On Linear Algebraic Computation for Logic Programming

by

Nguyen Quoc Tuan

Dissertation

submitted to the Department of Informatics

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

The Graduate University for Advanced Studies, SOKENDAI

September, 2022

Acknowledgments

First and foremost, I am extremely grateful to my supervisor, Prof. Katsumi Inoue, for his invaluable advice, continuous support, and infinite patience. His knowledge, experience and guidance have enlightened me through all the toughest times in the beginning of my research career.

I would like to extend my thank to Prof. Chiaki Sakama for his contribution, motivation, and immense knowledge. I have been lucky to be under his guidance that without his persistent help, the goal of this dissertation would not have been completed.

Besides, I wish to acknowledge the rest of my Ph.D. committee: Prof. Taisuke Sato, Prof. Takeaki Uno, Prof. Mahito Sugiyama for their encouragement, and insightful comments.

I am also grateful to be a part of the Inoue lab and extend my thank to all the lab members for supporting me, for inspiring me and for all unforgettable memories together.

On the journey to Ph.D, I have had to go through my greatest hardship ever. I could not make it this far without the continuous support and countless encouragement from so many people that I can name only a few. Hence, I would like to express my sincere to JICA staff Iijima Satoshi-san; JICA coordintor Miho Ishiura-san; medical specialists Dr. Yasuhiro Suyama; and NII Japanese teachers Kazumi Tajima-sensei and Naoko Adachi-sensei.

Last but not least, I am grateful to my family and my friends for always being by my side. This accomplishment would have not been possible without their constant source of love and endless encouragement.

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. The merit of logic reasoning in vector space is not only the scalability but also the capability of integrating with other Artificial Intelligence (AI) techniques such as Artificial Neural Network (ANN). Bridging Logic Programming (LP) and ANN can open up the gate to build more robust and explainable AI models. However, current work is usually based on manipulating the dense matrix format that is not efficient in both memory and time complexity. In addition, most of the work suffers from the combinatorial explosion problem that researchers have not yet found an appropriate approach to deal with in the language of algebra.

Following this direction, we have analyzed the sparsity of matrix representation of logic programs. Then we propose the use of general-purpose sparse representations to utilize the efficiency of linear algebraic approaches for deductive reasoning. 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 negations.

In this thesis, we also extend the linear algebraic characterization in abductive reasoning by exploiting the transpose of the program matrix. Then we propose a novel algorithm, which combines the flexibility and robustness of numerical computation with the compactness and efficiency of set operations, in order to compute solutions of abductive Horn propositional tasks. Experimental results demonstrate that our method is competitive with conflict-driven techniques and has the potential to speed up on parallel computing platforms.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Logic Programming (LP), which provides languages for declarative problem solving and symbolic reasoning [57], has started gaining more attention in terms of building explainable learning models [89, 18, 29]. For decades, LP representation has been considered mainly in the form of symbolic logic [53], 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 [85]. Recently, LP starts gaining more attention in order to build explainable learning models [16, 55, 8, 26], whereas it still has some limitations in terms of computation and efficiency. On the other hand, several researchers attempt to translate logical inference into numerical computation in the context of mixed integer programming [9, 34, 39, 56]. They exploit connections between logical inference and mathematical computation that open a new way for efficient implementation.

Lately, several studies have been done on embedding logic programs to numerical spaces and exploiting algebraic characteristics [75, 80, 7]. There are several reasons for considering linear algebraic computation of LP. First, linear algebra is at the heart of many applications of scientific computation, and integrating linear algebraic computation and symbolic computation is considered a challenging topic in Artificial Intelligence (AI) [78]. 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 [29]. Second, linear algebraic computation has the potential to cope with Web-scale symbolic data, and several studies develop scalable techniques to process huge relational knowledge bases [66, 73, 95] in tensor spaces. 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 we can exploit great com-

puting 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 [19].

In general, reasoning is the process of using existing knowledge to draw conclusions, construct explanations, or make predictions. The three methods of reasoning are the deductive, abductive, and inductive approaches.

1. *Deductive reasoning* (consequence): from prior knowledge (premises) to conclusions.

$$\frac{P \Rightarrow Q \quad P}{Q}$$

2. *Abductive reasoning* (explanation): from given effects to possible causes.

$$\frac{P \Rightarrow Q \quad Q}{P}$$

3. *Inductive reasoning* (generalization): from observed samples to causal rules.

$$\frac{P \quad Q}{P \Rightarrow Q}$$

In this thesis, we consider the linear algebraic computation in all three forms of reasoning. We have developed the methods for deduction and abduction while the method for induction is remaining as a future research work.

Deductive reasoning

In [14], 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 [79], Sato presented the use of first-order logic in vector spaces for Tarskian semantics, which demonstrates how tensorization realizes efficient computation of Datalog. In [80], 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 in this direction, Sato,

Inoue, and Sakama developed linear algebraic abduction to abductive inference in Datalog [83]. They did empirical experiments on linear and recursive cases and indicated that the approach can successfully abduce base relations.

In [38], 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 [25] 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 Artificial Neural Network (ANN) [88] 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 [75, 77]. 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 [61] and gain impressive performance. Sakama et al. has also proposed a partial evaluation method that precomputes the fixpoint based on matrix products of the program matrix with itself [76].

In [6], 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 [4]. 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 thesis, we have extended Sakama et al.'s idea of representing logic programs by tensors [75] to analyze the sparsity of program matrices and employ sparse representation for further enhancement.

Abductive reasoning

Abduction is a form of explanatory reasoning that has been not only discussed in Philos-

ophy of Science but used in AI for several tasks [68] including diagnosis and perception [48], automated planning [24], belief revision [11] and nonmonotonic reasoning [44]. In a nutshell, logic-based abduction is formulated as the search for a set of abducible propositions that together with a background theory entails the observations while preserving the consistency [23]. Abduction has often been used in the framework of LP, which is referred to as Abductive Logic Programming [50]. More recently, abductive reasoning has gained interests in connecting neural and symbolic reasoning [17] as well as explainable AI [42].

Abductive reasoning has been studied intensively in diagnosis and automated reasoning, and several procedures have been proposed in the literature. Inoue and Sakama has introduced the idea of disjunctive abduction and has presented several properties of disjunctive explanations in the context of (abductive) logic programming [47]. In the context of consistency-based diagnosis, the Assumption-based Truth Maintenance System (ATMS) has been used extensively [20]. Based on the background theory, ATMS constructs a directed graph in which propositions are represented as nodes, and in each node, ATMS stores all hypotheses allowing to infer this node. Further in [21], de Kleer developes an algorithm that ensures soundness, completeness, minimality, and consistency of every node label. In [70], Reiter has developed an approach via conflicts arising from the manifestation. Reiter exploits the hitting set relation between conflicts and consistency-based diagnoses to operate on a tree structure. In [35], Greiner et al. have extended Reiter's idea by utilizing a directed acyclic graph instead of a tree, then they have proposed Hitting Set Directed Acyclic Graph (HS-DAG).

In automated reasoning, Inoue proposed abduction as the search for logical consequences, in which explanations are derived deductively, via Skipping Ordered Linear (SOL) resolution [43]. SOL resolution has also been applied to compute nonmonotonic reasoning [44] and Meta-level abduction [46]. SOLAR (SOL for Advanced Reasoning) is the state-of-the-art implementation of SOL resolution based on the tableaux method [60].

In terms of linear algebraic computation, Sato et al. developed an approximate computation to abduce relations in Datalog [83], which is a new form of predicate invention in Inductive Logic Programming [59]. They did empirical experiments on linear and recursive cases and indicated that the approach can successfully abduce base relations, but their method cannot compute explanations consisting of possible abducibles in diagnosis.

In [5], Aspis et al. have proposed a linear algebraic transformation for abduction by exploiting Sakama et al.'s algebraic transformation. They have defined an explanatory operator based on third-order tensors for computing abduction in propositional Horn programs, which simulates deduction through Clark completion for abductive programs

[15]. Yet, the dimension explosion would arise unfortunately, and Aspis et al. have not yet reported an empirical evaluation.

With the goal to explore the potentials of linear algebraic computation for the Propositional Horn Clause Abduction Problem (PHCAP) in vector spaces, we continue to research the use of matrix representation of logic programs for solving abduction. We propose the use of the *transpose* of a program matrix that has been defined for deduction in [75, 63] to represent an *abductive matrix* for 1-step abduction in vector spaces, then solve the Minimal Hitting Sets (MHS) problem to deal with a number of alternative explanations in an efficient way, and finally we employ the sparse representation of abductive matrices for efficient computation.

Inductive reasoning
LP can be employed as a uniform representation for examples, background knowledge, and hypotheses, therefore, in Inductive Logic Programming (ILP) the goal is to derive a hypothesized logic program that entails all the positive and none of the negative examples. This problem also can be referred as a rule learning task. The rules extracted may represent a full scientific model of the data, or merely represent local patterns in the data. The rule learning process is performed on three levels [27]: Feature construction (1): In this phase, the object descriptions in the training data are turned into sets of features. Rule construction (2): Once the feature set is fixed, individual rules can be constructed, each covering a part of the example space. Hypothesis construction (3): A hypothesis consists of a set of rules. In propositional rule learning, hypothesis construction can be simplified by learning individual rules sequentially.

To the best of my knowledge, employing matrix representation requires a lot of effort in both theory and practice. The merit of learning rules in vector space is not only the scalability but also the capability of integrating with other AI techniques *e.g.* ANN. Bridging LP and ANN can open up the gate to build more robust and explainable AI models.

Structure of the Dissertation

In this section, we provide the layout of the remaining chapters and a summary of each:

- Chapter 2 provides background information necessary to understand the remainder of the dissertation. We go through a quick summary of the basic of LP, stable model computation and abduction problem.

- Chapter 3 presents in detail the linear algebraic approach for computing the least model of definite and normal programs. We also provides experimental results

to demonstrate the efficiency and scalability of the method.

- Chapter 4 discusses about the PHCAP and how to deal with this problem using the linear algebraic approach. The experimental results in this chapter validates the method with a competitive performance with traditional approaches while encourages further improvement in the future.
- Chapter 5 discusses in a broader view of linear algebraic approaches and similar methods in all three forms of logic reasoning: deduction, abduction and induction.
- Chapter 6 summarizes all the contributions of this thesis and also discusses about potential future research directions.

Chapter 2

Background

In this chapter, we will go through the necessary background knowledge and the basic notations we use in this thesis. The content of this chapter includes first-order logic, definite and normal logic programs, and abduction.

2.1 First-Order Logic

First-order logic - also known as predicate logic, quantificational logic, and first-order predicate calculus - is a collection of formal systems used in mathematics, philosophy, linguistics, and computer science. First-order logic uses quantified variables over non-logical objects, and allows the use of sentences that contain variables.

2.1.1 Syntax

The syntax of first-order logic defines what constitutes a well-formed formula.

Definition 1. First-Order Alphabet [57]: An alphabet of first-order logic comprises seven sets of symbols

1. *constants*
2. *variables*
3. *function* symbols
4. *predicate* symbols
5. *connectives*: $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftarrow$ and $\leftrightarrow$
6. *quantifiers*: $\exists$ (the existential quantifier) and $\forall$ (the universal quantifier)

7. Three punctuation symbols: '()', ')' and ','.

Definition 2. First-Order Term [57]: A *term* is defined as follows:

1. A *constant* is a *term*.
2. A *variable* is a *term*.
3. If f is an n-ary function symbol and t_1, ..., t_n are *terms*, then $f(t_1, \ldots, t_n)$ is a *term*.

A ground term is a term which does not contain any variables.

Definition 3. First-Order Formula [57]: A (well-formed) formula is defined as follows:

1. If p is an n-ary predicate symbol and t_1, ..., t_n are terms, then $p(t_1, ..., t_n)$ is a formula, called an atom. A ground atom is an atom which does not contain any variables.
2. If φ and ψ are formulas, then $\neg\varphi$, $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\varphi \rightarrow \psi)$, $(\varphi \leftarrow \psi)$, and $(\varphi \leftrightarrow \psi)$ are formulas.
3. If φ is a formula, and X is a variable, then $\exists X\ \varphi$ and $\forall X\ \varphi$ are formulas.

An infinite number of formulas can be constructed from a first-order alphabet. The set of such formulas is called a language.

Definition 4. First-Order Language [57]: The set of all formulas which can be constructed from the symbols of an alphabet is a first-order language.

The effect of a quantifier applies only to those occurrences of a variable in the region of the formula that is within the scope of the quantifier, as defined below. A variable that is within the scope of a quantifier is bound and a variable that is not within the scope of any quantifier is free.

Definition 5. Quantifier Scope [57]: The scope of $\forall X$ in $\forall X\ \varphi$ is φ. An occurrence of a variable immediately following a quantifier, or within the scope of a quantifier immediately followed by the same variable, is bound. Any other occurrence of a variable is free.

Definition 6. Closed Formula [57]: A closed formula contains no free occurrences of variables.

2.1.2 Semantics

The semantics of a system of logic determines the meaning of a formula. A formula is either $true$ or $false$, depending on the formula and whether the atoms in the formula have been given the value **T** (true) or **F** (false) according to some interpretation.

In first-order logic a term refers to an object in some domain. Thus to assign values to atoms in a language $\mathscr{L}$, it is first necessary to determine to which object in the domain each term refers. This is the purpose of a pre-interpretation of $\mathscr{L}$, and a variable assignment with respect to $\mathscr{L}$.

Definition 7. Pre-Interpretation [57]: Let $\mathscr{L}$ be a first-order language. A pre-interpretation of $\mathscr{L}$ consists of:

1. A set D, called the domain of the pre-interpretation.
2. The assignment of an element in D to each constant in the alphabet of $\mathscr{L}$.
3. For each n-ary function in the alphabet of $\mathscr{L}$, the assignment of a mapping from D^n to D, where D^n is the set of all n-tuples of elements in D.

Definition 8. Variable Assignment [57]: Let $\mathscr{L}$ be a first-order language, and let J be a pre-interpretation of $\mathscr{L}$. A variable assignment of $\mathscr{L}$ with respect to J is an assignment of an element in the domain of J to each variable in the alphabet of $\mathscr{L}$. The expression $V(X/d)$ is used to denote a variable assignment that maps variable X to domain element d and maps other variables according to V.

Definition 9. Interpretation [57]: Let $\mathscr{L}$ be a first-order language. An interpretation I of $\mathscr{L}$ consists of:

1. (i) A pre-interpretation J, with domain D, of $\mathscr{L}$.
2. (ii) For each n-ary predicate symbol p in the alphabet of $\mathscr{L}$, the assignment of a mapping I_p from D^n to $\{T, F\}$.

If, for all predicates p, the mapping I_p is a total mapping, then I is a total interpretation, otherwise I is a partial interpretation.

Unless otherwise stated, all interpretations are assumed to be total interpretations.

Definition 10. Truth Value of a Formula [57]: Let $\mathscr{L}$ be a first-order language. Let I be an interpretation of $\mathscr{L}$ based on pre-interpretation J with domain D, and let V be a variable assignment of $\mathscr{L}$ with respect to J. The truth value under I and V of a formula φ in $\mathscr{L}$ is:

- If φ is the atom $p(t_1, ..., t_n)$, and the domain elements assigned to t_1, ..., t_n by I and V are d_1, ..., d_n, then the truth value of φ is the value assigned to d_1, ..., d_n by I_p.
- If φ is of the form:
  - (a) $\neg\psi$, then the value of φ is **T** if and only if the value of ψ is **F** ;
  - (b) $(\psi \wedge \mathcal{X})$, then the value of φ is **T** if and only if the value of ψ is **T** and the value of $\mathcal{X}$ is **T** ;
  - (c) $(\psi \vee \mathcal{X})$, then the value of φ is **T** if and only if either the value of ψ is **T** or the value of $\mathcal{X}$ is **T** ;
  - (d) $(\psi \rightarrow \mathcal{X})$, then the value of φ is **F** if and only if the value of ψ is **T** and the value of $\mathcal{X}$ is **F** ;
  - (e) $(\psi \leftrightarrow \mathcal{X})$, then the value of φ is **T** if and only if either the value of both ψ and $\mathcal{X}$ is **T** , or the value of both ψ and $\mathcal{X}$ is **F**
- 3. If φ is of the form $\exists X\ \psi$, then the truth value of φ is **T** if there exists at least one element $d \in D$ such that ψ has value **T** with respect to I and $V(X/d)$, otherwise it is **F**.
- 4. If φ is of the form $\forall X\ \psi$, then the truth value of φ is **T** if, for every element $d \in D$, ψ has value **T** with respect to I and $V(X/d)$, otherwise it is **F**.

The truth value of a closed formula is independent of the variable assignment. Since this thesis is only concerned with closed formulas, then only the interpretation will be referred to from this point on. Also, from this point the term 'formula' will be used to mean 'closed formula.

If the truth value of a formula under an interpretation is true, then that interpretation is called a model of the formula. The following definitions formalise the notion of a model, and use it to define the concept of a logical consequence of a set of formulas.

Definition 11. Model of a Formula [57]: If a (closed) formula φ has the value T under an interpretation I, then I is said to satisfy φ, or to make φ true. If I satisfies φ, then I is a model of φ, and φ has a model I.

Definition 12. Model of a Set of Formulas [57]: Let Σ be a set of formulas and I an interpretation. If I satisfies every formula in Σ, then I is a model of Σ.

Definition 13. Logical Consequence [57]: Let Σ be a set of formulas and φ be a formula. If every model of Σ is also a model of φ, then φ is a logical consequence of Σ, written $\Sigma \models \varphi$. If φ is a logical consequence of Σ, then Σ entails φ.

Definition 14. Entailment of a Set [57]: Let Σ and ψ be sets of formulas. If $\Sigma \models \varphi$, for every formula $\varphi \in \psi$, then ψ is a logical consequence of Σ, written $\Sigma \models \psi$. If ψ is a logical consequence of Σ, then Σ entails ψ.

Definition 15. Semantic Terminology [57]: Let φ be a formula

- If ψ is a formula, and both $\varphi \models \psi$ and $\psi \models \varphi$, then φ and ψ are logically equivalent, written $\varphi \leftrightarrow \psi$.
- If there is some interpretation that is a model of φ, then φ is satisfiable.
- If every interpretation is a model of φ, then φ is a tautology.
- If there is no model of φ, then φ is a contradiction. Then φ is also called unsatisfiable, or inconsistent.

2.2 Definite Programs

Definition 16. Definite Clause [57]: A definite clause is a clause containing exactly one positive literal.

In the logic programming notation as presented above, a definite clause is of the form:

$$A \leftarrow B_1, ..., B_k$$

where A, B_1, ..., B_k are atoms. A is the head of the clause, and B_1, ..., B_k is the body of the clause. A definite clause $A \leftarrow$ is called a fact.

Definition 17. Definite Program [57]: A definite program Π is a finite set of definite clauses. The set ground(Π) is the set of all possible ground instances of the clauses in Π.

Definition 18. Definition of a Predicate [57]: Let p be a predicate symbol. The definition of p in a definite program Π is the set Π_p of all clauses in Π with p in the head.Then p is defined by Π_p.

Definition 19. Definite Goal [57]: A definite goal is a clause of the form

$$\leftarrow B_1, ..., B_k$$

with an empty consequent. Each atom B_i $(1 \leq i \leq n)$ is a subgoal of this goal.

Definition 20. Horn clause [57]: A Horn clause is a clause that is either a definite clause or a definite goal.

Proposition 1. Existence of Least Herbrand Model [57]: Let Π be a definite program. If $\{M_1, ...M_k, ...\}$ is a (possibly infinite) set of Herbrand models of Π, then their intersection $M = \cap_i M_i$ is a Herbrand model of Π.

Proof. Suppose M is not a Herbrand model of Π. Clearly M is a Herbrand interpretation, therefore it is not a model of Π. In this case, there is a clause $C \in \Pi$ such that some instance of C is not satisfied by M. Let this instance be $C\theta = A \leftarrow B_1, ..., B_n$. Then $B_j \in M$ for every j $(1 \leq j \leq n)$, and $A \notin M$. And since $M = \cap_i M_i$, then $B_j \in M_i$ for every j $(1 \leq j \leq n)$ and every $i \geq 1$. Furthermore, since every M_i, $i \geq 1$, is a model of Π, then $A \in M_i$ for all $i \geq 1$. But then $A \in M$, which is a contradiction. □

Definition 21. Least Herbrand Model [57]: Let Π be a definite program. The least Herbrand model of Π, denoted M_Π, is the intersection of all Herbrand models of Π.

By Proposition 1 M_Π is a model of Π. What is more, exactly those atoms that are logical consequences of Π are true in M_Π. The proof of the following theorem is given in [67].

Theorem 1. Herbrand Model and Implication of Atoms [57]: Let Π be a definite program. Then $A \in M_\Pi$ if and only if $\Pi \models A$.

Proof. Let A be an atom in B_Π.
$\Pi \models A$ if and only if
$\Pi \cup \{\neg A\}$ is unsatisfiable if and only if
$\Pi \cup \{\neg A\}$ has no models if and only if
$\Pi \cup \{\neg A\}$ has no Herbrand models if and only if
A is true in all Herbrand models of Π if and only if
$A \in M$. □

2.3 Normal Logic Program

Negative information cannot be inferred from a definite program. In order to add this feature to the logic program methods described so far it is necessary to introduce the closed world assumption, which gives rise to the negation as failure rule and normal logic programs.

Definition 22. Program Clause [57]: A program clause is a formula of the form

$$A \leftarrow L_1, ..., L_n$$

where A is an atom and L_1, ..., L_n are literals.

Definition 23. Normal Goal [57]: A normal goal is a formula of the form

$$\leftarrow L_1, ..., L_n$$

where L_1, ..., L_n are literals.

Definition 24. Level Mapping [57]: A level mapping of a normal program is a mapping from its set of predicate symbols to the non-negative integers. The value of a predicate under this mapping is the level of the predicate.

Definition 25. Hierarchical Program [57]: A normal program Π is hierarchical if there is a level mapping of Π such that, for every clause $p(t_1, ..., t_n) \leftarrow L_1, ..., L_m$ in Π, the level of every predicate in L_1, ..., L_m is less than the level of p.

