,100,988 (100.00%)	15,533,992 (70.29%)	34,219,356 (154.83%)	106,039,484 (479.80%)
20,000	320,000	396,128	4	2,092,310	25,107,720 (100.00%)	17,538,480 (69.85%)	39,012,940 (155.38%)	120,359,688 (479.37%)

η_z is the number of non-zero elements in the program matrix. Memory unit is Bytes

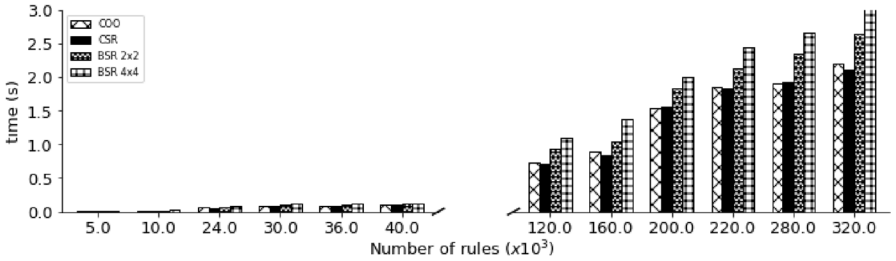


Fig. 8 Comparison of execution time between different sparse representations on normal programs

Table 10 Comparison of different sparse representations in terms of the memory size on special programs as defined above

n	m	n'	η_z	COO	CSR/CSC	BSR 2×2	BSR 4×4
1000	1000	1000	2000	24,000 (100.00%)	20,004 (83.35%)	12,004 (50.02%)	18,004 (75.02%)
1600	1600	1600	3200	38,400 (100.00%)	32,004 (83.34%)	19,204 (50.01%)	28,804 (75.01%)
2000	2000	2000	4000	48,000 (100.00%)	40,004 (83.34%)	24,004 (50.01%)	36,004 (75.01%)
10,000	10,000	10,000	20,000	240,000 (100.00%)	200,004 (83.34%)	120,004 (50.00%)	180,004 (75.00%)
16,000	16,000	16,000	32,000	384,000 (100.00%)	320,004 (83.33%)	192,004 (50.00%)	288,004 (75.00%)
20,000	20,000	20,000	40,000	480,000 (100.00%)	400,004 (83.33%)	240,004 (50.00%)	360,004 (75.00%)

η_z is the number of non-zero elements in the program matrix. Memory unit is Bytes

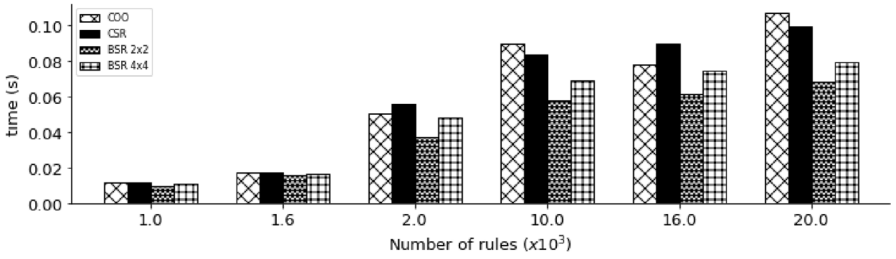
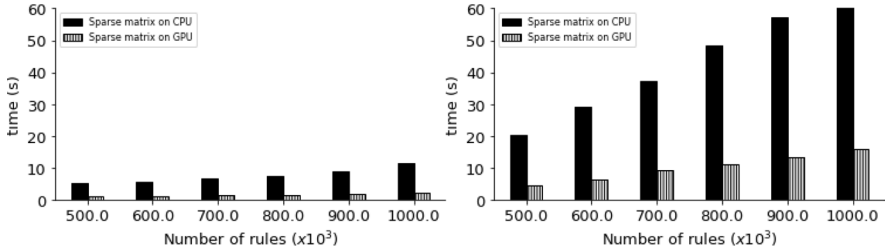


Fig. 9 Comparison of execution time between different sparse representations on special programs

If we need a flexible way to access and modify non-zero elements individually, we strongly recommend using the COO format. On the other hand, if we deal with special types of logic programs as demonstrated in Sect. 5, we can consider applying the BSR format or maybe other methods that meet the need.