Definition 26. Stratified Program [57]: A normal program Π is stratified if there is a level mapping of Π such that, for every clause $p(t_1, ..., t_n) \leftarrow L_1, ..., L_m$ in Π, the level of the predicate symbol of every positive literal in $L_1, ..., L_m$ is less than or equal to the level of p, and the level of the predicate symbol of every negative literal in L_1, ..., L_m is less than the level of p.

Theorem 2. Stratified Program [3]: Let Π be a stratified normal program. Then *comp*(Π) has a minimal normal Herbrand model.

Definition 27. Allowed Goal [57]: Let Π be a normal program and G be a normal goal. A clause $A \leftarrow L_1, ..., L_n$ in Π is admissible if every variable that occurs in the clause occurs either in A or in a positive literal in L_1, ..., L_n.
A clause $A \leftarrow L_1, ..., L_n$ in Π is allowed if every variable that occurs in the clause occurs in a positive literal in L_1, ..., L_n.
G is allowed if G is $\leftarrow L_1, ..., L_n$ and every variable that occurs in G occurs in a positive literal in $L_1, ..., L_n$.
$\Pi \cup \{G\}$ is allowed if the following conditions are satisfied:

1. (i) Every clause in Π is admissible.

2. (ii) For every clause C in the definition of a predicate symbol p, where p occurs in a positive literal in the body of G or in a positive literal in the body of a clause in Π, C is allowed.

3. (iii) G is allowed.

Several alternative declarative semantics have been proposed for logic programs. Among them the stable model semantics, introduced by Gelfond and Lifschitz in [33], is the most well known and successful.

Definition 28. Reduct of Program [33]: Let Π be a normal program and let $M \subseteq U_\Pi$ be a set of atoms. The reduct of Π with respect to M is the ground definite program Π^M, obtained from the set of all ground instances of clauses in Π by deleting

- (i) each clause containing a negative literal $\neg A$ in its body where $A \in M$, and
- (ii) all negative literals in the bodies of the remaining clauses.

Definition 29. Stable Model [33]: Let Π be a normal program and let $M \subseteq U_\Pi$ be a set of atoms. Then M is a stable model of Π if and only if M is the least Herbrand model of Π^M.

In general, a normal program can have zero, one, or many stable models.

2.4 Abduction

This section describes abduction. Abduction is known as one of three forms of logical reasoning: deduction, abduction and induction. While deduction produces conclusions which follow from some prior knowledge by logical implication, abduction produces explanations of observations that are not implied by existing knowledge.

Abduction is implemented in logic programming through an abductive, or open logic program. The term abductive program is the more common one. However, the term open program, proposed by Denecker and De Schreye [22], is adopted here since it generalises the notion to induction as well as abduction. An open program has a fully defined part, and an incomplete part.

Definition 30. Open Program [22]: An open program is a triple $\langle \Pi, U, I \rangle$, where Π is a program, U is a set of predicates called undefined or abducible, and I is a set of first-order axioms. If Π is a definite program and I is a set of definite goals, then $\mathbf{P}$ is a definite open program. A ground literal with predicate $p \in U$ is called an abducible literal.

The axioms I are also known as integrity constraints and serve to constrain which facts may be abduced. Unless otherwise stated, the language of an open program $\mathbf{P} = \langle \Pi, U, I \rangle$ is assumed to be given by the symbols appearing in Π and U and I.

The semantics of an open program $\mathbf{P}$ is given by the consequences of its completion, $comp(P)$. Since the predicates in U are not yet fully defined, the closed world assumption does not apply to them. Thus, the completion of an open program is formed by completing only the definitions of those predicates in U, the complement of U.

Definition 31. Open Program Completion [22]: Let $\mathscr{L}$ be a first-order language, let $Pred$ be the set of predicates in $\mathscr{L}$ and let $P = \langle \Pi, U, I \rangle$ be an open program in $\mathscr{L}$. The completion, $comp(P)$, of P is

$$EQ \;\; U\{p(t_1, ..., t_n) \leftarrow \varphi \in \Pi | p \in U\}$$
$$U\{compdef(q, \Pi) | q \in Pred \wedge q \in U\}.$$

This is the semantics presented by Denecker and De Schreye [22], which generalises that of Console et al. [15] by allowing predicates in U to be partially defined. Nevertheless, in practical abductive logic programming it is often simpler if none of the predicates in U are defined in Π. For this purpose, an open program can be transformed into a disjoint open program by introducing auxiliary predicates in such a way that the resulting program has no definitions for the abducible predicates [45].

Definition 32. Disjoint Open Program : Let $P = \langle \Pi, U, I \rangle$ be an open program. If Π does not contain a definition of any predicate in U, then P is a disjoint open program.

The task of abductive logic programming, then, is to answer queries by producing not only an answer substitution, as a deductive proof procedure does, but also a set of abduced facts. The set of abduced facts Δ is called an explanation or hypothesis.

Definition 33. Abductive Hypothesis : Let $P = \langle \Pi, U, I \rangle$ be an open program and G be a normal goal. An answer for G given P is a substitution θ for the variables in G, and a set Δ of abduced ground atoms whose predicates are in U. The set Δ is an abductive hypothesis or explanation for $G\theta$.

In general, G can be any goal but in practice G will often be ground. In this case the substitution θ will be the identity substitution ϵ. In order to be correct, a hypothesis must extend P to produce a program that implies the observations and is consistent with the integrity constraints.

Definition 34. Abductive Solution : Let $P = \langle \Pi, U, I \rangle$ be an open program, let G be a normal goal $\leftarrow L_1, ..., L_n$, and let $\langle \theta, \Delta \rangle$ be an answer for G given P . Then $\langle \theta, \Delta \rangle$ is a correct answer, or abductive solution for G given P if $\forall((L_1 \wedge ... \wedge L_n)\theta)$ is a logical consequence of $comp(\Pi \cup \Delta)$, and $comp(\Pi \cup \Delta) \cup I$ is consistent.

Definition 35. Minimal Abductive Solution : Let P be an open program, let G be a normal goal and let $\langle \theta, \Delta \rangle$ be an abductive solution for G given P . Then $\langle \theta, \Delta \rangle$ is a minimal abductive solution for G given P if there is no abductive solution $\langle \theta, \Delta' \rangle$ for G given $\langle \theta, \Delta \rangle$ such that $\Delta' \subset \Delta$.

Chapter 3

Linear Algebraic Computation in Deductive Reasoning

The linear algebraic characteristics of Logic Programming (LP) were first analyzed by Sakama, Inoue, and Sato [75]. Then after many researchers have developed variants based on the backbone of linear algebraic computation. However, almost all previous methods have not considered the sparse characteristic of program matrices. Thus, these algorithms suffer from high complexity of the general matrix multiplication operator due to using the dense format.

In this chapter, we will first investigate matrix representation of logic programs and then analyze their sparsity. Then next we discuss about different sparse representation and their features in terms of representing logic programs. Further, we do complexity analysis of the methods for definite programs and normal programs and prove a remarkable improvement in both time and space complexity. Later in the Section 3.4, we demonstrate our performance in a various experiments.

3.1 Linear Algebraic Computation of Logic Programs

3.1.1 Definite programs

We consider a language $\mathscr{L}$ that contains a finite set of propositional variables.

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

$$h \leftarrow b_1 \wedge \cdots \wedge b_m \quad (m \geq 0) \tag{3.1}$$

where h and b_i are propositional variables in $\mathscr{L}$. In (3.1) the left-hand side of $\leftarrow$ is called

the *head* and the right-hand side is called the *body*. We call a rule of the form (3.1) is an *And*-rule. A Horn logic program P is called *singly defined* if $h_1 \neq h_2$ for any two different rules $h_1 \leftarrow B_1$ and $h_2 \leftarrow B_2$ in P where B_1 and B_2 are conjunctions of atoms. That is, no two rules have the same head in an SD program. When P contains more than one rule $(h \leftarrow B_1), \ldots, (h \leftarrow B_n)$ $(n > 1)$, replace them with a set of new rules:

$$\begin{aligned} &h \leftarrow b_1 \vee \cdots \vee b_n \\ &b_1 \leftarrow B_1 \quad \cdots \quad b_n \leftarrow B_n \end{aligned} \tag{3.2}$$

where $b_1, \ldots, b_n$ are new atoms such that $b_i \notin B_P$ $(1 \leq i \leq n)$ and $b_i \neq b_j$ if $i \neq j$. Every Horn logic program P is transformed to $\Pi = Q \cup D$ such that Q is an SD program and D is a set of rules of the form (3.2). The resulting Π is called a *standardized program*. Note that the rule (3.2) is a shorthand of n rules: $h \leftarrow b_1, \ldots, h \leftarrow b_n$, so a standardized program is considered a Horn logic program. In this paper, we refer to the rule of the form (3.2) as an *Or*-rule.

Throughout the thesis, a program means a standardized program unless stated otherwise. Additionally, we refer abduction problem to Propositional Horn Clause Abduction Problem (PHCAP) that we consider logic program in Horn clauses. For each rule r of the form (3.1) or (3.2), define $head(r) = h$ and $body(r) = \{b_1, \ldots, b_m\}$ (or $body(r) = \{b_1, \ldots, b_n\}$). A rule is called a *fact* if $body(r) = \emptyset$. A rule is called a *constraint* if $head(r) = \emptyset$. A constraint $\leftarrow\ b_1 \wedge \cdots \wedge b_m$ is replaced with

$$\bot \leftarrow\ b_1 \wedge \cdots \wedge b_m$$

where $\bot$ is a symbol representing **False**. When there are multiple constraints, say $(\bot \leftarrow B_1), \ldots, (\bot \leftarrow B_n)$, they are transformed to

$$\bot \leftarrow \bot_1 \vee \cdots \vee \bot_n \quad \text{and} \quad \bot_i \leftarrow B_i \ \ (i = 1, \ldots, n)$$

where $\bot_i \notin B_P$ is a new symbol. Given a program P, the set of all propositional variables appearing in P is the *Herbrand base* of P (written B_P). An *interpretation* $I\ (\subseteq B_P)$ is a *model* of a program P if $\{b_1, \ldots, b_m\} \subseteq I$ implies $h \in I$ for every rule (3.1) in P, and $\{b_1, \ldots, b_n\} \cap I \neq \emptyset$ implies $h \in I$ for every rule (3.2) in P. A model I is the *least model* of P (written LM_P) if $I \subseteq J$ for any model J of P. We write $P \models a$ when $a \in LM_p$. For a set $S = \{a_1, \ldots, a_n\}$ of atoms, we write $P \models S$ if $P \models a_1 \wedge \cdots \wedge a_n$. A program P is *consistent* if $P \not\models \bot$.

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 (3.1) in P, and $\{b_1, \ldots, b_m\} \cap I \neq \emptyset$ implies $h \in I$ for every rule (3.2) in P. A model I is the

least model of P if $I \subseteq J$ for any model J of P. A mapping $T_P : 2^{B_P} \to 2^{B_P}$ (called a T_P-*operator*) is defined as: $T_P(I) = \{\, h \mid h \leftarrow b_1 \wedge \cdots \wedge b_m \in P \text{ and } \{b_1, \ldots, b_m\} \subseteq I \,\} \cup \{\, h \mid h \leftarrow b_1 \vee \cdots \vee b_n \in P \text{ and } \{b_1, \ldots, b_n\} \cap I \neq \emptyset \,\}$.

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 [93].

The idea of deduction computation is based on how do we encode logic programs into matrices and how do we interpret numerical values into models of logic programs. At a general level, it is easier to imagine a set of propositional variables as a vector in which the value of each element in the vector indicates the existence of a corresponding variable. We can say if the value in the vector is 0 then the corresponding variable is excluded while if the value is larger than 0, the propositional variable is included. The more important part is how do we manipulate these variables using matrix operators to realize logic inferencing in vector spaces.

Let us consider a very simple program with only a single *And*-rule $p \leftarrow q \wedge r$. Then the set of all propositional variables is $\{p,\ q,\ r\}$ can be represented by a vector v of 3 elements that each element 0, 1, 2 indicates the existence of each variable $p,\ q,\ r$ respectively. We can interpret this vector by saying q is included if $v[1] > 0$ and so on. Now we also need to represent the logic program which is a set of rules into something in vector spaces. We can consider to use vector as mentioned for each rule, however we need a way to identify each rule in the program. Thus, a matrix, which is a set of multiple row vectors, is an ideal choice for this purpose. We can address each row is a rule in the program and set a mapping between each row index and each rule in the program. To simplify this task, we can introduce some conditions such as there is no two rules having the same head atom, so we can have a direct mapping between the indices of propositional variables and rules via the propositional variable in the rule head. Accordingly, we can have a sample matrix of the mentioned program as follows:

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

This program has only a single *And*-rule with the head atom is p so we only need to use the first row of the matrix. The reason for choosing the value $1/2$ is that each variable in the rule body $q,\ r$ contributes the same amound of information to deduce p. Next,

let us see the behavior of the above matrix if we multiply it with a vector (0, 1, 1).

$$\begin{array}{c} \\ p \\ q \\ r \end{array}\!\!\begin{array}{c} \begin{array}{ccc} p & q & r \end{array} \\ \begin{pmatrix} 0 & 1/2 & 1/2 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \end{array} \cdot \begin{array}{c} \\ p \\ q \\ r \end{array}\!\!\begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} = \begin{array}{c} \\ p \\ q \\ r \end{array}\!\!\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

Interestingly, the behavior is similar to apply inferencing on a set of two variables p and q. In fact, we can do more with logic in vector spaces that we are going to present more formally later in this section and the next section.

In order to deal with deductive reasoning, Sakama et al. has formally described the method in [75]. To extend this idea to work with abduction, we slightly modify the definition by Sakama et al. to define a matrix program of a logic program P in a vector space.

Definition 36. Matrix representation of standardized programs [75]: 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. $a_{ij_k} = \frac{1}{m}$ $(1 \leq k \leq m;\ 1 \leq i, j_k \leq n)$ 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 36, let's 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 Singly-Defined (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 36, 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 37). 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 38).

Definition 37. Interpretation vector [75]: 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 38. 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, $a_i = 0$.

In order 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 39. 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.

3.1.2 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.

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: **return** v

Definition 40. Normal program: A *normal program* is a finite set of normal rules:

$$h \leftarrow b_1 \wedge b_2 \wedge ... \wedge b_l \wedge \neg b_{l+1} \wedge ... \wedge \neg b_m \ (m \geq l \geq 0) \tag{3.3}$$

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

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

$$h \leftarrow b_1 \wedge b_2 \wedge ... \wedge b_l \wedge \overline{b}_{l+1} \wedge ... \wedge \overline{b}_m \ (m \geq l \geq 0) \tag{3.4}$$

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 $\bar{I} = \{\bar{p} \mid p \in B_P \setminus I\}$, 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 41. Matrix representation of normal programs [61]: 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 36.

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 $P^+ = \{p \leftarrow q \wedge s,\ q \leftarrow p \wedge t,\ s \leftarrow \overline{t},\ t \leftarrow,\ u \leftarrow v\}$. Then applying Definition 41, we obtain:

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

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 42. Initial matrix [61]: 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 41 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 of 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 49
5: **while** $U \neq M$ **do**
6: $\quad M = U$, $U = \theta(M_P M)$ ▷ refer to Definition 49
7: $V =$ find stable models of P ▷ refer to Algorithm 3
8: **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: $\quad v = (a_1, \ldots a_n, a_{n+1}, \ldots, a_m)^T$ (i^{th}-column of M)
4: $\quad$ **for** j from $n+1$ to m **do**
5: $\quad\quad \overline{q}_j = row_j(M)$
6: $\quad\quad$ **for** l from 1 to n **do**
7: $\quad\quad\quad$ **if** $row_l(M) = q_j$ **then**
8: $\quad\quad\quad\quad$ **if** $a_l + a_j \neq 1$ **then** break;
9: $\quad\quad$ **if** $l \leq n$ **then** break;
10: $\quad$ **if** $j \leq m$ **then** break;
11: $\quad$ **else** $V = V \cup \{v\}$
12: **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.

3.2 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 [61]. In order to solve this problem, this thesis 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.

3.2.1 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 [13]. Because there are a large number of zero elements in sparse matrices, we can save the computation by ignoring these zero values [37]. According to the conversion method of linear algebraic approach, we can calculate the sparsity of a program P [1]. 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:

$$sparsity(P) = 1 - \frac{\sum_{r \in P} |body(r)|}{n^2} \tag{3.5}$$

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.

[1] We only consider the programs in Definition 36 and Definition 41.

3.2.2 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 [13]. Among several sparse storage formats, we select the three formats Coordinate (COO), Compressed Sparse Row (CSR) and Block Compressed Sparse Row (BSR) [10] 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 Knowledge Base (KB)s.

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 [2] is used and we follow row-major order [3].

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 usings. 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.

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

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

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 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 [4] and compress the column index array. Hence, the space

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

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's consider the BSR format for the logic program P in 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

	p	q	r	s	t	u	v	$-$
p	0	0	0	0	0	1	1	0
q	0	0	0	0	1	0	0	0
r	0	0	0	1	0	0	0	0
s	0	0	0	1	0	0	0	0
t	0	0	0	0	1	0	0	0
u	0	1/2	1/2	0	0	0	0	0
v	0	0	0	1/2	1/2	0	0	0
$-$	0	0	0	0	0	0	0	0

eliminate zero blocks →

	p	q	r	s	t	u	v	$-$
p					0	1	1	0
q					1	0	0	0
r			0	1				
s			0	1				
t	0	0	0	0	1	0		
u	0	1/2	1/2	0	0	0		
v			0	1/2	1/2	0		
$-$			0	0	0	0		

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.

3.3 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.

3.3.1 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. [5]

Proposition 2. 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 3. The time complexity of linear algebraic method for definite programs is

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.

3.3.2 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. [6]

Proposition 4. The space complexity of the linear algebraic method for normal programs is

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

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

[6] 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.

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 5. 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 $O(n^2 \times (2^k - 1))$. □

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.

3.4 Experiments

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. In order 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 [32]. In terms of matrix representations and operators for (*iii*) and (*iv*), we use Eigen 3 library [36] with the default backend. The computer running experiments has the following configurations: CPU: Intel Core i7-4770 (4 cores, 8 threads) @3.4GHz; RAM: 16GB DDR3 @1333MHz; GPU: NVIDIA GTX 1080; Operating system: Ubuntu 18.04 LTS 64bit.

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 [61] 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 3.1.

Table 3.1: 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%

According to Algorithm 1 and Algorithm 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 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 [61] 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 [32] 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 [54]. This dataset contains binary tuples and we compute the transitive closure of them using the following rules:

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

3.4.1 Definite programs

The final results on definite programs are illustrated in Table 3.2 and Figure 3.1.

Table 3.2: Details of experimental results on definite programs of Hashset method, Clasp and linear algebraic methods (with dense and sparse representation).

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

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**

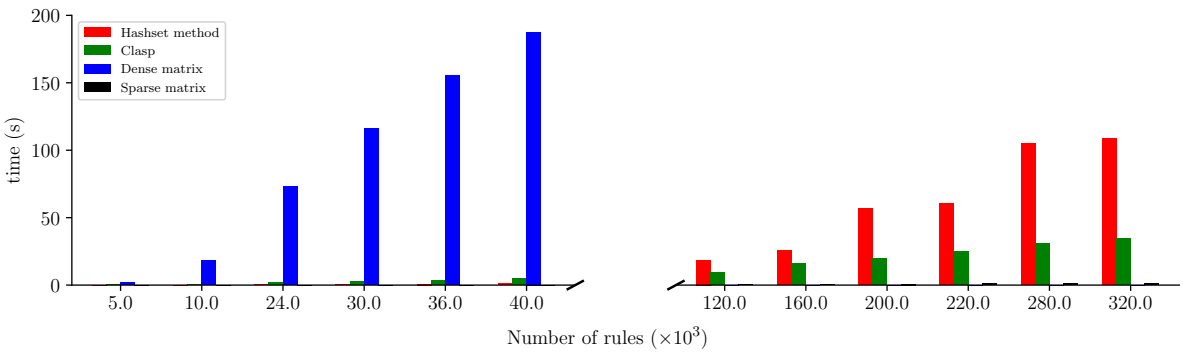


Figure 3.1: Comparison of execution time between Hashset method, Clasp and linear algebraic methods (with dense and sparse representation) on definite programs.

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 the number of rules m is used as horizontal axis in the Figure 3.1 similar to the experiments in [61]. The reason for choosing n and m is to generate actual matrix size n' increasing linearly with two different levels: smaller scale ($n < 10000$) and larger scale ($n > 10000$). The same parameters are used for other experiments using the 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 3.3 and Figure 3.2. 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.

Table 3.3: 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' is the actual matrix size after transformation. Time unit is second.

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**

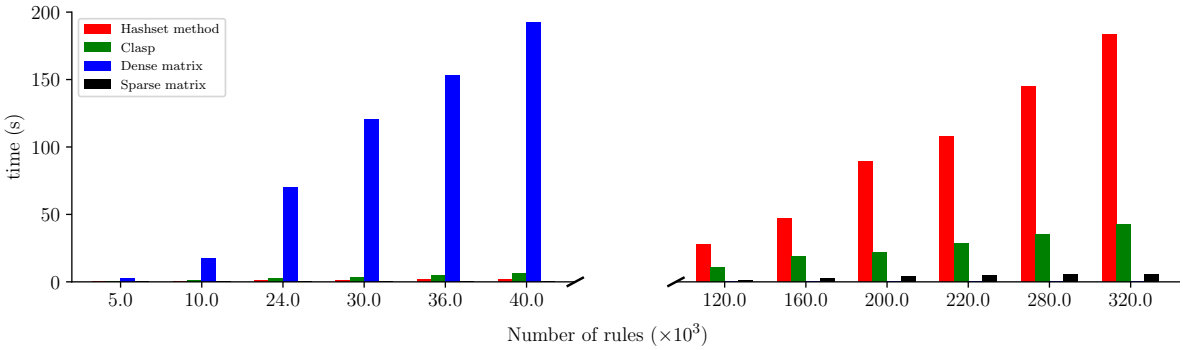


Figure 3.2: Comparison of execution time between Hashset method, Clasp and linear algebraic methods (with dense and sparse representation) on definite programs with lower sparsity level.

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 3.4 and Figure 3.3. 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 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 3.2–3.4. 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.

Table 3.4: Details of experimental results on the transitive closure problem of Hashset method, Clasp and sparse representation approach.

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

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**

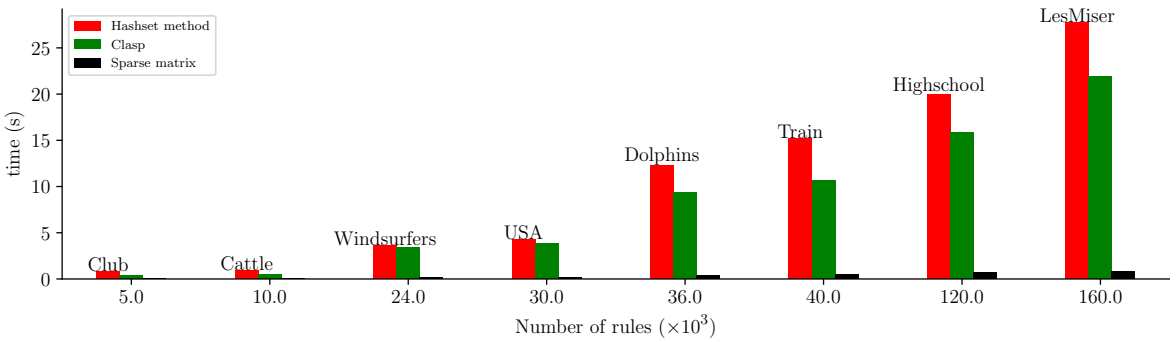


Figure 3.3: Comparison of execution time between Hashset method, Clasp and linear algebraic methods (with dense and sparse representation) on definite programs with Transitive closure problem using Koblenz network datasets.

3.4.2 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 42)

grows exponentially by the number of negations, we limit the number of negations in this benchmark by 8 [7] 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 3.5 and Figure 3.4 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 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 3.6 and Figure 3.5 present the data for this benchmark. Once again, with a limited number of negations, the sparse matrix method holds the winner position.

Table 3.5: Details of experimental results on normal programs of Hashset method, Clasp and linear algebraic methods (with dense and sparse representation).

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

Time unit is second.

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**

[7] 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).

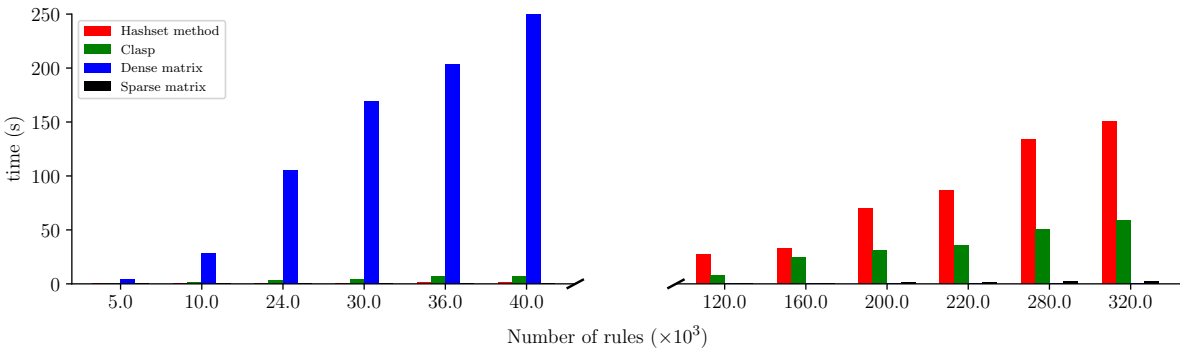


Figure 3.4: Comparison of execution time between Hashset method, Clasp and linear algebraic methods (with dense and sparse representation) on normal programs.

Table 3.6: 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' is the actual matrix size after transformation. k is the number of negations.

Time unit is second.

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**

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.

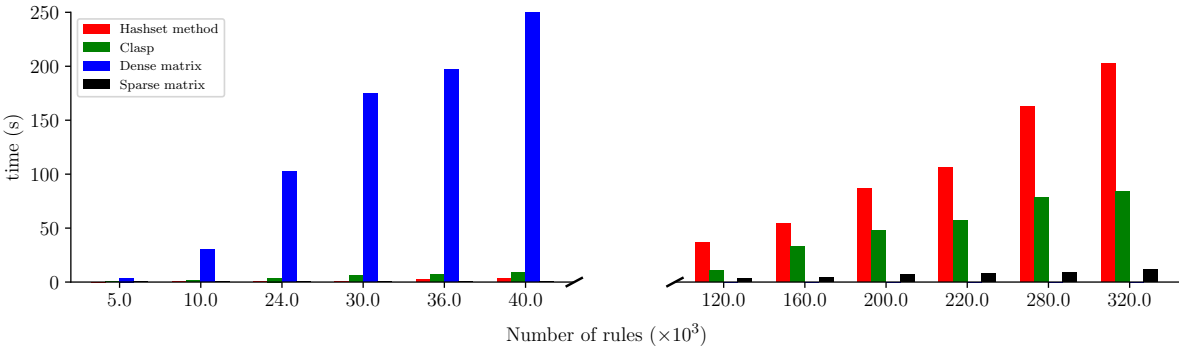


Figure 3.5: Comparison of execution time between Hashset method, Clasp and linear algebraic methods (with dense and sparse representation) on normal programs with lower sparsity level.

3.4.3 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. In order 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 Section 3.2 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.

Table 3.7: Comparison of different sparse representations in terms of the memory size on definite programs in Table 3.2.

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

n	m	n'	η_z	**COO**	**CSR/CSC**	**BSR** 2×2	**BSR** 4×4
1000	5000	5788	24,848	298,176	221,940	459,876	1,414,952
				(100.00%)	(74.43%)	(154.23%)	(474.54%)
1000	10,000	10,799	50,805	609,660	449,640	935,720	2,879,924
				(100.00%)	(73.75%)	(153.48%)	(472.38%)
1600	24,000	25,198	122,466	1,469,592	1,080,524	2,258,796	7,023,964
				(100.00%)	(73.53%)	(153.70%)	(477.95%)
1600	30,000	31,285	154,395	1,852,740	1,360,304	2,850,912	8,870,200
				(100.00%)	(73.42%)	(153.88%)	(478.76%)
2000	36,000	37,596	185,092	2,221,104	1,631,124	3,418,412	10,668,648
				(100.00%)	(73.44%)	(153.91%)	(480.33%)
2000	40,000	41,936	208,352	2,500,224	1,834,564	3,851,612	12,019,048
				(100.00%)	(73.38%)	(154.05%)	(480.72%)
10,000	120,000	127,119	606,233	7,274,796	5,358,344	11,201,120	35,233,888
				(100.00%)	(73.66%)	(153.97%)	(484.33%)
10,000	160,000	167,504	817,728	9,812,736	7,211,844	15,130,228	47,646,464
				(100.00%)	(73.49%)	(154.19%)	(485.56%)
16,000	200,000	211,039	1,009,279	12,111,348	8,918,392	18,647,660	58,712,052
				(100.00%)	(73.64%)	(153.97%)	(484.77%)
16,000	220,000	231,439	1,116,473	13,397,676	9,857,544	20,645,480	65,034,080
				(100.00%)	(73.58%)	(154.10%)	(485.41%)
20,000	280,000	297,293	1,442,651	17,311,812	12,730,384	26,730,648	84,262,608
				(100.00%)	(73.54%)	(154.41%)	(486.73%)
20,000	320,000	337,056	1,649,792	19,797,504	14,546,564	30,563,432	96,370,464
				(100.00%)	(73.48%)	(154.38%)	(486.78%)

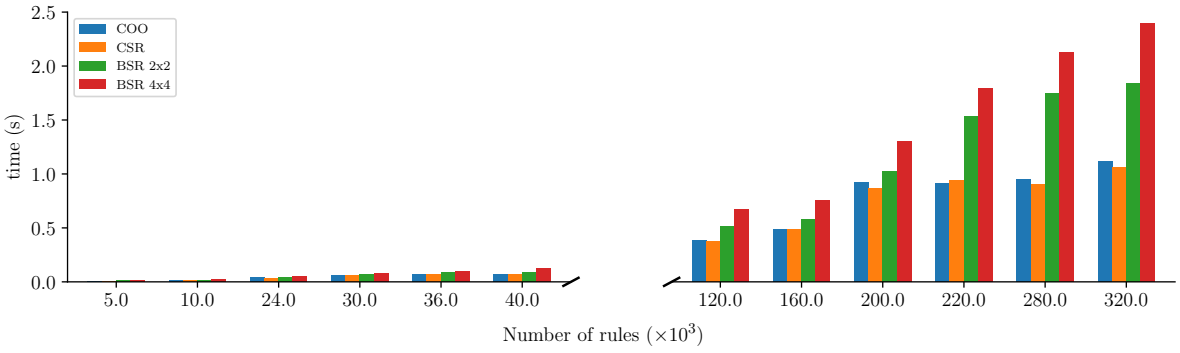


Figure 3.6: Comparison of execution time between different sparse representations on definite programs.

Table 3.8: Comparison of different sparse representations in terms of the memory size on definite programs for the transitive closure problem in Table 3.4.

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

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

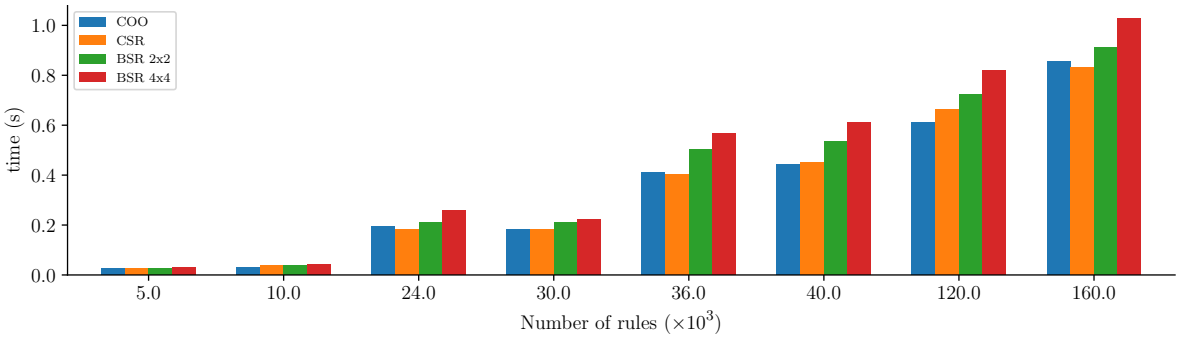


Figure 3.7: Comparison of execution time between different sparse representations on definite programs for the transitive closure problem.

The experimental results for definite programs, definite programs for the transitive closure problem and normal programs are illustrated in Table 3.7, Table 3.8 and Table 3.9 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 Section 3.2. On the other hand, the data for the BSR format shows 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.

Table 3.9: Comparison of different sparse representations in terms of the memory size on normal programs in Table 3.5.

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

n	m	n'	k	η_z	**COO**	**CSR/CSC**	**BSR** 2×2	**BSR** 4×4
1000	5000	6379	8	26,147	313,764	249,176	487,336	1,514,852
					(100.00%)	(79.42%)	(155.32%)	(482.80%)
1000	10,000	12,745	8	54,814	657,768	478,512	1,019,432	3,163,488
					(100.00%)	(72.75%)	(154.98%)	(480.94%)
1600	24,000	30,061	8	135,661	1,627,932	1,149,288	2,528,412	7,840,124
					(100.00%)	(70.60%)	(155.31%)	(481.60%)
1600	30,000	36,402	7	183,703	2,204,436	1,533,624	3,402,460	10,528,348
					(100.00%)	(69.57%)	(154.35%)	(477.60%)
2000	36,000	42,039	5	205,800	2,469,600	1,726,400	3,824,712	11,829,348
					(100.00%)	(69.91%)	(154.87%)	(479.00%)
2000	40,000	48,187	8	238,597	2,863,164	1,988,776	4,437,184	13,764,920
					(100.00%)	(69.46%)	(154.97%)	(480.76%)
10,000	120,000	171,967	6	716,115	8,593,380	6,128,920	13,523,496	42,851,300
					(100.00%)	(71.32%)	(157.37%)	(498.65%)
10,000	160,000	207,432	7	917,746	11,012,952	7,741,968	17,043,484	52,633,948
					(100.00%)	(70.30%)	(154.76%)	(477.93%)
16,000	200,000	250,194	5	1,129,348	13,552,176	9,674,784	21,203,960	66,035,684
					(100.00%)	(71.39%)	(156.46%)	(487.27%)
16,000	220,000	278,190	6	1,547,360	18,568,320	13,018,880	29,028,448	90,012,932
					(100.00%)	(70.11%)	(156.33%)	(484.77%)
20,000	280,000	357,001	4	1,841,749	22,100,988	15,533,992	34,219,356	106,039,484
					(100.00%)	(70.29%)	(154.83%)	(479.80%)
20,000	320,000	396,128	4	2,092,310	25,107,720	17,538,480	39,012,940	120,359,688
					(100.00%)	(69.85%)	(155.38%)	(479.37%)

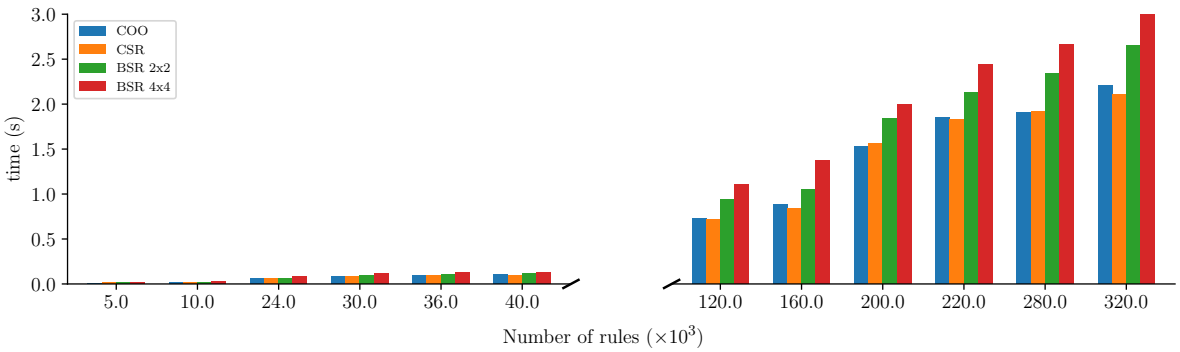


Figure 3.8: Comparison of execution time between different sparse representations on normal programs.

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:

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

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 3.10. 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.

Table 3.10: Comparison of different sparse representations in terms of the memory size on special programs as defined above.

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

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

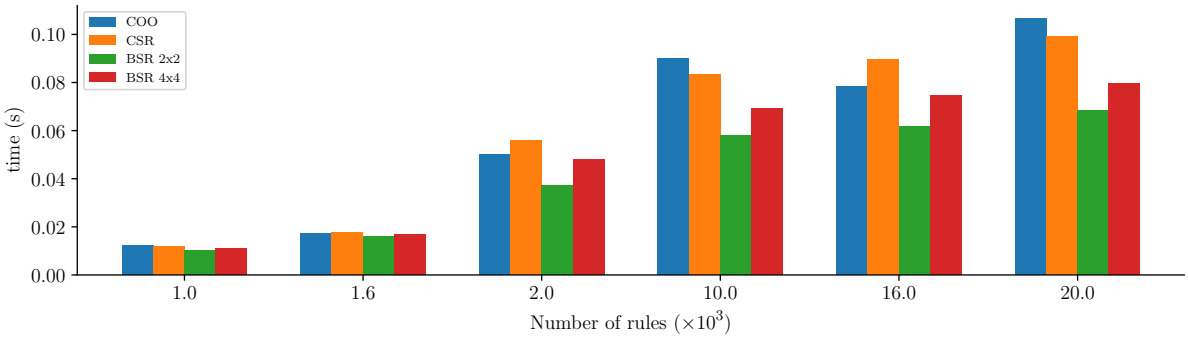
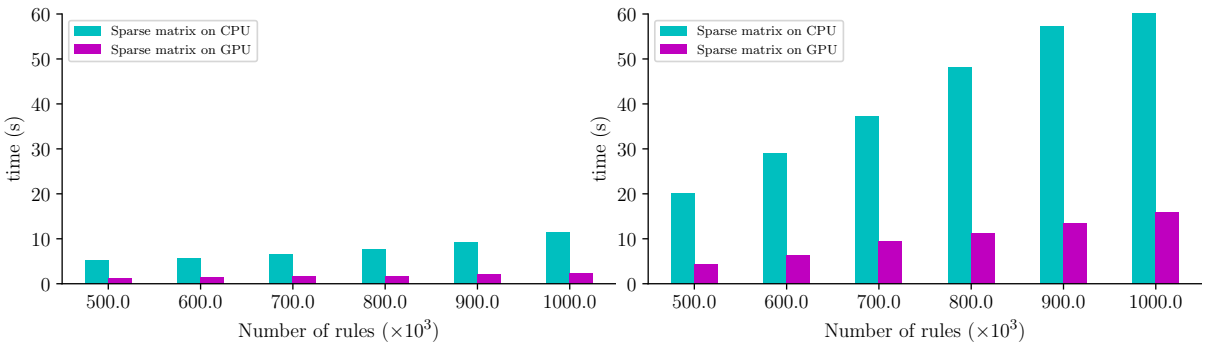


Figure 3.9: Comparison of execution time between different sparse representations on special programs.

3.4.4 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 [8].

As we can see in Figure 4.3, the implementation on GPU is faster than that on CPU approximately 3 to 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.



(a) Definite programs with higher sparsity level (0.99) (b) Definite programs with lower sparsity level (0.95)

Figure 3.10: Comparison of execution time between sparse matrix implementations on CPU and GPU.

[8] CUDA version 10.0.130

Table 3.11: Details of experimental results of sparse matrix implementations on CPU and GPU (higher sparsity level).

Time unit is second.

n	m	Sparsity	**CPU time**	**GPU time**
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

Table 3.12: Details of experimental results of sparse matrix implementations on CPU and GPU (lower sparsity level).

Time unit is second.

n	m	Sparsity	**CPU time**	**GPU time**
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

3.5 Conclusion

In this thesis, we analyze 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. 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 Section 3.4, we can consider applying the BSR format or maybe other methods that meet the need.

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 [80]. 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 [7], 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 [62]. Further research directions on implementing disjunctive LP and abductive LP should be considered in order 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 [64]. 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.

Chapter 4

Linear Algebraic Computation in Abductive Reasoning

The linear algebraic approach for abduction was considered by Aspis, Broda, and Russo [6]. They have defined an explanatory operator based on third-order tensor for computing abduction in Horn propositional programs that simulates deduction through Clark completion for the abductive program [15]. Unfortunately, the dimension explosion would arise in their method if they have not considered a more sophisticated elimination strategy. Additionally, 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. and Aspis et al. have not yet reported an empirical work.

In this chapter, we explore different approaches for linear algebraic abduction in top-down approach starting from the observation set. First, we investigate on how to define Propositional Horn Clause Abduction Problem (PHCAP) in the language of linear algebra. Then, we propose a linear algebraic method for computing all minimal explanations. We further discuss on other special characteristics of sparse representation and the difference of computing methods among dense and sparse formats. In Section 4.4, we demonstrate the efficiency of our computation method with other traditional approaches.

4.1 Linear Algebraic Encoding of Propositional Horn Clause Abduction Problem

Before diving deeper into the detail method, let us consider again the very simple example in the beginning of the previous section that we consider a very simple program

with only a single *And*-rule $p \leftarrow q \wedge r$. We already know the program matrix of its is:

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

By simply transposing it, we can obtain a new matrix:

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

Now let us see the behavior if we multiply the new matrix with a vector (1, 0, 0).

$$\begin{array}{c} \\ p \\ q \\ r \end{array} \begin{array}{c} \begin{array}{ccc} p & q & r \end{array} \\ \begin{pmatrix} 0 & 1/2 & 1/2 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \end{array} \cdot \begin{array}{c} \\ p \\ q \\ r \end{array} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{array}{c} \\ p \\ q \\ r \end{array} \begin{pmatrix} 0 \\ 1/2 \\ 1/2 \end{pmatrix}$$

Surprisingly, the behavior is similar to abduce the explanation of p and we can say that in order to explain p, we have to explain both q and r. Of course this example here is not well general enough but it gives us an initial idea about how to compute abduction in vector spaces. Now we move on to the more formal theory of the linear algebraic method for abduction.

Definition 43. Horn clause abduction: An abduction problem consists of a tuple $\langle \mathscr{L}, \mathbb{H}, \mathbb{O}, \mathrm{P} \rangle$, where $\mathbb{H} \subseteq \mathscr{L}$ (called *hypotheses* or *abducibles*), $\mathbb{O} \subseteq \mathscr{L}$ (called *observations*), and P is a consistent Horn logic program.

In this thesis, we assume a program P is *acyclic* [2] and in its standardized form. A program P is acyclic if the *dependency graph* of P is acyclic or it contains no loop. The *dependency graph* of a logic program P is a graph (V, E), where the nodes V are the atoms of P and, for each rule from P, there are edges in E from the atoms appearing in the body to the atom in the head.

For convenience, P is partitioned into $\mathrm{P}_{And} \cup \mathrm{P}_{Or}$ where P_{And} is a set of *And*-rules (including facts) of the form (3.1) and P_{Or} is a set of *Or*-rules of the form (3.2).

$$\begin{array}{rrcll} \textit{And}\text{-rule} & h & \leftarrow & b_1 \wedge \cdots \wedge b_m & (m \geq 0) \\ \textit{Or}\text{-rule} & h & \leftarrow & b_1 \vee \cdots \vee b_n & (n \geq 2) \end{array}$$

Given P, define $head(\mathrm{P}) = \{head(r) \mid r \in \mathrm{P}\}$, $head(\mathrm{P}_{And}) = \{head(r) \mid r \in \mathrm{P}_{And}\}$, and

$head(\mathrm{P}_{Or}) = \{head(r) \mid r \in \mathrm{P}_{Or}\}$.