(a) Definite programs with higher sparsity level (0.99)

(b) Definite programs with lower sparsity level (0.95)

Fig. 10 Comparison of execution time between sparse matrix implementations on CPU and GPU

Table 11 Details of experimental results of sparse matrix implementations on CPU and GPU (higher sparsity level)

n	m	Sparsity	Sparse matrix on CPU	Sparse matrix on GPU
100,000	500,000	0.99	5.12	1.26
110,000	600,000	0.99	5.69	1.40
120,000	700,000	0.99	6.63	1.58
130,000	800,000	0.99	7.57	1.75
140,000	900,000	0.99	9.12	2.04
150,000	1,000,000	0.99	11.39	2.29

Time unit is second

Table 12 Details of experimental results of sparse matrix implementations on CPU and GPU (lower sparsity level)

n	m	Sparsity	Sparse matrix on CPU	Sparse matrix on GPU
100,000	500,000	0.95	20.23	4.43
110,000	600,000	0.95	29.12	6.35
120,000	700,000	0.95	37.28	9.39
130,000	800,000	0.95	48.23	11.33
140,000	900,000	0.95	57.24	13.46
150,000	1,000,000	0.95	66.23	15.89

Time unit is second

Sato's linear algebraic method is based on a completely different idea to represent logic programs, where each predicate is represented in one matrix and an approximation method is used to compute the extension of a target predicate of a recursive program [24]. We should note that this approximation method is limited to a matrix size of 10,000, while our exact method is comfortable with 320,000. Further comparison is a future research topic, yet we could expect that Sato's method can also be enhanced by sparse representation.

The encouraging results open up room for improvement and optimization. Potential future work is to apply a sampling method to reduce the number of guesses in

the initial matrix for normal programs. An algorithm would be to prepare some manageable size of the initial matrix, and if all guesses fail then we do some local search and replace column vectors with new assignments and repeat it until a stable model is found. Using a gradient-based search algorithm in continuous vector spaces could be another potential approach [4], this method could also be beneficial from using sparse representation. In addition, the sparse method also can combine with the partial evaluation that has been introduced in [17]. Further research directions on implementing disjunctive LP and abductive LP should be considered to reveal the applicability of tensor-based approaches for LP. In our recent work, we have extended the use of program matrix transpose to realize abduction in vector spaces [19]. Additionally, more complex types of the program should be taken into account to be represented in vector space, for instance, 3-valued logic programs and answer set programs with aggregates and constraints.

Acknowledgements We would like to thank Prof. Taisuke Sato for his valuable comments and useful critiques of this research. This work has been supported in part by JSPS KAKENHI Grant Numbers 17H00763 and 18H03288. Tuan Nguyen Quoc has also been supported by Japan International Cooperative Agency "Innovative Asia".

References

1. Alferes, J.J., Leite, J.A., Pereira, L.M., Przymusinska, H., Przymusinski, T.C.: Dynamic updates of non-monotonic knowledge bases. J. Logic Progr. **45**(1–3), 43–70 (2000)
2. Aspis, Y.: A Linear Algebraic Approach to Logic Programming. Master thesis at Imperial College London (2018)
3. Aspis, Y., Broda, K., Russo, A.: Tensor-based abduction in horn propositional programs. In: CEUR Workshop Proceedings, vol. 2206, pp. 68–75 (2018)
4. Aspis, Y., Broda, K., Russo, A., Lobo, J.: 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, pp. 59–68 (2020)
5. Bell, N., Garland, M.: Efficient sparse matrix-vector multiplication on cuda. Technical report, Citeseer (2008)
6. Bunch, J.R., Rose, D.J.: Sparse Matrix Computations. Academic Press (2014)
7. Cohen, W.W.: Tensorlog: A differentiable deductive database (2016). arXiv:1605.06523
8. D'Asaro, F.A., Spezialetti, M., Raggioli, L., Rossi, S.: 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, pp. 855–859 (2020)
9. Funahashi, K.-I.: On the approximate realization of continuous mappings by neural networks. Neural Netw. **2**(3), 183–192 (1989)
10. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Wanko, P.: Theory solving made easy with clingo 5. In: OASIcs-OpenAccess Series in Informatics, vol. 52. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2016)