Without loss of generality, we assume that any abducible atom $h \in \mathbb{H}$ does not appear in any head of rule in P. If there exists $h \in \mathbb{H}$ and a rule $r : h \leftarrow body(r) \in P$, we can replace r with $r' : h \leftarrow body(r) \vee h'$ in P and then replace h by h' in $\mathbb{H}$. If r is in the form (3.2), then r' is an Or-rule and no need to further update r'. On the other hand, if r is in the form (3.1), then we can update r' to become an Or-rule by introducing an And-rule $b_r \leftarrow body(r)$ in P and then replace $body(r)$ by b_r in r'.

Definition 44. Explanation of PHCAP: A set $E \subseteq \mathbb{H}$ is a *solution* of a PHCAP $\langle \mathscr{L}, \mathbb{H}, \mathbb{O}, \mathrm{P} \rangle$ if $\mathrm{P} \cup E \vDash \mathbb{O}$ and $\mathrm{P} \cup E$ is consistent. E is also called an *explanation* of $\mathbb{O}$. An explanation E of $\mathbb{O}$ is *minimal* if there is no explanation E' of $\mathbb{O}$ such that $E' \subset E$.

According to [87], deciding the set of all minimal solutions of the PHCAP $\mathbb{E} \neq \emptyset$ is NP-complete that is proved by a transformation from satisfiability problem [30].

In this thesis, we want to find the set $\mathbb{E}$ of minimal explanations E for a PHCAP $\langle \mathscr{L}, \mathbb{H}, \mathbb{O}, \mathrm{P} \rangle$.

The key idea of doing logic inference is to incorporate set operations and handle them by manipulating real values in vector spaces. We will give an overview of how to create the vector space and how to perform set operations on that vector space.

Definition 45. Vector represetation of subsets in PHCAP: Any subset $s \subseteq \mathscr{L}$ can be represented by a vector v of the length $|\mathscr{L}|$ such that the i-th value $v[i] = 1$ $(1 \leq i \leq |\mathscr{L}|)$ iff the i-th atom p_i of $\mathscr{L}$ is in s; otherwise $v[i] = 0$.

This definition is similar to the definition interpretation vector which is defined in [75]. Here we need to define for both deductive and abductive reasoning so we use the term correspondent vector.

In some specific cases, we also use v as a vectorizing method to define a set in vector spaces: $v(\mathbb{O})$ observation vector, $v(\mathbb{H})$ hypotheses vector, $v(\bot)$ integrity vector (shorthand of $v(\{\bot\})$ where $\{\bot\} \subset \mathscr{L}$), $v(head(\mathrm{P}_{And}))$ vector of all atoms that appear in the head of a conjunctive rule in P_{And}, $v(head(\mathrm{P}_{Or}))$ vector of all atoms that appear in the head of a disjunctive rule in P_{Or}. We use this notation for better indexing each element and vector value in the set/vector. If there is no need to indicate each individual item, we can omit the vectorizing method v.

Some basic notions In this chapter, we use some following notions:

- v: a vector. We refer to the i-th element of v by $v[i]$.

- M: a matrix. We refer to the i-th row of M by $M[i]$. Then $M[i]$ is a vector that we can refer to each element j by $M[i][j]$.
- M^T: a matrix which is the transpose of M.
- Inner product of two vectors u and v is denoted by $u \cdot v$.
- Matrix - vector multiplication between a matrix M and a vector v is denoted by $M \cdot v$.

Please note that we follow zero-based indexing that $v[0]$ is the first element of a vector v while $M[0]$ is the first column of a matrix M.

4.1.1 Program matrix and abductive matrix

We slightly modify the definition by Sakama et al. to define a matrix program of P in a vector space.

Definition 46. Matrix representation of standardized programs in PHCAP [75]: Let a PHCAP $\langle\ \mathscr{L}, \mathbb{H}, \mathbb{O}, P\ \rangle$ with P is a standardized program with $\mathscr{L} = \{p_1, \ldots, p_n\}$. Then P is represented by a *program matrix* $M_P \in \mathbb{R}^{n \times n}$ $(n = |\mathscr{L}|)$ 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_{And} and $m > 0$;
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_{Or};
3. $a_{ii} = 1$ if $p_i \leftarrow$ is in P_{And} or $p_i \in \mathbb{H}$.;
4. $a_{ij} = 0$, otherwise.

In Definition 46, we have an update in the condition 3 that we set 1 for all abducible atoms $p_i \in \mathbb{H}$. We further extend Definition 46 to define the abductive matrix of a theory P.

Definition 47. Abductive matrix of PHCAP: Suppose that a PHCAP has P with its *program matrix* M_P. The *abductive matrix* of P is the transpose of M_P represented as ${M_P}^T$.

Let us consider a logic circuit that can be formulated as a PHCAP in Example 6.

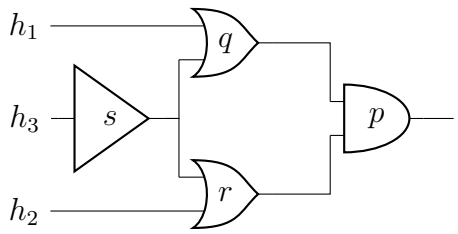


Figure 4.1: An example of logic circuit.

Example 6. Consider a PHCAP such that: $\mathscr{L} = \{p,\ q,\ r,\ s,\ h_1,\ h_2,\ h_3\}$, $\mathbb{O} = \{p\}$, $\mathbb{H} = \{h_1,\ h_2,\ h_3\}$, $\mathrm{P} = \{p \leftarrow q \wedge r,\ q \leftarrow h_1 \vee s,\ r \leftarrow s \vee h_2,\ s \leftarrow h_3\}$.
The program matrix and the abductive matrix of P are:

$$M_P = \begin{array}{c} \\ p \\ q \\ r \\ s \\ h_1 \\ h_2 \\ h_3 \end{array} \begin{array}{c} \begin{array}{ccccccc} p & q & r & s & h_1 & h_2 & h_3 \end{array} \\ \begin{pmatrix} & 1/2 & 1/2 & & & & \\ & & & 1 & 1 & & \\ & & & 1 & & 1 & \\ & & & & & & 1 \\ & & & & 1 & & \\ & & & & & 1 & \\ & & & & & & 1 \end{pmatrix} \end{array}, \quad {M_P}^T = \begin{array}{c} \\ p \\ q \\ r \\ s \\ h_1 \\ h_2 \\ h_3 \end{array} \begin{array}{c} \begin{array}{ccccccc} p & q & r & s & h_1 & h_2 & h_3 \end{array} \\ \begin{pmatrix} & & & & & & \\ 1/2 & & & & & & \\ 1/2 & & & & & & \\ & 1 & 1 & & & & \\ & 1 & & & 1 & & \\ & & 1 & & & 1 & \\ & & & 1 & & & 1 \end{pmatrix} \end{array}$$

In abduction, we also make use of the correspondent vector as defined in Definition 48 to realize an explanation - a set of propositional variables.

4.1.2 How a PHCAP can be represented in a vector space

By defining the program matrix and the abductive matrix, we can define correspondent vector of PHCAP in vector space.

Definition 48. Correspondent vector of PHCAP: Any subset $s \subseteq \mathscr{L}$ can be represented by a *corresponding vector* v of the length $|\mathscr{L}|$ such that the i-th value $v[i] = 1$ $(1 \leq i \leq |\mathscr{L}|)$ iff the i-th atom p_i of $\mathscr{L}$ is in s; otherwise $v[i] = 0$.

Without ambiguity, we will identify the set representation s with the vector representation v, so we denote them all as v from now on. Henceforth, v_i is the i-th atom of $\mathscr{L}$ that constitutes s, while $v[i]$ is the value of the vector at index i.

In some specific cases, we also use v as a special function that outputs a corresponding vector of a subset in vector spaces: $v(\mathbb{O})$ the observation vector, $v(\mathbb{H})$ the hypotheses vector, $v(\bot)$ the integrity vector (shorthand of $v(\{\bot\})$ where $\{\bot\} \subset \mathscr{L}$), $v(\mathit{head}(\mathrm{P}_{\mathit{And}}))$ the vector of all head atoms of *And*-rules in $\mathrm{P}_{\mathit{And}}$, $v(\mathit{head}(\mathrm{P}_{\mathit{Or}}))$ the vector of all head atoms of *Or*-rules in P_{Or}. We use this notation for better indexing

each element and a vector value in the set/vector. If there is no need to indicate each individual item, we can omit the function notation $v()$.

In order to utilize the use of correspondent vectors, we define a thresholding method to perform needed set operations in vector spaces.

Definition 49. **θ-thresholding:**

1. Given a value $x \in \mathbb{R}$, define $\theta(x) = x'$ such that $x' = 1$ if $x > 0$; otherwise, $x' = 0$
2. Given a vector $v \in \mathbb{R}^n$, define $\theta(v) = v'$ such that $v'[i] = 1$ if $v[i] > 0$; otherwise $v'[i] = 0$
3. Given a matrix $M \in \mathbb{R}^{n \times m}$, define $\theta(M) = M'$ such that $M'[i][j] = 1$ if $M[i][j] > 0$; otherwise $M'[i][j] = 0$

where $1 \leq i \leq n,\ 1 \leq j \leq m$.

Proposition 6. The following equivalence relations hold :

$$u \cap v = \emptyset \Leftrightarrow u \cdot v = 0 \tag{4.1}$$

$$u \cap v \neq \emptyset \Leftrightarrow u \cdot v > 0 \tag{4.2}$$

$$u \subseteq v \Leftrightarrow \theta(u + v) \leq \theta(v) \tag{4.3}$$

where $\cdot$ is the inner product.

Proof.

- (4.1) $u \cap v = \emptyset \Leftrightarrow u \cdot v = 0$
  - If $u \cap v = \emptyset$, then based on Definition 48, $\{i \mid u[i] = 1\} \cap \{j \mid v[j] = 1\} = \emptyset$. So $u \cap v = \emptyset \Rightarrow u \cdot v = 0$.
  - If $u \cdot v = 0$, then there is no index i such that $u[i] = 1$ and $v[i] = 1$. So $\{i \mid u[i] = 1\} \cap \{j \mid v[j] = 1\} = \emptyset$ or $u \cap v = \emptyset$ by Definition 48.
- (4.2) $u \cap v \neq \emptyset \Leftrightarrow u \cdot v > 0$
  - If $u \cap v = \emptyset$, then there is at least an index i such that $u[i] = 1$ and $v[i] = 1$. So $u \cdot v > 0$.
  - If $u \cdot v > 0$, then we must find at least an index i such that $u[i] = 1$ and $v[i] = 1$. So $\{i \mid u[i] = 1\} \cap \{j \mid v[j] = 1\} \neq \emptyset$ or $u \cap v \neq \emptyset$ by Definition 48.
- (4.3) $u \subseteq v \Leftrightarrow \theta(u + v) \leq \theta(v)$

– If $u \subseteq v$, then $\{i \mid u[i] = 1\} \subseteq \{j \mid v[j] = 1\}$ by Definition 48. Assigning $z = u + v$ then we have $\{k \mid z[k] = 1\} \subseteq \{j \mid v[j] = 1\}$. By applying θ, we limit all values in those vectors by 1, so we have $\theta(z) \leq \theta(v)$ or $\theta(u + v) \leq \theta(v)$.

– If $\theta(u + v) \leq \theta(v)$, then $\{k \mid z[k] = 1\} \subseteq \{j \mid v[j] = 1\}$, where $z = u + v$. Accordingly, $z \subseteq v$ by Definition 48. It is obvious that $u \subseteq z$, so $u \subseteq v$.

□

4.2 Linear Algebraic Computation

In this section, we will present the linear algebraic method for abduction in detail with a single example of PHCAP throughout the whole section. According to the previous section, we can define the vector space for the PHCAP as well as the program matrix and the abductive matrix of the theory associated with the PHCAP.

Example 7. Consider a PHCAP such that:
$\mathscr{L} = \{p,\ q,\ r,\ s,\ h_1,\ h_2,\ h_3\}$, $\mathbb{O} = \{p\}$, $\mathbb{H} = \{h_1,\ h_2,\ h_3\}$,
$\mathrm{P} = \{p \leftarrow q \wedge r,\ q \leftarrow h_1 \vee s,\ r \leftarrow s \vee h_2,\ s \leftarrow h_3\}$.
The program matrix and the abductive matrix of P are [1]:

$$M_P = \begin{array}{c} \\ p \\ q \\ r \\ s \\ h_1 \\ h_2 \\ h_3 \end{array} \begin{array}{c} \begin{array}{ccccccc} p & q & r & s & h_1 & h_2 & h_3 \end{array} \\ \begin{pmatrix} & 1/2 & 1/2 & & & & \\ & & & 1 & 1 & & \\ & & & 1 & & 1 & \\ & & & & & & 1 \\ & & & & 1 & & \\ & & & & & 1 & \\ & & & & & & 1 \end{pmatrix} \end{array}, \quad {M_P}^T = \begin{array}{c} \\ p \\ q \\ r \\ s \\ h_1 \\ h_2 \\ h_3 \end{array} \begin{array}{c} \begin{array}{ccccccc} p & q & r & s & h_1 & h_2 & h_3 \end{array} \\ \begin{pmatrix} & & & & & & \\ 1/2 & & & & & & \\ 1/2 & & & & & & \\ & 1 & 1 & & & & \\ & 1 & & & 1 & & \\ & & 1 & & & 1 & \\ & & & 1 & & & 1 \end{pmatrix} \end{array}$$

4.2.1 Vector representation and explanation vector

The goal of PHCAP is to find the set of minimal explanations $\mathbb{E}$ according to Definition 44. Using Definition 48, we can represent any $E \in \mathbb{E}$ by a column vector $E \in \mathbb{R}^{|\mathscr{L}| \times 1}$. To compute E, we define an *explanation vector* $v \in \mathbb{R}^{|\mathscr{L}| \times 1}$. We use the explanation vector v to demonstrate linear algebraic computation of abduction to reach an explanation E starting from an initial vector $v = v(\mathbb{O})$ which is the observation vector (note that we can use the notation $\mathbb{O}$ as a vector without the function notation $v()$ as stated before). At each computation step, we can interpret the meaning of the explanation vector v as: in order to explain $\mathbb{O}$, we have to explain all atoms v_i such that $v[i] > 0$.

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

Definition 50. Answer of a PHCAP: The explanation vector v *reaches* an answer E if $v \subseteq \mathbb{H}$. This condition can be written in linear algebra as follows:

$$\theta(v + \mathbb{H}) \leq \theta(\mathbb{H}) \tag{4.4}$$

where $\mathbb{H}$ is the short hand of $v(\mathbb{H})$ which is the hypotheses set/vector mentioned above.

Similar to the minimality characteristic in Definition 44, an answer is called *minimal* iff there is no other subset of it is also an answer.

Next, we define the consitency checking of a vector by employing the least fixpoint of the T_P-operator [93]. According to [77], $lfp(M_P, v)$ is the vector representation of the least fixpoint of the T_P-operator starting from v. In orther words, from a vector v, by continuously applying matrix multiplication, we will obtain a fixpoint which represents all atoms can be reached from v through deductive steps. This is the key to define the consistency checking as following:

Proposition 7. An explanation vector v is *consistent* with P if $lfp(M_P, v) \cap \{\bot\} \neq \emptyset$. This condition can be written in linear algebra as follows:

$$v(\bot) \cdot lfp(M_P, v) = 0 \tag{4.5}$$

where M_P is the program matrix of P.

Proof. The *lfp* can be computed in the vector space by applying matrix multiplication $M_P \cdot v$ continuously until the fixpoint is reached. The resulting vector corresponds to the least model of $P \cup v$ [75]. If this model contains $\bot$, $P \cup v$ is inconsistent; otherwise v is consistent with P. We can perform this test using Definition 6. □

An efficient method to compute *lfp* of a definite program has been developed in [63, 1].

4.2.2 The two 1-step abduction in $\mathbf{P}_{And}$ and $\mathbf{P}_{Or}$

We now define 1-step abduction in PHCAP step by step. First, let us define the *reduct abductive matrix*.

Definition 51. Reduct abductive matrix of PHCAP: We can obtain a *reduct abductive matrix* $M_P(P_{And})^T$ from the abductive matrix ${M_P}^T$ by:

1. Removing all columns *w.r.t.* Or-rules in P_{Or}.
2. Setting 1 at the diagnal corresponding to all atoms which are at the head of Or-rules.

The reduct abductive matrix is a reduced version of the abductive that we define it to use in the 1-step abduction for P_{And}. We use the superscript (t) to denote the explanation vector v at a step t. The 1-step abduction applies on v at a step t to reach a new vector v at the next step.

Definition 52. 1-step abduction for P_{And} of a vector:

$$v^{(t+1)} = M_P(P_{And})^T \cdot v^{(t)} \tag{4.6}$$

The 1-step abduction (4.6) is a reverse version of the T_P-operator on a single vector. By transposing the program matrix to an abductive matrix, we represent the abductive step in a vector space that computes the explanation $v^{(t+1)}$ for $v^{(t)}$. This step corresponds to a deductive step through Clark completion in an SD program [15]. Suppose that there is an index i such that $v_i \in v^{(t)} \cap head(P_{And})$, according to Definition 46 and Definition 47 there is a column *w.r.t.* v_i in $M_P(P_{And})^T$, denoted by $col(v_i)$. By applying (4.6), $v^{(t+1)}[j] = \dfrac{v^{(t)}[i]}{|col(v_i)|} > 0$, for any j such that $v_j^{(t+1)} \in col(v_i)$. Then vector $v^{(t+1)}$ represents the set of atoms required to explain $v^{(t)}$.

Example 8 (cont. Example 7)**.** $\mathrm{P}_{And} = \{p \leftarrow q \wedge r,\ s \leftarrow h_3\}$. We can obtain a reduct abductive matrix $M_P(\mathrm{P}_{And})^T$ by removing columns *w.r.t.* rules $\{q \leftarrow h_1 \vee s,\ r \leftarrow s \vee h_2\}$ in the original abductive matrix. Consider applying 1-step abduction for P_{And} with $v^{(t)} = \mathbb{O}$:

$$v^{(t)} = (1,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0)^T\ (= \mathbb{O})$$

$$v^{(t+1)} = M_P(\mathrm{P}_{And})^T \cdot v^{(t)}$$

$$= \begin{array}{c} \\ p \\ q \\ r \\ s \\ h_1 \\ h_2 \\ h_3 \end{array} \begin{array}{c} \begin{array}{ccccccc} p & q & r & s & h_1 & h_2 & h_3 \end{array} \\ \begin{pmatrix} & & & & & & \\ 1/2 & 1 & & & & & \\ 1/2 & & 1 & & & & \\ & & & & & & \\ & & & & 1 & & \\ & & & & & 1 & \\ & & & 1 & & & 1 \end{pmatrix} \end{array} \cdot \begin{array}{c} \\ p \\ q \\ r \\ s \\ h_1 \\ h_2 \\ h_3 \end{array} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{array}{c} \\ p \\ q \\ r \\ s \\ h_1 \\ h_2 \\ h_3 \end{array} \begin{pmatrix} 0 \\ 1/2 \\ 1/2 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

The vector $v^{(t+1)}$ can be interpreted as: in order to explain p, both q and r are to be explained.

Definition 52 illustrates that we can apply continuously the 1-step abduction (4.6) with $v^{(0)} = \mathbb{O}$ until it reaches an explanation by the condition in Definition 50 and satisfies consistency in Proposition 7. In fact, Definition 50 may not hold in case where there is an atom in the explanation vector that we have no rule in P_{And} to apply to find its explanation.

Proposition 8. The summation of $v^{(t)}$ is bounded.

$$sum(v^{(t+1)}) \leq sum(v^{(t)}) \leq \cdots \leq sum(v^{(0)}) \tag{4.7}$$

where $sum(v) = \Sigma_{v_i \in v} v[i]$.

This proposition is trivial to prove using Definition 46 and Definition 47. For simplicity, we can initialize the starting point $v^{(0)}$ that statisfies $sum(v^{(0)}) = 1$. If there are multiple observations $o_1, o_2, \ldots, o_k \in \mathbb{O}$, a new atom o is introduced to replace the current observation set. Then a new conjunctive rule $o \leftarrow o_1 \wedge o_2 \wedge \cdots \wedge o_k$ is introduced to the theory P. Then we can initialize the starting point $\mathbb{O} = \{o\}$ such that summation of the corresponding vector is 1. From now on, we assume $sum(v^{(0)}) = 1$ without loss of generality.

Proposition 9. If $sum(v^{(t)}) < 1$, then $v^{(t)} \cup \mathrm{P}_{And} \nvDash \mathbb{O}$.

Proof. According to Proposition 8, for any explanation vector $v^{(t)}$, we have $sum(v^{(t)}) \leq sum(v^{(t-1)}) \leq \cdots \leq sum(v^{(0)}) = 1$. Assume the equality holds until the step $t-1$ of the 1-step abduction (4.6). If there is any index i such that $v_i \in v^{(t-1)} \setminus head(\mathrm{P}_{And})$, the column *w.r.t.* v_i in the reduct abductive matrix is encoded as a zero column. Thus, when applying matrix multiplication in Definition 52, at the index i, $v^{(t)}[i] = 0$ while $v^{(t-1)}[i] > 0$. That is: $sum(v^{(t-1)}) - sum(v^{(t)}) \geq v^{(t-1)}[i] > 0 \Leftrightarrow sum(v^{(t)}) < 1$. This behavior is equivalent to considering an explanation of v_i but there is no rule in P_{And} that can explain v_i. □

Combining Definition 52 with Definition 50 and conditions in Proposition 7 and Proposition 9, we can deal with *And*-rules in P_{And}. This is just an initial step to solve the PHCAP $\langle \mathscr{L}, \mathbb{H}, \mathbb{O}, \mathrm{P} \rangle$. In abductive reasoning, *Or*-rules are more complicated to handle because they increase the number of possible explanations. Hence, we need an efficient method dealing with the growth of possibilities in vector spaces.

According to Definition 48, an explanation vector v can be represented by a column vector $v \in \mathbb{R}^{|\mathscr{L}| \times 1}$. We can stack multiple vectors v to form an explanation matrix $M \in \mathbb{R}^{|\mathscr{L}| \times |M|}$ while all definitions and propositions with the 1-step abduction for P_{And} of a vector still work. Therefore, we can rewrite Definition 52 as follows:

Definition 53. 1-step abduction for P_{And}:

$$M^{(t+1)} = M_P(\mathrm{P}_{And})^T \cdot M^{(t)} \tag{4.8}$$

We now introduce a notation M as a matrix that is equivalent to a vector of vectors or a set of sets. Note that we denote $|M|$ as the number of vectors or sets in M. We

also use the same notation we mentioned above that M_i is the i-th set of M, while $M[i]$ is the vector at an index i.

Let v be an explanation vector in $\langle \mathscr{L}, \mathbb{H}, \mathbb{O}, \mathrm{P} \rangle$ such that $v \cap head(\mathrm{P}_{Or})$ where $head(\mathrm{P}_{Or}) = \{head(r_1),\ head(r_2),\ \ldots,\ head(r_k)\}$ with $r_1,\ r_2,\ \ldots,\ r_k \in \mathrm{P}_{Or}$. In order to compute explanations of v we have to explore all combinations c extracted from $\{body(r_1),\ body(r_2),\ \ldots,\ body(r_k)\}$ such that $\forall j \in \{1,\ 2,\ \ldots,\ k\}$, $c \cap body(r_j) \neq \emptyset$. It turns out that this is equivalent to enumerate the Minimal Hitting Sets (MHS) with the input set is $\{body(r_1),\ body(r_2),\ \ldots,\ body(r_k)\}$ [28].

We denote $\mathbf{MHS}(\mathbb{S})$ as all MHS of a family of sets to be hit $\mathbb{S}$. Now we can define 1-step abduction for P_{Or}.

Definition 54. 1-step abduction for $\mathbf{P}_{Or}$:

$$M^{(t+1)} = \bigcup_{\forall v \in M^{(t)}} \bigcup_{\forall s \in \mathbf{MHS}(\mathbb{S}_{(v,\ \mathrm{P}_{Or})})} \Big(\big(v \setminus head(\mathrm{P}_{Or})\big) \cup s \Big) \tag{4.9}$$

where: $\mathbb{S}_{(v,\ \mathrm{P}_{Or})} = \{body(r_1), body(r_2), \ldots, body(r_k)\}$ is a family of sets to be hit such that $v \cap head(\mathrm{P}_{Or}) = \{head(r_1), head(r_2), \ldots, head(r_k)\}$.

Note that all new vectors $v \in M^{(t+1)}$ will be reallocated values such that $sum(v) = 1$ to maintain the condition in Proposition 9 of the 1-step abduction (4.8) for P_{And}.

Example 9 (cont. Example 8)**.** $\mathrm{P}_{or} = \{q \leftarrow h_1 \vee s,\ r \leftarrow s \vee h_2\}$. We use the output of Example 8 as the input of the 1-step abduction for P_{Or}, but now we treat it as a matrix instead:

$$\begin{aligned}
M^{(t)^T} &= \begin{array}{c} \\ 0 \end{array} \begin{array}{c} \begin{array}{ccccccc} p & q & r & s & h_1 & h_2 & h_3 \end{array} \\ \begin{pmatrix} 0 & 1/2 & 1/2 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \\
M^{(t)} &= \{\{q, r\}\} \\
\mathbb{S}_{(M_0^{(t)},\ \mathrm{P}_{Or})} &= \{\{h_1,\ s\},\ \{s,\ h_2\}\} \\
\mathbf{MHS}(\mathbb{S}_{(M_0^{(t)},\ \mathrm{P}_{Or})}) &= \{\{s\},\ \{h_1,\ h_2\}\} \\
M^{(t+1)} &= \{\{s\},\ \{h_1,\ h_2\}\} \\
M^{(t+1)^T} &= \begin{array}{c} \\ 0 \\ 1 \end{array} \begin{array}{c} \begin{array}{ccccccc} p & q & r & s & h_1 & h_2 & h_3 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1/2 & 1/2 & 0 \end{pmatrix} \end{array}
\end{aligned}$$

To the best of our knowledge, it is not trivial to implement an efficient method in a vector space that enumerates exactly all MHS as we defined in Definition 54. Hence, to implement (4.9) at this time, we have no choice but to treat all explanation vectors as sets instead of vectors. Fortunately, we can perform the vector-set conversion with minimal cost using the sparse representation we are going to discuss later.

4.2.3 Computable characteristics

Up to now, we have defined 1-step abduction for P_{And} and P_{Or}. Although each method itself is not sufficient to solve the PHCAP $\langle \mathscr{L}, \mathbb{H}, \mathbb{O}, \mathrm{P} \rangle$, their characteristics are important for us to define a general approach. Hereafter we will define the characteristic of each 1-step abduction in solving a PHCAP.

Definition 55. *Or*-**computable and** *And*-**computable:**

1. A vector v is *Or*-computable iff $v \cap head(\mathrm{P}_{Or}) \neq \emptyset$.
2. A matrix M is *Or*-computable iff $\exists v \in M$, v is *Or*-computable.
3. A vector v is *And*-computable iff v is not *Or*-computable.
4. A matrix M is *And*-computable iff $\forall v \in M$, v is not *Or*-computable.

Proposition 10. For any matrix M which is *Or*-computable in $\langle \mathscr{L}, \mathbb{H}, \mathbb{O}, \mathrm{P} \rangle$, there exists a fixpoint t of (4.9), such that $M^{(t+k)} = M^{(t)}$, $\forall k > 0,\ k \in \mathbb{N}$.

Proof. For each *Or*-computable vector $v \in M$, the 1-step abduction (4.9) replaces all atoms in the intersection of v and $head(\mathrm{P}_{Or})$ by the corresponding MHS. In addition, P is finite and acyclic so there is a fixpoint such that there is no *Or*-rule that can be used to abduce v or we can say that v is *And*-computable. That means $v \cap head(\mathrm{P}_{Or}) = \emptyset$, so the corresponding MHS is an empty set then $\forall k > 0$, $v^{(t+k)} = v^{(t)}$ $(k \in \mathbb{N})$. Extend this to other explanation vectors in M we have that M is *And*-computable and $\forall k > 0$, $M^{(t+k)} = M^{(t)}$ $(k \in \mathbb{N})$. □

Corollary 1. For any matrix M which is *Or*-computable in $\langle \mathscr{L}, \mathbb{H}, \mathbb{O}, \mathrm{P} \rangle$, if t is the fixpoint of (4.9) then $M^{(t)}$ is *And*-computable in $\langle \mathscr{L}, \mathbb{H}, \mathbb{O}, \mathrm{P} \rangle$.

Proposition 11. For any matrix M which is *And*-computable in $\langle \mathscr{L}, \mathbb{H}, \mathbb{O}, \mathrm{P} \rangle$, $M_P(\mathrm{P}_{And})^T \cdot M = {M_P}^T \cdot M$.

Proof. As in Definition 52, $M_P(\mathrm{P}_{And})^T$ is a reduct abductive matrix from ${M_P}^T$ by removing all columns *w.r.t.* *Or*-rules in P_{Or}. So $M_P(\mathrm{P}_{And})^T \cdot M$ has no effect on *Or*-computable vector $v \in M$. Moreover, M is *And*-computable in $\langle \mathscr{L}, \mathbb{H}, \mathbb{O}, \mathrm{P} \rangle$ by definition, therefore $M_P(\mathrm{P}_{And})^T \cdot M = {M_P}^T \cdot M$. □

4.2.4 The algorithm

Based on the two 1-step abduction (4.8) and (4.9), we propose an exhaustive search strategy to solve the PHCAP $\langle \mathscr{L}, \mathbb{H}, \mathbb{O}, \mathrm{P} \rangle$ in a vector space as illustrated in Algorithm 4.

Some explanations are in order:

Algorithm 4 Explanations finding in a vector space

Input: PHCAP consists of a tuple $\langle \mathscr{L}, \mathbb{H}, \mathbb{O}, \mathrm{P} \rangle$
Output: Set of explanations $\mathbb{E}$

Create an abductive matrix ${M_P}^T$ from P
Initialize the observation matrix M from $\mathbb{O}$
$\mathbb{E} = \emptyset$
while True do
    $M' = {M_P}^T \cdot M$
    $M' = \textbf{consistent}(M')$ ▷ Proposition 7
    $v\_sum = sum_{col}(M') < 1 - \epsilon$ ▷ Proposition 9
    $M' = M'[v\_sum = \textbf{False}]$
    if $M' = M$ **or** $M' = \emptyset$ **then**
        $v\_ans = \theta(M + \mathbb{H}) \leq \theta(\mathbb{H})$ ▷ Definition 50
        $\mathbb{E} = \mathbb{E} \cup M[v\_ans = \textbf{True}]$
        return minimal$(\mathbb{E})$ ▷ Minimality check
    do
        $v\_ans = \theta(M' + \mathbb{H}) \leq \theta(\mathbb{H})$ ▷ Definition 50
        $\mathbb{E} = \mathbb{E} \cup M'[v\_ans = \textbf{True}]$
        $M' = M'[v\_ans = \textbf{False}]$
        $M = M \cup M'[\textbf{not } Or\text{-computable}]$
        $M' = M'[Or\text{-computable}]$
        $M' = \bigcup_{\forall v \in M'} \bigcup_{\forall s \in \mathbf{MHS}(\mathbb{S}_{(v,\, \mathrm{P}_{Or})})} \Big(\big(v \setminus head(\mathrm{P}_{Or})\big) \cup s \Big)$
        $M' = \textbf{consistent}(M')$ ▷ Proposition 7
    while $M' \neq \emptyset$

- Step 5: We can use $M_P(\mathrm{P}_{And})^T$ and $M_P{}^T$ interchangeably, and there is no difference in terms of the effect on algorithm behavior as proved in Proposition 11. In fact, using a reduct abductive matrix may slightly improve the performance.
- Step 6 & 20: Consistency checking accroding to Proposition 7.
- Step 7: $sum_{col}(M')$ means applying summation on each vector $v \in M'$ to return a vector. Then we compare each element of this vector with $1-\epsilon$ following the Proposition 9 to return a corresponding Boolean vector. Due to the numerical issue with floating-point numbers in computer *e.g.* $\frac{1}{3}+\frac{1}{3}+\frac{1}{3} = 0.999\ldots$, a small fraction ϵ is introduced to relax the condition in Proposition 9. Choosing the best ϵ depends on actual $\langle \mathscr{L}, \mathbb{H}, \mathbb{O}, \mathrm{P}\rangle$. If we set ϵ too small, we may filt out good explanation vectors and the algorithm might not gives expected output. While setting ϵ too large, we may waste of computation in unexplainable paths.
- Step 8: We use the Boolean vector in Step 7 to eliminate unexplainable explanation vectors. We keep only vectors that their Boolean value is **False**. [] is the projection method that extracts from M' only vectors that satisfy the condition inside []. Similarly, we also use the projection method in Steps 15-18.
- Step 10-11 & 14-15: Check the condition in Definition 50 and add all found explanations to $\mathbb{E}$.
- Step 12: Applying the minimality check on the set $\mathbb{E}$ to eliminate redundant explanations according to Definition 43. We implement this method by sorting all $E \in \mathbb{E}$ by their cardinality, then applying a simple set iteration loop.
- Step 17: Add vectors which are *And*-computable to the matrix M for the next loop.
- Steps 16,18-19: Construct a matrix M' which is Or-computable then perform the 1-step abduction (4.9). Here we have to solve the MHS problem many times. We implement a naive approach in which we enumerate all combinations then apply the minimality check similar to Step 12. However, this implementation can deal with up to 500,000 combinations, therefore, we exploit PySAT[2] to solve large-size MHS problems [41].

Theorem 3. The output of Algorithm 4 is the set of all minimal explanations of the PHCAP $\langle \mathscr{L}, \mathbb{H}, \mathbb{O}, \mathrm{P}\rangle$.

Proof. Definition 48 defines 1-1 correspondence between subsets of $\mathscr{L}$ and vectors. Algorithm 4 employs both the 1-step abduction (4.8) and (4.9) in a vector space, which are equivalent to abductive steps in P_{And} and P_{Or} respectively, exploring all

[2]https://github.com/pysathq/pysat

possibilities that satisfy both Definition 50 and Proposition 7. Therefore, $\forall E \in \mathbb{E}$, $E \subseteq \mathbb{H}$ we have $E \cup \mathrm{P} \vDash \mathbb{O}$ and $E \cup \mathrm{P} \nvDash \bot$. Futher, Algorithm 4 employs minimality check on $\mathbb{E}$, therefore $\forall E_1, E_2 \in \mathbb{E}$, $E_1 \nsubseteq E_2$. □

Example 10. Let us demonstrate how to solve the PHCAP in Example 7 using Algorithm 4. Actually, we have done the first iteration of Algorithm 4 as illustrated in Example 8 and Example 9. We continue the next iteration with the explanation matrix $M = M^{(t+1)}$ obtained in Example 9.

$$M^T = \begin{array}{c} \\ 0 \\ 1 \end{array} \begin{array}{c} \begin{array}{ccccccc} p & q & r & s & h_1 & h_2 & h_3 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1/2 & 1/2 & 0 \end{pmatrix} \end{array}$$

$$M'^T = \left(M_P{}^T \cdot M\right)^T = \begin{array}{c} \\ 0 \\ 1 \end{array} \begin{array}{c} \begin{array}{ccccccc} p & q & r & s & h_1 & h_2 & h_3 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1/2 & 1/2 & 0 \end{pmatrix} \end{array}$$

Here Algorithm 4 stops because all explanation vectors reach answer of Definition 50, satisfying the condition of Proposition 7, and $M' = \emptyset$ after that. Finally, the algorithm applies minimal checking and gives the output set of minimal explanations $\mathbb{E} = \{\{h_3\}, \{h_1, h_2\}\}$.

The main idea of Algorithm 4 is applying the 1-step abduction (4.8) and (4.9) continuously in a vector space. Except for the MHS enumerator, almost everything can be implemented using matrix operations. Therefore, it is possible to gain remarkable boosting performance by implementing a parallel version of Algorithm 4 using powerful BLAS library *e.g.* Intel MKL, NVIDIA cuBLAS. On the other hand, in some circumstances, because Algorithm 4 is an exhaustive search, it will take time to explore all possibilities. To obtain an acceptable set of explanations, we can apply some early stop conditions in Step 9 *e.g.* number of columns of M exceeds a fixed value or how many times the 1-step abdution have been called. To sum up, we have presented a general framework to solve PHCAP in a vector space that can be modified easily for different purposes.

4.3 Matrix Representation

In our previous work, we have analyzed the sparsity of logic programs in vector spaces and have a conclusion that program matrices are sparse in general [63, 1]. The paper indicates that implementing the T_P-operator using a sparse format outperforms that using the dense format in large-scale logic programs. Similarly, the sparse representation will be promising in abductive reasoning.

The sparsity of a matrix equals the number of zero-valued elements divided by the total number of elements [13]. By definition, there is no doubt that in a PHCAP $\langle \mathscr{L}, \mathbb{H}, \mathbb{O}, \mathrm{P} \rangle$, the sparsity of the abductive matrix and that of the program matrix are equal and can be computed by the following equation [63, 1]:

$$sparsity(\mathrm{P}) = 1 - \frac{\sum_{r \in \mathrm{P}} |body(r)|}{|\mathscr{L}|^2} \tag{4.10}$$

Extend the definition of sparsity to an explanation matrix M, we have the following equation:

$$sparsity(M) = 1 - \frac{\sum_{v \in M} |v|}{|\mathscr{L}| \times |M|} \tag{4.11}$$

Because M is growing while we explore different possible explanations, there is no warranty that M always has a high level of sparsity. In case M is not sparse ($sparsity(M) \leq 0.9$), although the sparse representation may not help much in terms of performance, it provides faster vector-set conversion. In Section 4.4, we will analyze more detail about the sparsity level of explanation matrices.

Example 11.

- Consider the theory P in Example 7:

$$sparsity(\mathrm{P}) = 1 - \frac{2+2+2+1+1+1+1}{7^2} = 0.796$$

- Consider the explanation matrix $M^{(t+1)}$ in Example 9:

$$sparsity(M^{(t+1)}) = 1 - \frac{1+2}{7 \times 2} = 0.786$$

Suppose we have a matrix M of the size $\mathbb{R}^{m \times n}$ with η_z non-zero elements. Now let us quick summarize three most popular general-purpose sparse matrix formats by considering the explanation matrix $V^{(t+1)}$ in Example 9.

1. **Coordinate (COO)** format uses 3 arrays of length η_z to store the two coordinates and value of non-zero elements following the row-major order or the column-major order. In row-major order, the consecutive elements of a row reside next to each other while in the column-major order, the consecutive elements of a column reside next to each other. We denote M_{row}, M_{col}, M_{val} for the row indices array, column indices array and values array respectively. We know that

$|M_{row}| = |M_{col}| = |M_{val}| = \eta_z$. $V^{(t+1)}$ can be represented in the COO format following the row-major order:

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

Then each non-zero element can be extracted by a tuple $M_{row}[i]$, $M_{col}[i]$, $M_{val}[i]$ for $0 \leq i < \eta_z$.

2. **Compressed Sparse Row (CSR)** format is similar to the COO format except that we use the compressed format for row indices and strictly follow the row-major order. We store $m + 1$ values for row indices rather than η_z values. We denote M_{row}, M_{col}, M_{val} for the compressed row indices array, column indices array and values array respectively. We know that $|M_{row}| = m+1$ while $|M_{col}| = |M_{val}| = \eta_z$. $V^{(t+1)}$ can be represented in the CSR format as follows:

Compressed row	0	0	0	0	1	2	3	3
Col index	0	1	1					
Value	1.0	0.5	0.5					

Then each non-zero element of the i-th row ($0 \leq i \leq m$) can be extracted by a tuple i, $M_{col}[j]$, $M_{val}[j]$ for $M_{row}[i] \leq j < M_{row}[i + 1]$.

3. **Compressed Sparse Column (CSC)** format is similar to the CSR format but we compress the column indices instead and follow the column-major order. We store $n + 1$ values for column indices rather than η_z values. We denote M_{row}, M_{col}, M_{val} for the row indices array, compressed column indices array and values array respectively. We know that $|M_{col}| = n + 1$ while $|M_{row}| = |M_{val}| = \eta_z$. $V^{(t+1)}$ can be represented in the CSC format as folows:

Row index	0	1	3
Compressed col	3	4	5
Value	1.0	0.5	0.5

Then each non-zero element of the i-th column ($0 \leq i \leq n$) can be extracted by a tuple $M_{row}[j]$, i, $M_{val}[j]$ for $M_{col}[i] \leq j < M_{col}[i + 1]$.

Regarding memory usage among general-purpose sparse representations, the CSR and the CSC formats are similar in case the matrix is square and they are usually better than the COO format. The CSR format enables faster lookup by row while the CSC format provides faster lookup by column. In our previous work, we suggest

using the CSR for the program matrix then when transpose it to obtain an abductive matrix, it will become a CSC matrix naturally. Therefore, we suggest using the CSC format for both the abductive matrix and the explanation matrix.

In the case of an explanation matrix, we initialize it as a column matrix of the observation matrix then it grows to multiple columns while we exploring. Henceforth, the CSC format is an ideal format because it is faster in terms of looking up values by column. As we can see in the CSC representation of $V^{(t+1)}$ in the previous example, the CSC format stores both the row indices and values that we need to form the corresponding set of indices. If we use a dense matrix, we have to iterate over the column and check whether each value is larger than zero or not then extract its row index to the set of indices. Using the sparse format, which is the CSC concretely, we store all row indices of non-zero elements in an explanation vector then we can form a set of row indices at almost no cost. We use this set of row indices as input for solving the MHS as represented in the Algorithm 4. Moreover, representing the explanation matrix in the CSC format is compatible with the abductive matrix, which is in the CSC format as we mentioned before. So when we apply the dot product in Step 5 of Algorithm 4, we obtain output in the CSC format. Therefore, we suggest using the CSC format for the explanation matrix.

4.4 Experiments

4.4.1 Benchmark datasets

To demonstrate the linear algebraic computation of abduction, we conduct experiments on the same benchmark datasets used in [51, 52]. The benchmark problems are built based on Failure Modes and Effects Analysis (FMEA), which is a well-known critical fault identifying framework in industry, through a direct mapping described by Wotawa [94]. The benchmark consists of both real-world and artificial samples and all benchmark problems do not contain constraints. Based on the generation method, each problem set has different characteristic:

- Artificial samples I: deeper but narrower graph structure.
- Artificial samples II: deeper and wider graph structure, some problem involves solving a large number of medium-size MHS problems.
- FMEA samples: shallower but wider graph structure, usually involing a few (but) large-size MHS problems.

Table 4.1 further represents statistics data of the datasets in more detail.

As stated before, we need an extra step to transform the program into equivalent standardized format. Accordingly, we denote the input PHCAP as $\langle \mathbb{P}', \mathbb{H}, \mathbb{O}, \mathbb{T}' \rangle$ while

the standardized PHCAP is $\langle \mathscr{L}, \mathbb{H}, \mathbb{O}, \mathrm{P} \rangle$. The detail statistical information of all problem sets are represented in the first 4 rows $|\mathbb{H}|$, $|\mathbb{P}' \setminus \mathbb{H}|$, $|\mathbb{T}'|$, and $|\mathbb{O}|$ of Table 4.1. The next 4 rows of Table 4.1 represent the transformed P and $\mathscr{L}$, while $|\mathscr{L}|$ is also the dimension of a corresponding abductive matrix.

Table 4.1 also records the sparsity analysis data on all benchmark datasets. $\eta_z({M_P}^T)$ and $\eta_z(M)$ are the number of non-zero elements in the abductive matrix and the explanation matrix respectively. Similarly, $sparsity({M_P}^T)$ and $sparsity(M)$ are the sparsity of the abductive matrix and the explanation matrix, respectively. Because explanation matrices are not fixed, we record only the maximum number of explanation vectors ($max(|M|)$), the maximum η_z ($max(\eta_z(M))$), and the minimum sparsity $min(sparsity(M))$ for each explanation matrix. Finally, *max_iter* is the number of iterations of the main loop in the Algorithm 4 and $|\mathbb{E}|$ is the number of correct minimal explanations.