11. Guennebaud, G., Jacob, B., et al.: Eigen v3 (2010). http://eigen.tuxfamily.org
12. Gustavson, F.G.: Two fast algorithms for sparse matrices: multiplication and permuted transposition. ACM Trans. Math. Softw. (TOMS) **4**(3), 250–269 (1978)
13. Hitzler, P., Hölldobler, S., Seda, A.K.: Logic programs and connectionist networks. J. Appl. Logic **2**(3), 245–272 (2004)
14. Kowalski, R.: Logic for Problem Solving. Elsevier, North Holland (1979)
15. Kunegis, J.: Konect: the Koblenz network collection. In: Proceedings of the 22nd International Conference on World Wide Web, pp. 1343–1350 (2013)
16. Nguyen, H.D., Sakama, C., Sato, T., Inoue, K.: Computing logic programming semantics in linear algebra. In: Proceedings of the 12th International Conference on Multi-disciplinary Trends in Artificial Intelligence (MIWAI 2018), Lecture Notes in Artificial Intelligence, vol. 11248, pp. 32–48. Springer, Heidelberg (2018)
17. Nguyen, H.D., Sakama, C., Sato, T., Inoue, K.: An efficient reasoning method on logic programming using partial evaluation in vector spaces. J. Log. Comput. **31**(5), 1298–1316 (2021)
18. Nguyen, T.Q., Inoue, K., Sakama, C.: Enhancing linear algebraic computation of logic programs using sparse representation. In: EPTCS Online Proceedings of ICLP (2020), vol. 325, pp. 192–205 (2020)
19. Nguyen, T.Q., Inoue, K., Sakama, C.: Linear algebraic computation of propositional horn abduction. In: 2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI) (2021)
20. Rocktäschel, T., Bosnjak, M., Singh, S., Riedel, S.: Low-dimensional embeddings of logic. In: Proceedings of the ACL 2014 Workshop on Semantic Parsing, pp. 45–49 (2014)
21. Sakama, C., Inoue, K., Sato, T.: Linear algebraic characterization of logic programs. In: Proceedings of the 10th International Conference on Knowledge Science, Engineering and Management (KSEM 2017), Lecture Notes in Artificial Intelligence, vol. 10412, pp. 520–533. Springer, Heidelberg (2017)
22. Sakama, C., Inoue, K., Sato, T.: Logic programming in tensor spaces. Ann. Math. Artif. Intell. (2021). https://doi.org/10.1007/s10472-021-09767-x
23. Sato, T.: Embedding Tarskian semantics in vector spaces. In: AAAI-17 Workshop on Symbolic Inference and Optimization (2017)
24. Sato, T.: A linear algebraic approach to datalog evaluation. Theory Pract. Logic Program. **17**(3), 244–265 (2017)
25. Sato, T., Inoue, K., Sakama, C.: Abducing relations in continuous spaces. In: Proceedings of IJCAI-18, pp. 1956–1962 (2018)
26. Serafini, L., Garcez, A.D.: Logic tensor networks: deep learning and logical reasoning from data and knowledge (2016). arXiv:1606.04422
27. Shakerin, F., Gupta, G.: White-box induction from SVM models: explainable AI with logic programming. Theory Pract. Logic Progr. **20**(5), 656–670 (2020)
28. Van Emden, M.H., Kowalski, R.A.: The semantics of predicate logic as a programming language. J. ACM (JACM) **23**(4), 733–742 (1976)
29. Yang, B., Yih, W., He, X., Gao, J., Deng, L.: Embedding entities and relations for learning and inference in knowledge bases. In: 3rd International Conference on Learning Representations, ICLR 2015, Conference Track Proceedings (2015)