In order to demonstrate the performance of the linear algebraic approach using sparse representation, we propose a way to upscale the standard benchmark dataset. In such a problem instance of the dataset, we conjunct the corresponding dependency graph of the instance by itself multiple times and add an *And*-node as a new root of the graph. Additionally, we keep the observations set unchanged from only a single instance. By doing this way, we can upscale any problem instance in the standard dataset to a more complicated problem. In this experiment, we generate two more problem set accrodingly by doubling and quadrupling of each instance in the standard benchmark. The two new introduced benchmark datasets are called, upscaled 2× dataset and upscaled 4× dataset respectively. The two new benchmarks have the same characteristic as the original benchmark but with more nodes and are supposed to be more challenging for solvers.

4.4.2 Experimental setup

To demonstrate the linear algebraic computation of abduction, we conduct experiments on the benchmark datasets used in [51, 52]. The purpose of this thesis is to compare the effectiveness of our method with that of other general-purpose solvers. Thus, we implement our method as two versions including a dense matrix method (*Dense matrix* for short) and a sparse matrix method (*Sparse matrix* for short). For both the abductive matrix and the explanation matrix, we use dense format in *Dense matrix* while in *Sparse matrix*, CSC format is used.Our code is implemented in Python 3.7 using Numpy and Scipy for matrices representation and computation. As stated in Algorithm 4, we implement a naive approach for solving MHS that we only use built-in Python set operations. For large-size MHS problems, which have more than 50,000

combinations, we use MHS enumerator provided by PySAT. Importantly, we force to execute our code in a single core, in order to make a fair comparison with other methods. The computer we perform this experiment has the following configurations: CPU: Intel(R) Xeon(R) Bronze 3106 CPU @1.70GHz; RAM: 64GB DDR3 @1333MHz; Operating system: Ubuntu 18.04 LTS 64bit.

The other competitors are $ATMS$, ASP, CF, $HS\text{-}DAG$ and $HS\text{-}DAG_{QX}$ that we reuse the baseline Java source code in [51, 52][3]: $ATMS$ implementation is based on the assumption-based LTUR in the jdiagengine [69]; ASP implementation exploits an encoding of propositional abduction by [74] then calls Clingo solver [32], which is written in C++, for stable computation; CF implementation is based on the SOLAR [60]; $HS\text{-}DAG$ implementation is based on the jdiagengine [69]; $HS\text{-}DAG_{QX}$ is an improved implementation of $HS\text{-}DAG$ employing QuickXplain [49].

Similar to the experiment setup in [52], each method is conducted 10 times with a limited runtime on each PHCAP problem to record execution time and correctness of the output. The time limit for each run is 20 minutes that is if a solver cannot output the correct output in this limit, 40 minutes will be penalized to its execution time. Accordingly, we denote t as the effective solving time and t_p as the penalty time, consequently, $t + t_p$ is the total running time. The extra time for transforming to the standardized format of our methods is included in t.

As stated before, we need an extra step to transform the program into equivalent standardized format. Accordingly, we denote the input PHCAP as $\langle \mathbb{P}', \mathbb{H}, \mathbb{O}, \mathbb{T}' \rangle$ while the standardized PHCAP is $\langle \mathscr{L}, \mathbb{H}, \mathbb{O}, \mathrm{P} \rangle$. The detailed statistical information of all problem sets are represented in the first 4 rows $|\mathbb{H}|$, $|\mathbb{P}' \setminus \mathbb{H}|$, $|\mathbb{T}'|$, and $|\mathbb{O}|$ of Table 4.1. The next 4 rows of Table 4.1 represent the transformed P and $\mathscr{L}$, while $|\mathscr{L}|$ is also the dimension of a corresponding abductive matrix.

Table 4.1 also records the sparsity analysis data on all benchmark datasets. $\eta_z({M_P}^T)$ and $\eta_z(M)$ are the number of non-zero elements in the abductive matrix and the explanation matrix respectively. Similarly, $sparsity({M_P}^T)$ and $sparsity(M)$ are the sparsity of the abductive matrix and the explanation matrix, respectively. Because explanation matrices are not fixed, we record only the maximum number of explanation vectors ($max(|M|)$), the maximum η_z ($max(\eta_z(M))$), and the minimum sparsity $min(sparsity(M))$ for each explanation matrix. Finally, max_iter is the number of iterations of the main loop in the Algorithm 4 and $|\mathbb{E}|$ is the number of correct minimal explanations.

In order to visualize the results, for each problem set, we illustrate using a line

[3]Readers should follow the paper for more detail about other algorithms. Among the reported algorithms, we omit the $Explorer$ and $Explorer_{QX}$ because they fall behind other algorithms by a large margin.

graph for convergence trend comparison, a bar graph displaying $t + t_p$ (full height) and t (height of solid color), a table for more detail in numbers. For better visualization, all figures and data are illustrated in a log-scale of runtime in millisecond (ms), and in all figures for runtime trends, we order successful runs by their execution time before plotting the data.

4.4.3 Results for The Standard Dataset

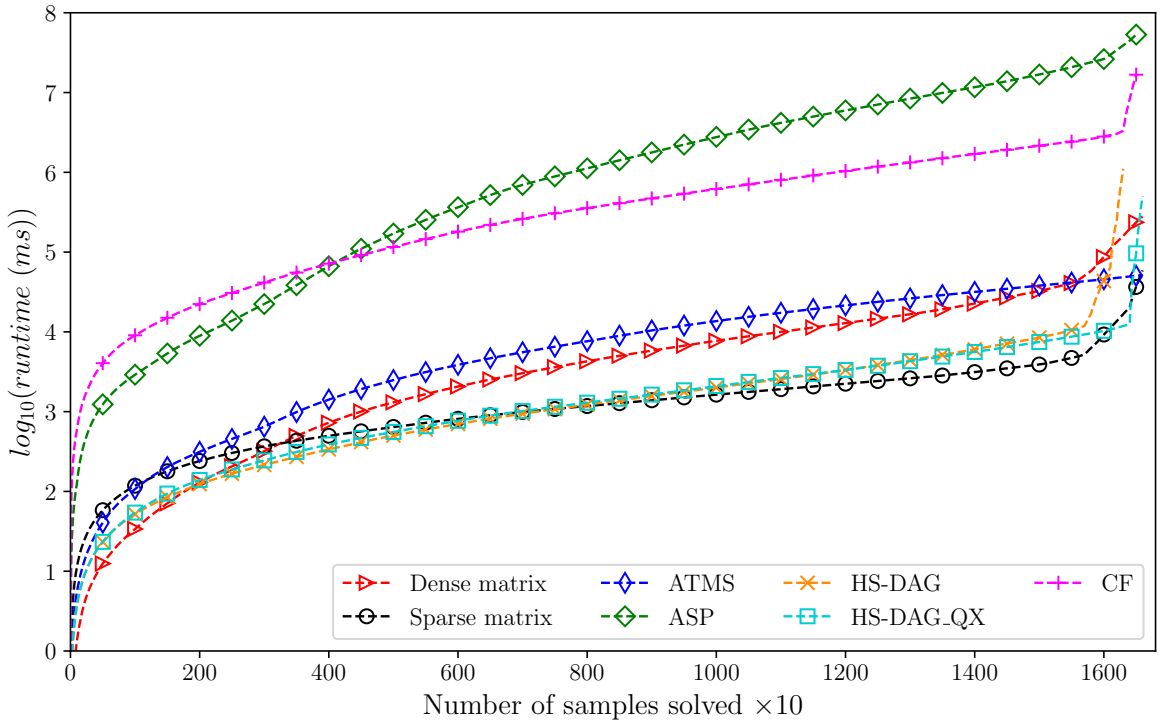
Artificial benchmarks

Figure 4.2 and Figure 4.3 illustrate the comparison on the Artificial samples I and II, while Table 4.2 and Table 4.3 give more detail information. As witnessed in Figure 4.2 and Figure 4.3, runtime trends of all algorithms grow exponentially by the number of solved samples (**#solved**).

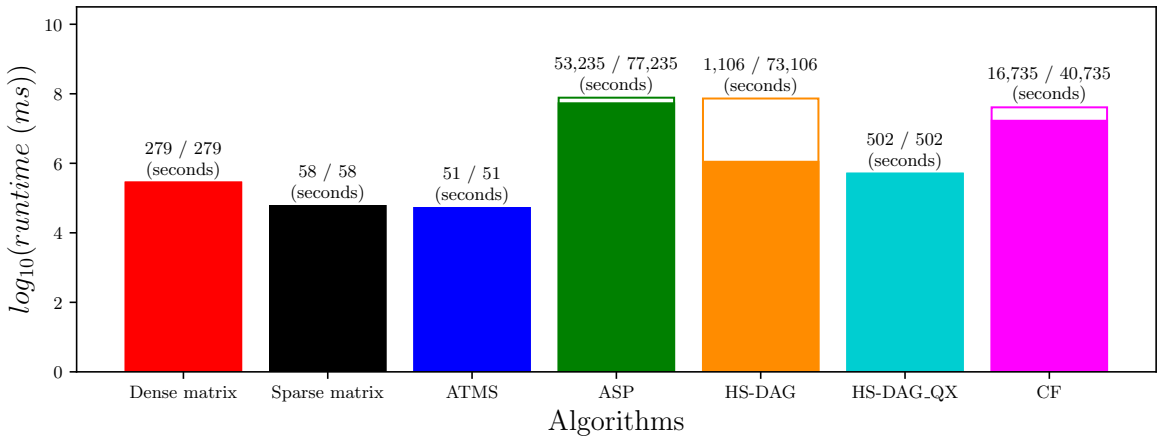
In the Artificial samples I, together with $ATMS$ and $HS\text{-}DAG_{QX}$, our linear algebraic approaches are able to solve all problems. Surprisingly, in terms of total runtime, *Dense matrix* is even faster than $HS\text{-}DAG_{QX}$ while *Sparse matrix* is just a few seconds behind the fastest - $ATMS$. Other methods fall behind by 3a large margin because they are penalized on unresolved samples. Table 4.2 further reveals the efficiency of linear algebraic methods that *Dense matrix* is the fastest in 110 runs while *Sparse matrix* is the fastest in 930 runs. In this dataset, the sparsity of abductive matrices and explanation matrices maintains at a good level of **mean** (Table 4.1).

Benchmark dataset	**Artificial samples I** (166 problems)				**Artificial samples II** (118 problems)				**FMEA samples** (213 problems)			
Parameters	**mean**	**std**	**min**	**max**	**mean**	**std**	**min**	**max**	**mean**	**std**	**min**	**max**
$\|\mathbb{H}\|$	275.07	167.12	10	504	120.42	74.35	12	235	26.16	20.81	3	90
$\|\mathscr{L}' \setminus \mathbb{H}\|$	1903.23	1504.9	6	6466	252.74	220.5	13	1055	27.58	19.32	6	84
$\|P'\|$	2951.1	2131.57	11	7187	417.7	320.56	21	1147	71.59	75.88	13	299
$\|\mathbb{O}\|$	2.86	1.38	1	5	2.72	1.71	1	13	10.79	6.94	1	29
$\|P\|$	2088.32	1584.48	11	6601	321.86	252.64	18	1110	27.58	19.32	6	84
$\|P_{And}\|$	1188.63	1349.59	8	6375	201.86	186.64	9	1007	16.1	9.23	1	43
$\|P_{Or}\|$	899.69	839.58	3	3345	119.99	107.4	4	437	11.48	11.01	1	41
$\|\mathscr{L}\|$	2372.36	1730.91	24	7148	450.89	318.33	38	1397	53.74	39.59	9	174
$\eta_z({M_P}^T)$	6354.9	4902.87	50	22,307	1180.36	861.83	83	4117	107.54	98.57	18	413
$sparsity({M_P}^T)$	0.99	0.02	0.9	1	0.99	0.01	0.9	1	0.95	0.04	0.73	0.99
$max(\|M\|)$	250.34	1729.52	1	16,866	16,494.04	149,787.13	1	1,618,050	2126.49	15,512.54	1	154,440
$max(\eta_z(M))$	5138.28	37,776.87	1	428,754	390,900.36	3,240,888.43	1	34,882,765	43,738.87	334,393.4	1	3,459,456
$min(sparsity(M))$	0.98	0.05	0.68	1	0.94	0.08	0.59	1	0.79	0.13	0.46	0.99
max_iter	4.63	5.36	2	65	6.56	8.56	2	58	1.94	0.24	1	2
$\|\mathbb{E}\|$	2.77	5.06	1	50	499.6	5386.87	1	58,520	68.89	272.54	1	2288

Table 4.1: Statistics and sparsity analysis on benchmark datasets



(a) Effective runtime by number of solved samples.



(b) Total runtime (10 runs) on the whole dataset (166 files).

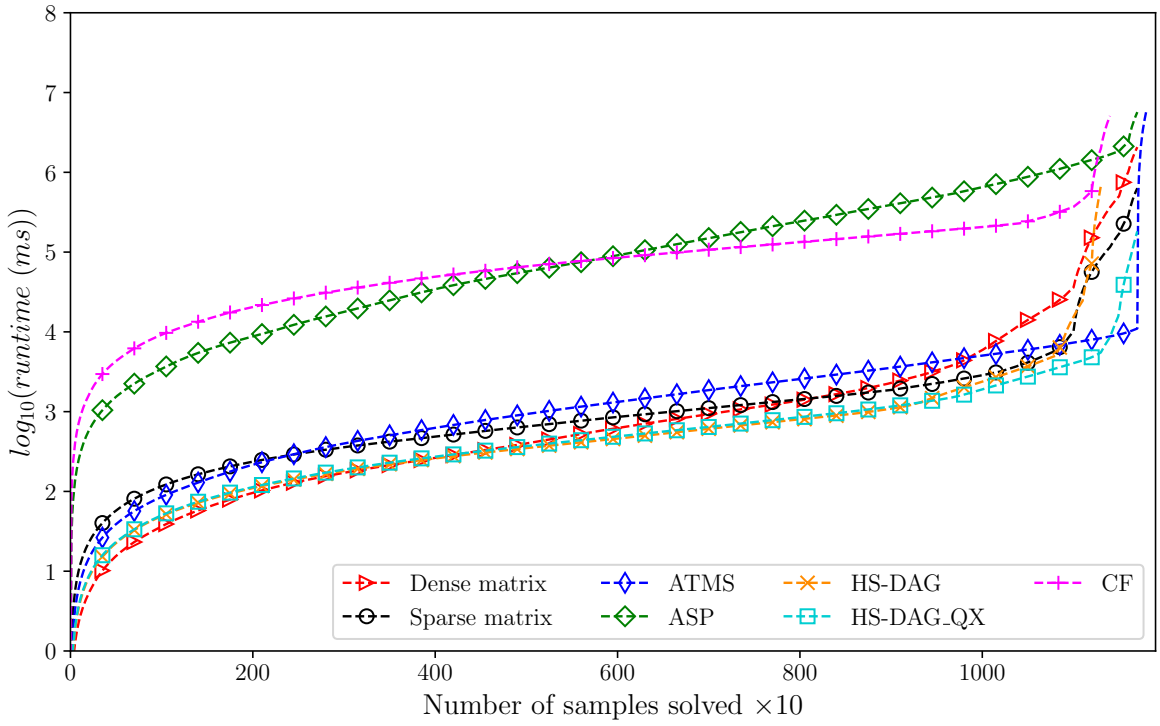
Figure 4.2: Experimental results for the Artificial samples I.

Algorithms	**#solved**	**#fastest**	**mean**(t) (ms)	**std**(t) (ms)	**mean**($t + t_p$) (ms)	**std**(t+t_p) (ms)
Dense matrix	1660	110	27,902	335	27,902	1744
Sparse matrix	1660	930	5900	45	5900	95
ATMS	1660	68	5171	154	5171	1089
ASP	1650	0	5,323,586	103,734	7,723,586	317,719
HS-DAG	1630	344	110,640	30,280	7,310,640	33,035
HS-DAG_QX	1660	208	50,230	9766	50,230	12,389
CF	1650	0	1,673,516	59,781	4,073,516	91,042

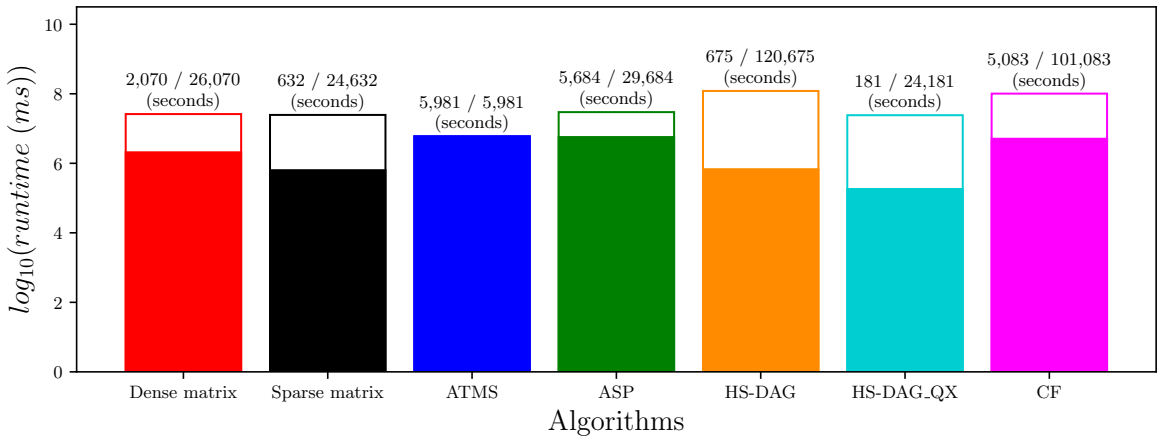
Table 4.2: Detail runtime results on the Artificial samples I.

In the Artificial samples II, only $ATMS$ is able to handle all problems although it is not the fastest algorithm. ASP, $HS\text{-}DAG_{QX}$ and linear algebraic methods are equal in terms of **#solved** that is 117/118. Table 4.3 gives a more detailed comparison in the Artificial samples II that *Dense matrix* and *Sparse matrix* are competitive as being the fastest algorithm in 248 and 120 runs, respectively. From Table 4.1 we also can see that $|\mathbb{E}|$ and $max(|M|)$ surge to very large figures, $58,520$ and $1,618,050$, respectively. This happens in the only one problem instance that our methods are failed to solve in time.

Notably in both the benchmarks, *Dense matrix* takes the lead over *Sparse matrix* in the beginning. This is understandable because the sparsity level of explanation matrices, for example in the data for Artificial samples II in Table 4.1, drops to **min** 0.59 and **mean** 0.94. In this situation, sparse representation cannot take much benefit. Due to that fact, *Sparse matrix* still takes over *Dense matrix* in the end with much better total execution time as can be seen in Figure 4.2. Further, *Sparse matrix* is the most stable algorithm with the best **std**.



(a) Effective runtime by number of solved samples.



(b) Total runtime (10 runs) on the whole dataset (118 files).

Figure 4.3: Experimental results for the Artificial samples II.

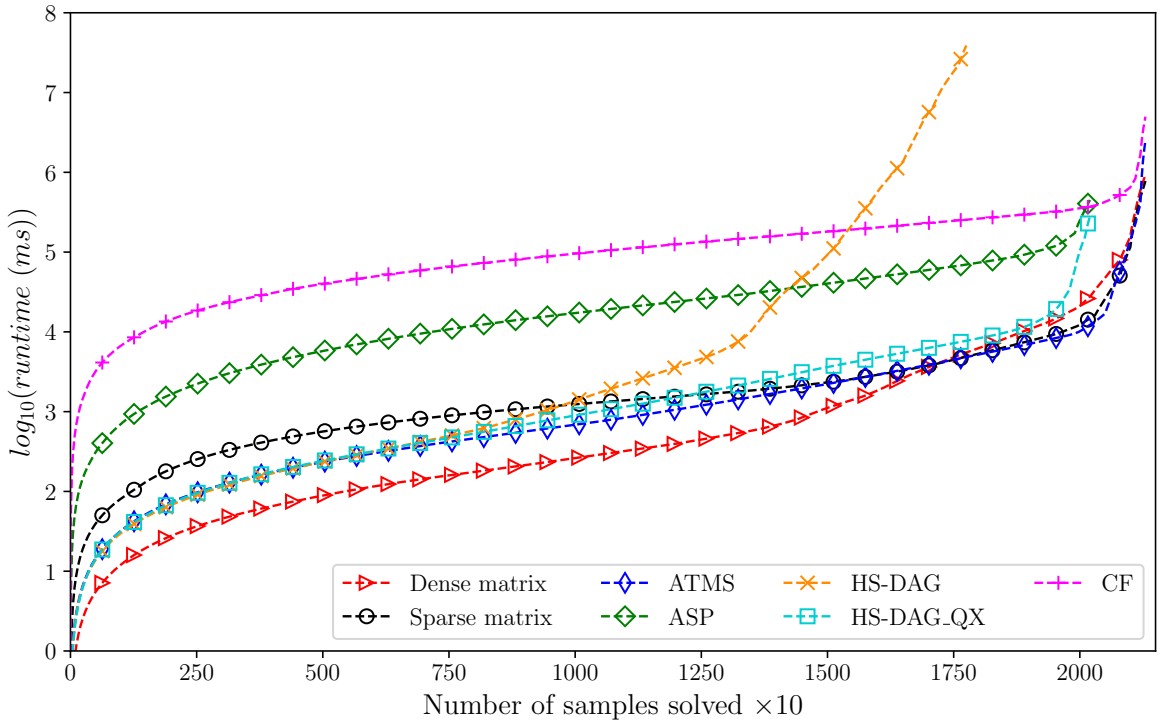
Algorithms	#solved	#fastest	mean(t) (ms)	std(t) (ms)	mean($t + t_p$) (ms)	std(t+t_p) (ms)
Dense matrix	1170	248	207,015	3573	2,607,015	6906
Sparse matrix	1170	120	63,251	235	2,463,251	596
ATMS	1180	119	598,145	63,317	598,145	67,146
ASP	1170	0	568,408	2868	2,968,407	12,196
HS-DAG	1130	436	67,567	16,943	12,067,567	18,572
HS-DAG_QX	1170	257	18,198	4106	2,418,198	6745
CF	1140	0	508,309	7850	10,108,309	13,188

Table 4.3: Detail runtime results on the Artificial samples II.

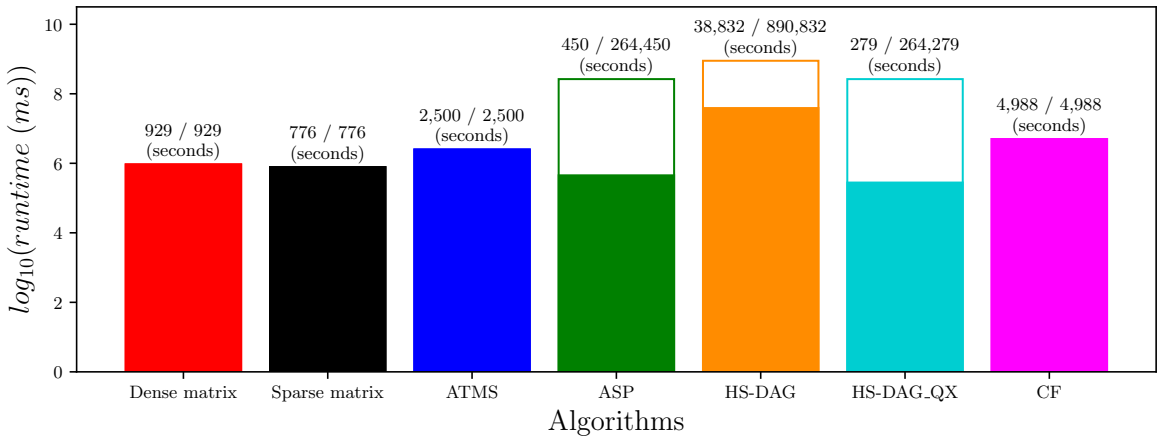
Real-world samples

Figure 4.4 illustrates the comparison on FMEA samples benchmark while Table 4.4 gives more detail information about each algorithm. In this benchmark, *ATMS*, *CF* and linear algebraic methods are able to solve all instances without penalty. Surprisingly, *Dense matrix* outperforms others and takes the lead by a remarkable margin (Figure 4.4) and ends up even more than 2 times faster than the 3rd place algorithm - *ATMS* in terms of total execution time (Figure 4.4). *Sparse matrix* starts with a humble beginning but performs very well after that and finishes at the first place with the lowest execution time in total.

From Table 4.1, we can see that $sparsity({M_P}^T)$ and $sparsity(M)$ drop to **mean** 0.95, **min** 0.73 and **mean** 0.79, **min** 0.46, respectively. That is the reason for the good performance of *Dense matrix* in many runs. Despite of that fact, *Sparse matrix* is still better in overall because of faster lookup by column as explained in Section 4.1. Moreover, *Sparse matrix* still is the best stable algorithm with the lowest **std** among those with highest **#solved**.



(a) Effective runtime by number of solved samples.



(b) Total runtime (10 runs) on the whole dataset (213 files).

Figure 4.4: Experimental results for the FMEA diagnosis problems.

Algorithms	**#solved**	**#fastest**	**mean**(t) (ms)	**std**(t) (ms)	**mean**($t + t_p$) (ms)	**std**(t+t_p) (ms)
Dense matrix	2130	1166	92,985	1548	92,985	3558
Sparse matrix	2130	160	77,685	886	77,685	1424
ATMS	2130	579	250,000	16,202	250,000	23,799
ASP	2020	0	45,051	763	26,445,051	2799
HS-DAG	1775	31	3,883,276	650,599	89,083,276	950,627
HS-DAG_QX	2020	184	27,926	350	26,427,926	1492
CF	2130	10	498,885	16,325	498,885	25,601

Table 4.4: Detail runtime results on the FMEA diagnosis problems.

4.4.4 Results for the Upscaled 2× Dataset

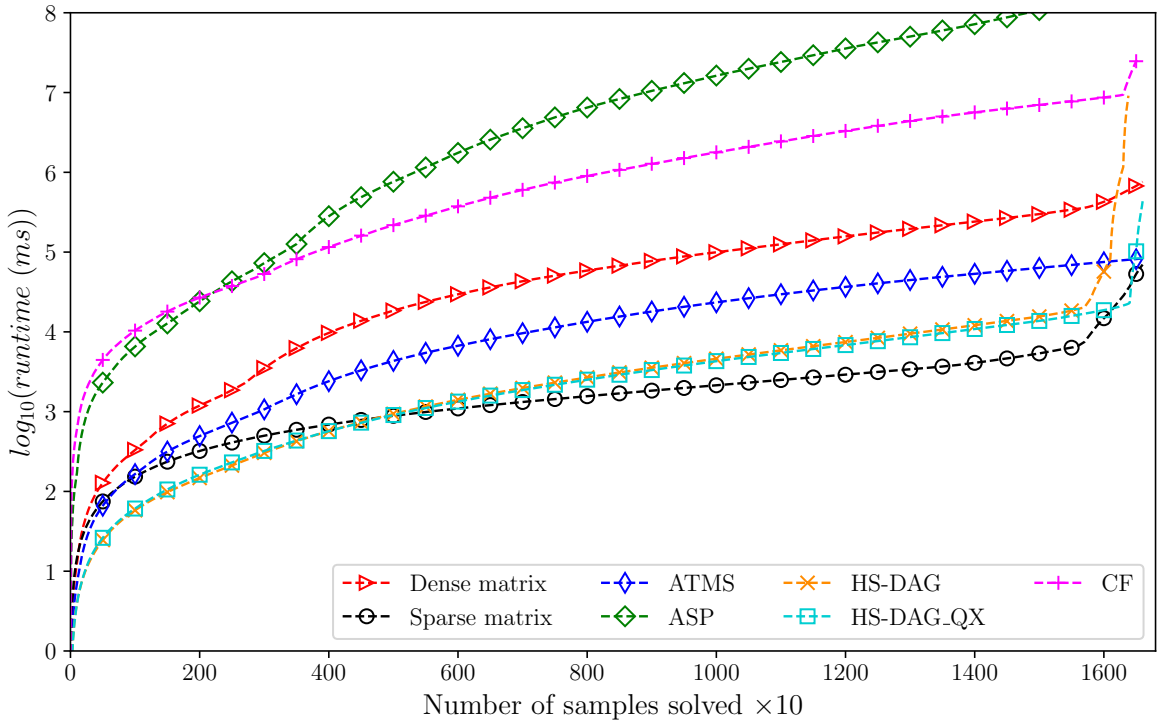
Figure 4.5, Figure 4.6 and Figure 4.7 demonstrate comparison between algorithms while further sparsity analysis is reported in Table 4.5.

Artificial benchmarks

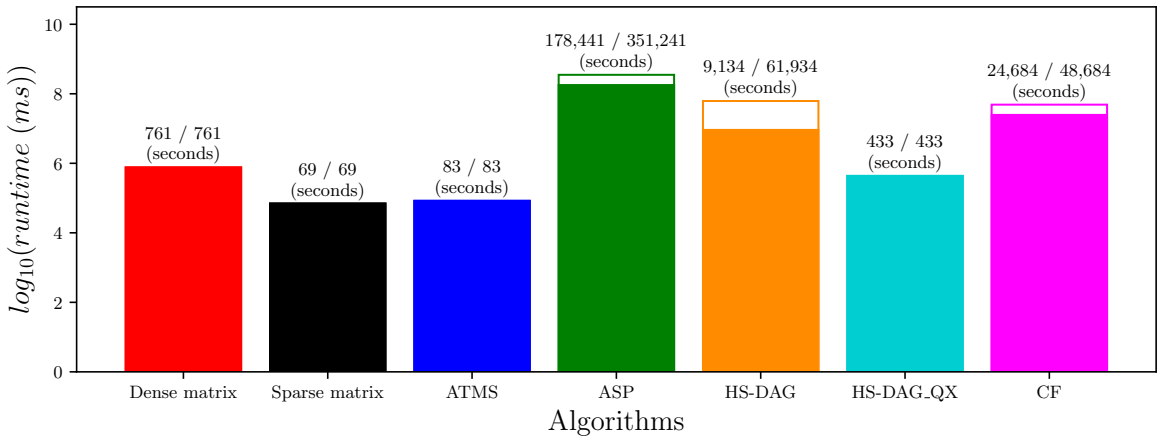
In the Artificial samples I, *Sparse matrix* is able to solve all instances again and this time it takes the lead with 1089 **#fastest** and 22 **#std**. $ATMS$ now comes at second place with, despite the fact that it is only being fastest if 54 instances, less than HS-DAG and $HS\text{-}DAG_{QX}$. The matrix size increasing is really a matter for *Dense matrix* that it fails to keep up with $HS\text{-}DAG_{QX}$.

Benchmark dataset	**Artificial samples I** (166 problems)				**Artificial samples II** (118 problems)				**FMEA samples** (213 problems)			
Parameters	**mean**	**std**	**min**	**max**	**mean**	**std**	**min**	**max**	**mean**	**std**	**min**	**max**
$\|\mathbb{H}\|$	550.13	334.25	20	1008	240.85	148.71	24	470	52.32	41.62	6	180
$\|\mathscr{L}' \setminus \mathbb{H}\|$	3807.47	3009.79	13	12,933	506.47	440.99	27	2111	54.16	38.64	11	167
$\|P'\|$	5902.2	4263.13	22	14,374	835.41	641.12	42	2294	143.17	151.77	26	598
$\|\mathbb{O}\|$	1	0	1	1	1	0	1	1	1	0	1	1
$\|P\|$	4176.64	3168.95	22	13,202	643.71	505.28	36	2220	55.16	38.64	12	168
$\|P_{And}\|$	2377.25	2699.18	16	12,750	403.73	373.29	18	2014	32.2	18.47	2	86
$\|P_{Or}\|$	1799.39	1679.16	6	6690	239.98	214.8	8	874	22.97	22.02	2	82
$\|\mathscr{L}\|$	4744.72	3461.82	48	14,296	901.78	636.66	76	2794	107.48	79.19	18	348
$\eta_z({M_P}^T)$	12,707.95	9805.91	100	44,613	2358.99	1723.72	162	8231	205.29	194.44	36	802
$sparsity({M_P}^T)$	1	0.01	0.95	1	0.99	0.01	0.95	1	0.98	0.02	0.88	0.99
$max(\|M\|)$	230.66	1605.54	1	16,866	2646.09	18,060	1	188,921	2126.49	15,512.54	1	154,440
$max(\eta_z(M))$	4725.49	36,392.55	1	428,754	80,256.09	464,108.84	1	4,638,576	43,738.87	334,393.4	1	3,459,456
$min(sparsity(M))$	0.99	0.03	0.83	1	0.97	0.05	0.76	1	0.9	0.06	0.73	1
max_iter	5.63	5.36	3	66	7.81	10.69	3	92	2.94	0.24	2	3
$\|\mathbb{E}\|$	2.77	5.06	1	50	3.67	9.58	0	63	68.89	272.54	1	2288

Table 4.5: Statistics and sparsity analysis on benchmark datasets 2× upscaled



(a) Effective runtime by number of solved samples.



(b) Total runtime (10 runs) on the whole dataset.

Figure 4.5: Experimental results for the 2× upscaled Artificial samples I (166 files).

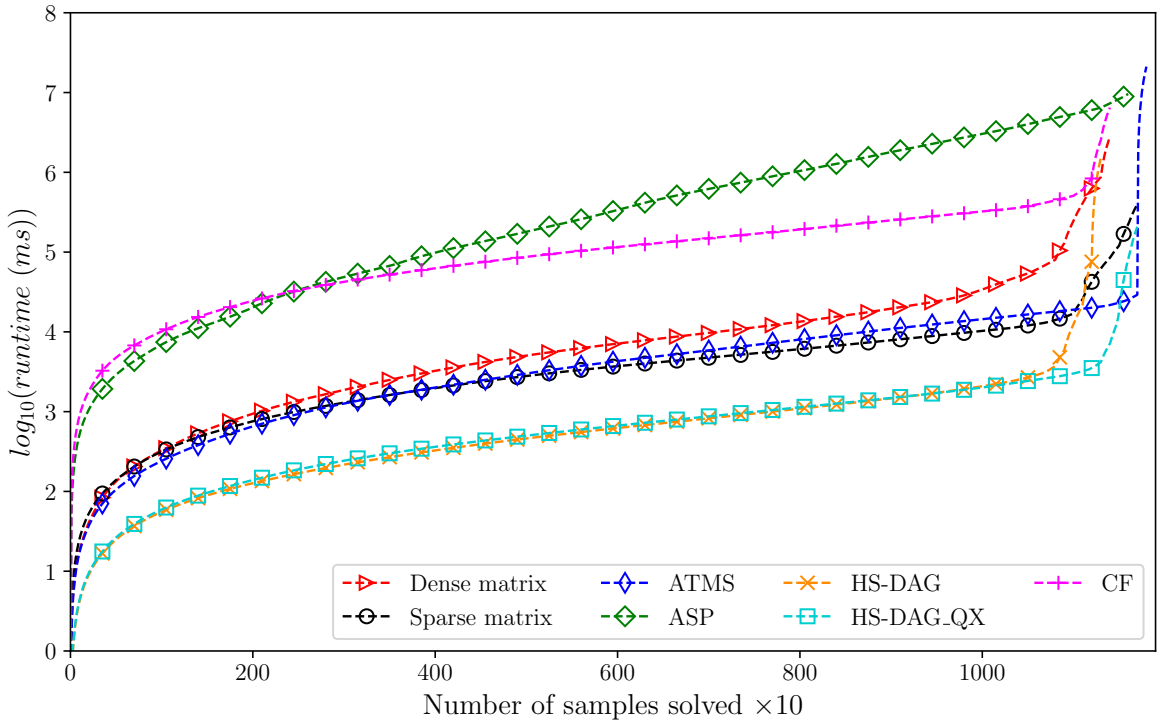
Algorithms	**#solved**	**#fastest**	**mean**(t) (ms)	**std**(t) (ms)	**mean**($t + t_p$) (ms)	**std**(t+t_p) (ms)
Dense matrix	1660	0	76,121	372	76,121	2593
Sparse matrix	1660	1089	6974	22	6974	91
ATMS	1660	54	8324	171	8324	1476
ASP	1588	0	17,844,157	523,062	35,124,157	1,845,769
HS-DAG	1638	220	913,448	400,588	6,193,448	602,237
HS-DAG_QX	1660	297	43,389	8692	43,389	10,967
CF	1650	0	2,468,475	29,531	4,868,475	70,697

Table 4.6: Detail runtime results on the 2× upscaled Artificial samples I.

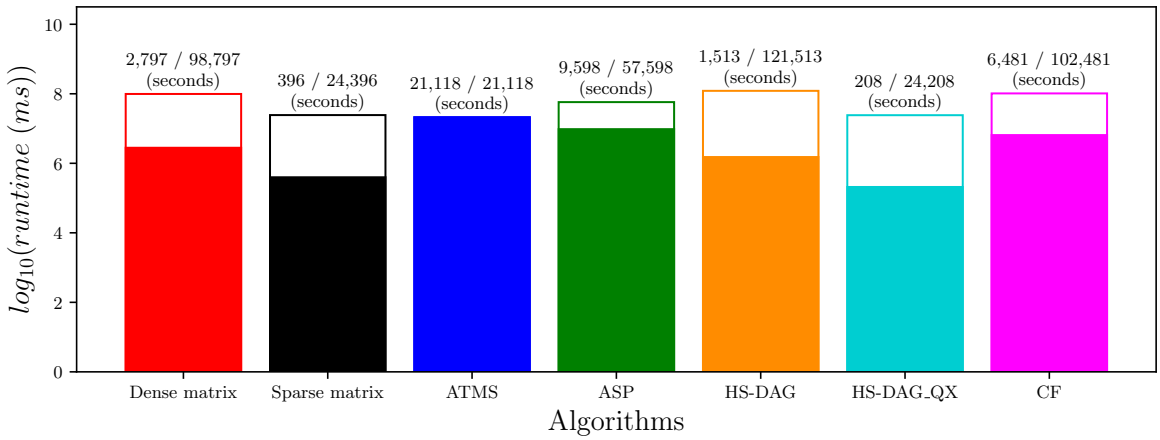
Algorithms	**#solved**	**#fastest**	**mean**(t) (ms)	**std**(t) (ms)	**mean**($t + t_p$) (ms)	**std**(t+t_p) (ms)
Dense matrix	1140	1	279,704	3011	9,879,704	5462
Sparse matrix	1170	8	39,615	157	2,439,615	411
ATMS	1180	65	2,111,843	114,332	2,111,843	120,907
ASP	1160	0	959,808	5894	5,759,808	16,229
HS-DAG	1130	620	151,362	8685	12,151,362	9770
HS-DAG_QX	1170	486	20,831	4173	2,420,831	8093
CF	1140	0	648,167	3356	10,248,167	10,840

Table 4.7: Detail runtime results on the 2× upscaled Artificial samples II.

In the Artificial samples II, one more time, only $ATMS$ is able to handle all problem instances although it takes much more execution time than before. $HS\text{-}DAG_{QX}$ still holds the second place but the gap is shortened by *Sparse matrix*. *Dense matrix* this time is failed to solve 4 instances but is still competitive with other algorithms.



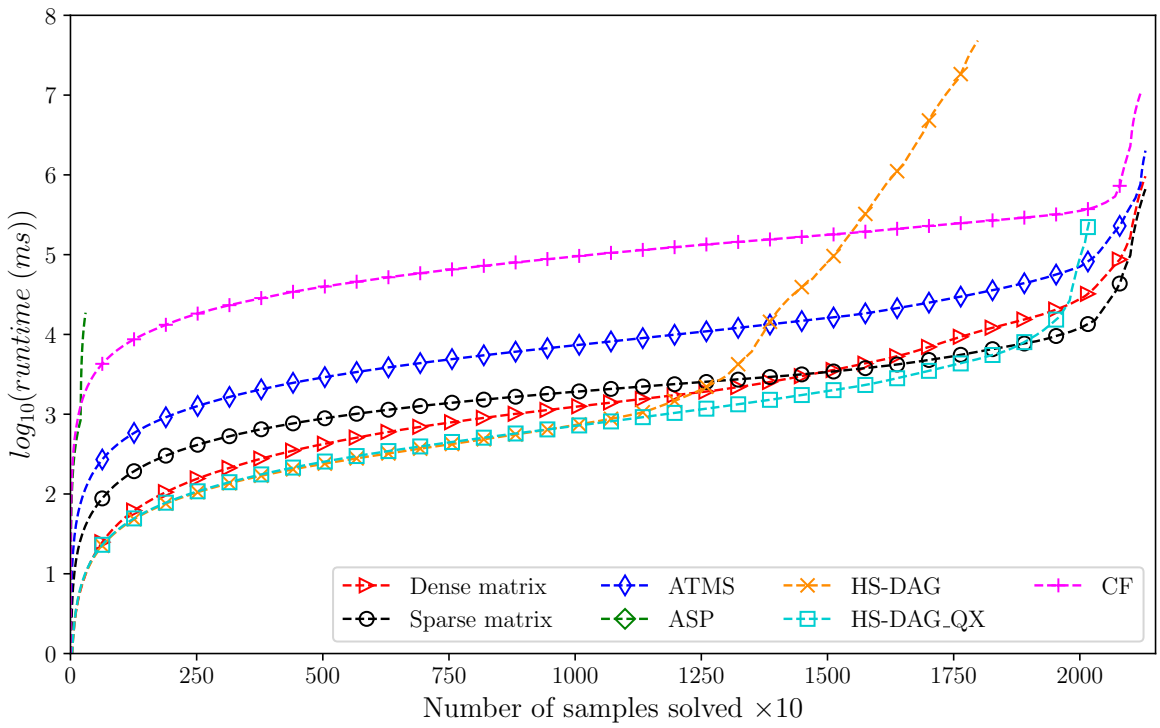
(a) Effective runtime by number of solved samples.



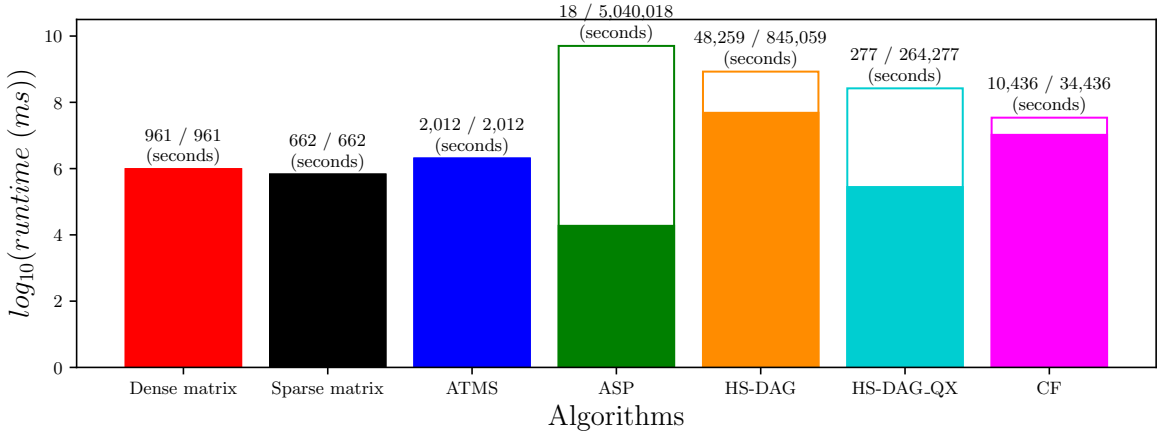
(b) Total runtime (10 runs) on the whole dataset.

Figure 4.6: Experimental results for the 2× upscaled Artificial samples II (118 files).

Real-world samples



(a) Effective runtime by number of solved samples.



(b) Total runtime (10 runs) on the whole dataset.

Figure 4.7: Experimental results for the 2× upscaled FMEA samples (213 files).

Algorithms	#solved	#fastest	mean(t) (ms)	std(t) (ms)	mean($t + t_p$) (ms)	std(t+t_p) (ms)
Dense matrix	2130	191	96,193	1108	96,193	2070
Sparse matrix	2130	440	66,277	245	66,277	477
ATMS	2130	20	201,249	3479	201,249	9231
ASP	30	0	1873	10	510,480,187	2,170,317
HS-DAG	1798	618	4,825,902	427,943	84,505,902	986,115
HS-DAG_QX	2020	861	27,747	241	26,427,747	903
CF	2120	0	1,043,638	13,228	3,443,638	23,337

Table 4.8: Detail runtime results on the 2× upscaled FMEA diagnosis problems.

In this benchmark, *ATMS* finishes slightly faster than itself in the standard benchmark. However, linear algebraic approaches are successful to defend their places. *HS-DAG* and *HS-DAG*$_{QX}$ are the fastest in many instances with 618 and 861 **#fastest** respectively, but they are unable to solve all problems. This time *ASP* shows a substandard performance with only 3 solvable instances.

4.4.5 Results for the Upscaled 4× Dataset

Figure 4.8, Figure 4.9 and Figure 4.10 demonstrate comparison between algorithms while further sparsity analysis is reported in Table 4.9.

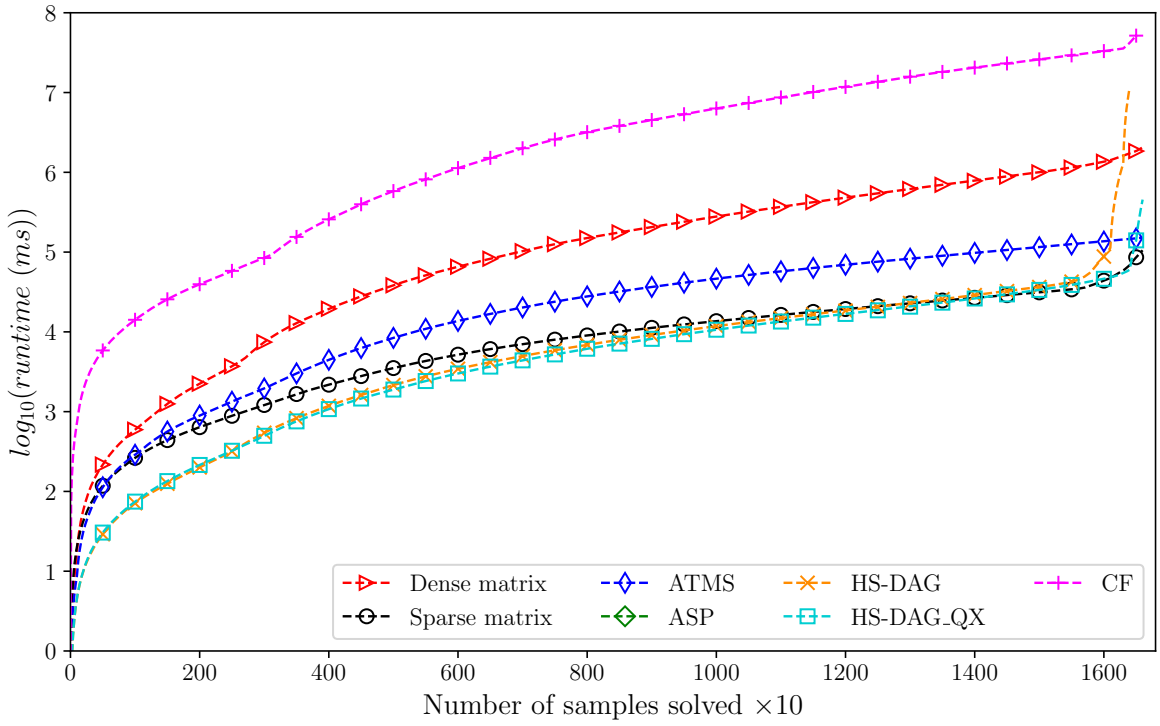
Artificial benchmarks

In the Artificial samples I, *Sparse matrix* is able to solve all instances again and this time it takes the lead with 1089 **#fastest** and 22 **#std**. *ATMS* now comes at second place with, despite the fact that it is being fastest in only 47 instances, less than *HS-DAG* and *HS-DAG*$_{QX}$. The increasing size of the set of literals causes a significant impact on the execution of all algorithms, especially for *Dense matrix* that it fails to keep up with *HS-DAG*$_{QX}$. *Sparse matrix* comes again as the fastest and the most stable solver in this benchmark.

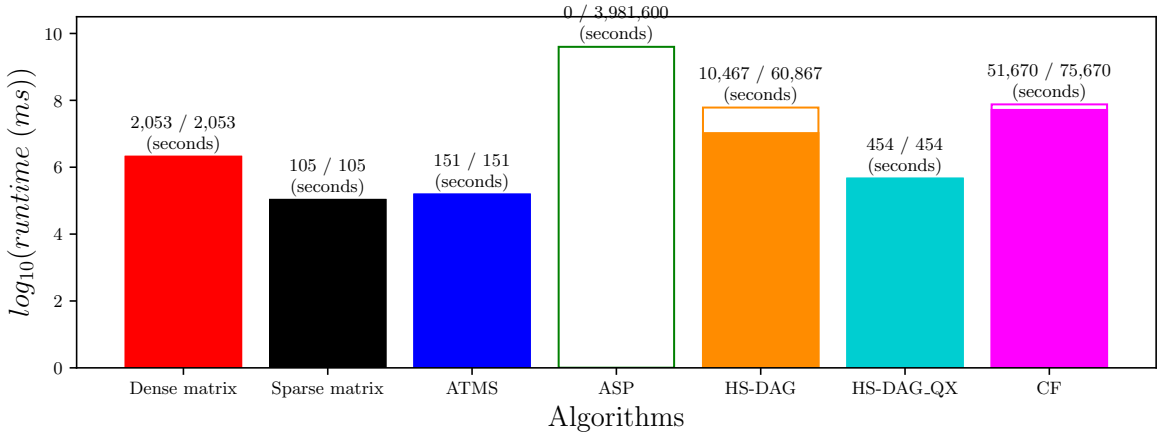
In the Artificial samples II, it is the first time that *ATMS* is failed to solve the hardest problem, so there is no solver is able to do that. Accordingly, *Sparse matrix*, *ATMS* and *HS-DAG*$_{QX}$ are the top three with highest **#solved**. The ranking has been updated that *HS-DAG*$_{QX}$ is the fastest with 526 **#solved**, *ATMS* falls to second place and *Sparse matrix* remains at the third place. *Dense matrix* this time is failed to solve 4 instances but is still competitive with other algorithms.

Benchmark dataset	**Artificial samples I** (166 problems)				**Artificial samples II** (118 problems)				**FMEA samples** (213 problems)			
Parameters	**mean**	**std**	**min**	**max**	**mean**	**std**	**min**	**max**	**mean**	**std**	**min**	**max**
$\lvert\mathbb{H}\rvert$	1100.27	668.49	40	2016	481.69	297.42	48	940	104.64	83.25	12	360
$\lvert\mathscr{L}' \setminus \mathbb{H}\rvert$	7615.94	6019.58	27	25,867	1013.95	881.99	55	4223	108.33	77.28	22	334
$\lvert P'\rvert$	11,804.41	8526.27	44	28,748	1670.81	1282.24	84	4588	286.35	303.53	52	1196
$\lvert\mathbb{O}\rvert$	1	0	1	1	1	0	1	1	1	0	1	1
$\lvert P\rvert$	8353.28	6337.91	44	26,404	1287.42	1010.56	72	4440	110.27	77.29	23	336
$\lvert P_{And}\rvert$	4754.51	5398.37	32	25,500	807.46	746.57	36	4028	63.91	37.02	3	172
$\lvert P_{Or}\rvert$	3598.77	3358.32	12	13,380	479.97	429.61	16	1748	46.36	43.99	4	164
$\lvert\mathscr{L}\rvert$	9489.45	6923.64	96	28,592	1803.56	1273.31	152	5588	214.91	158.38	35	696
$\eta_z({M_P}^T)$	25,415.9	19,611.81	200	89,226	4717.98	3447.43	324	16,462	411.52	388.89	72	1605
$sparsity({M_P}^T)$	1	0	0.98	1	1	0	0.98	1	0.99	0.01	0.94	1
$max(\lvert M\rvert)$	230.66	1605.54	1	16,866	2646.09	18,060	1	188,921	2126.49	15,512.54	1	154,440
$max(\eta_z(M))$	4725.49	36,392.55	1	428,754	80,256.09	464,108.84	1	4,638,576	43,738.87	334,393.4	1	3,459,456
$min(sparsity(M))$	0.99	0.01	0.92	1	0.98	0.02	0.88	1	0.95	0.03	0.87	1
max_iter	6.63	5.36	4	67	8.81	10.69	4	93	3.88	0.41	2	4
$\lvert\mathbb{E}\rvert$	2.77	5.06	1	50	3.67	9.58	0	63	71.81	273.06	2	2295

Table 4.9: Statistics and sparsity analysis on benchmark datasets 4× upscaled



(a) Effective runtime by number of solved samples.



(b) Total runtime (10 runs) on the whole dataset.

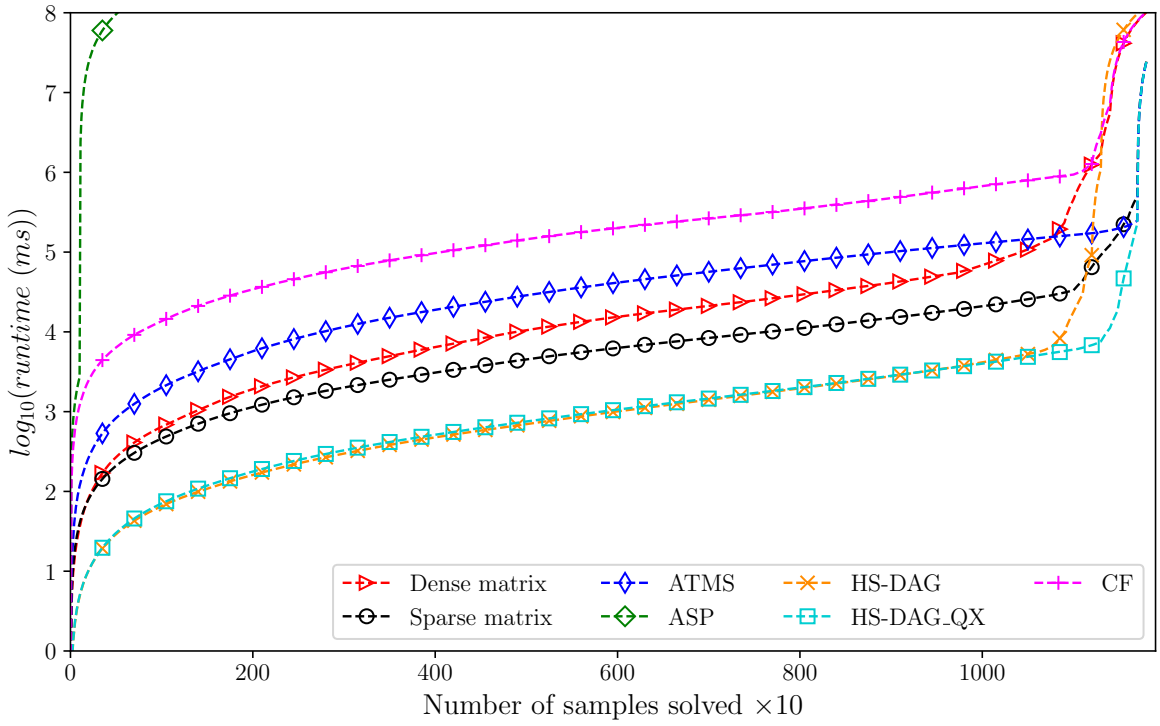
Figure 4.8: Experimental results for the 4× upscaled Artificial samples I (166 files).

Algorithms	**#solved**	**#fastest**	**mean**(t) (ms)	**std**(t) (ms)	**mean**($t + t_p$) (ms)	**std**(t+t_p) (ms)
Dense matrix	1660	0	205,379	521	205,379	4339
Sparse matrix	1660	534	10,535	188	10,535	742
ATMS	1660	47	15,192	295	15,192	2844
ASP	0	0	0	0	398,400,000	0
HS-DAG	1639	519	1,046,773	310,786	6,086,773	447,425
HS-DAG_QX	1660	560	45,494	4724	45,494	7863
CF	1650	0	5,167,075	19,810	7,567,075	131,840

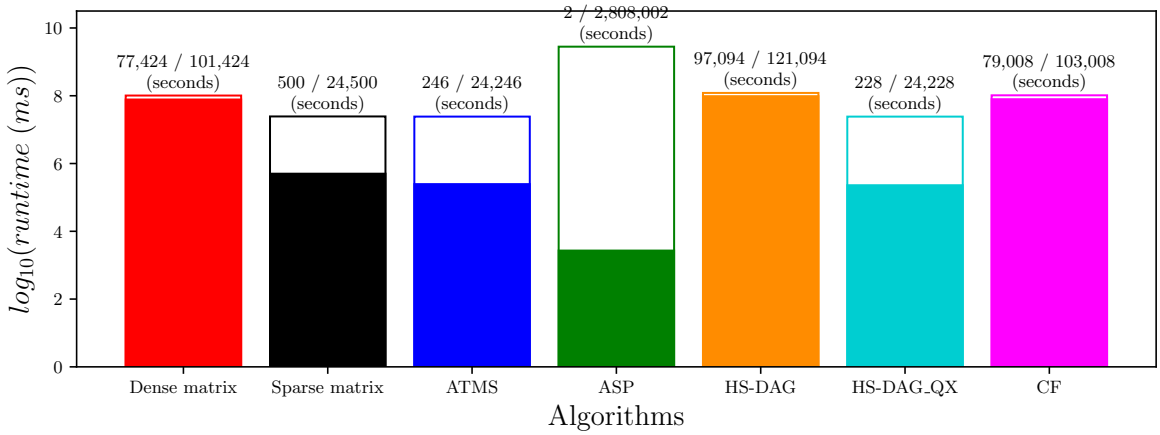
Table 4.10: Detail runtime results on the 4× upscaled Artificial samples I.

Algorithms	**#solved**	**#fastest**	**mean**(t) (ms)	**std**(t) (ms)	**mean**($t + t_p$) (ms)	**std**(t+t_p) (ms)
Dense matrix	1140	0	542,489	7954	10,142,489	12,529
Sparse matrix	1170	18	50,081	133	2,450,081	320
ATMS	1170	32	24,671	383	2,424,671	2257
ASP	10	0	270	4	280,800,270	4
HS-DAG	1130	584	109,442	4607	12,109,442	5287
HS-DAG_QX	1170	526	22,866	7125	2,422,866	10,960
CF	1140	0	700,890	3966	10,300,890	10,614

Table 4.11: Detail runtime results on the 4× upscaled Artificial samples II.



(a) Effective runtime by number of solved samples.

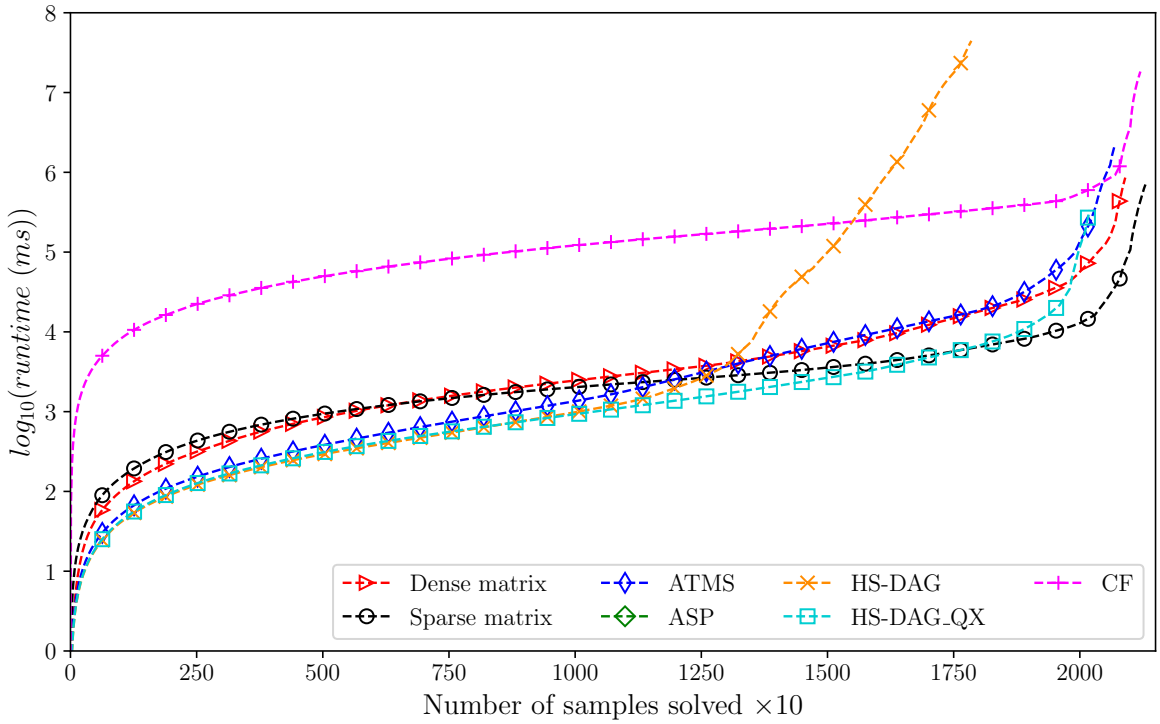


(b) Total runtime (10 runs) on the whole dataset.

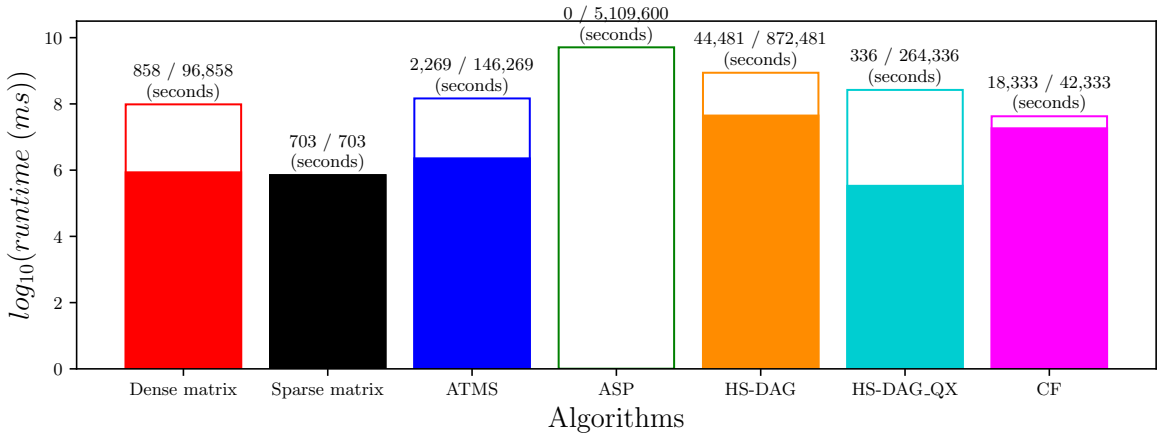
Figure 4.9: Experimental results for the 4× upscaled Artificial samples II (118 files).

Real-world samples

In the upscaled 4× real-world benchmark, it is clear that *Sparse matrix* is an absolute winner that it can manage all problem instances before timing out with outstanding execution time. *HS-DAG* and $HS\text{-}DAG_{QX}$ are the fastest in many instances with 480 and 642 **#fastest** respectively, but they are unable to solve all problems. This time *ASP* shows a substandard performance with no solvable instances. Surprisingly, *CF* demonstrates a stable performance while other algorithms have dropped significantly.



(a) Effective runtime by number of solved samples.



(b) Total runtime (10 runs) on the whole dataset.

Figure 4.10: Experimental results for the 4× upscaled FMEA diagnosis problems (213 files).

Algorithms	#solved	#fastest	mean(t) (ms)	std(t) (ms)	mean($t + t_p$) (ms)	std(t+t_p) (ms)
Dense matrix	2090	3	85,842	1784	9,685,842	2856
Sparse matrix	2130	521	70,327	210	70,327	538
ATMS	2070	484	226,953	4950	14,626,953	15,864
ASP	0	0	0	0	511,200,000	0
HS-DAG	1785	480	4,448,106	570,420	87,248,106	1,367,565
HS-DAG_QX	2020	642	33,604	487	26,433,604	1275
CF	2120	0	1,833,364	17,105	4,233,364	35,222

Table 4.12: Detail runtime results on the 4× upscaled FMEA diagnosis problems.

4.4.6 Discussion

Comparing the FMEA-based samples to the artificial samples, our method has to perform more iterations within the latter, as illustrated by *max_iter* in Table 4.1 (similar data in Table 4.9). This happens because the graph structure of FMEA-based samples is limited by at most three levels with the last level only containing the explanation node while the artificial samples are generated with deeper structure intendedly [52]. Additionally, the number of sets to be hit in the corresponding MHS problem in each 1-step abduction (4.9) is larger. Therefore, there are 5 problems in the FMEA samples that are not effectively solved by our naive MHS enumerator. In a comparison with the MHS enumerator of PySAT [41] on a single instance, *Sparse matrix* finishes in about 25 *mins* without PySAT, while with PySAT it takes just under 4 *secs*. This highlights the importance of implementing the 1-step abduction in Definition 54. Due to that fact, our methods work fine with the naive MHS implementation that we only call PySAT in some problems of the FMEA samples and in the only problem that we already failed to solve in the Artificial samples II.

In overall, linear algebraic approaches are competitive with other algorithms. An important note is that we have conducted experiments with a single-threaded setup to make a fair comparison. *Dense matrix* may perform better in a parallel environment, however, the current implementation is not fully exploit parallel computation that is because we have not yet focused on this aspect at this moment. *Sparse matrix* is the most stable algorithm in terms of **std** among those having the highest **#solved**. The results show the potential for further advancement to the linear algebraic method using sparse representation.

In fact, our proposed Algorithm 4 is parallelizable because we do not use any complex data structure and the computation on each explanation vector is independent of

each other. Hence, there are many rooms for further improvement using a more powerful BLAS library which is designed for parallel computing. Further, we can consider improving the 1-step abduction for P_{Or} by employing a simple caching technique that memorizes the output of the MHS enumerator to save redundant computation.

4.5 Conclusion

We have proposed a linear algebraic approach for solving PHCAP using the abductive matrix in either dense or sparse formats. Experimental results demonstrate that Algorithm 4 is competitive with other existing methods especially with sparse representation on upscaled benchmarks. The merit of solving PHCAP in vector space is not only the scalability but also the capability of integrating with other Artificial Intelligence (AI) techniques *e.g.* Artificial Neural Network (ANN). In the experiments using datasets of [52], the goal was the enumeration of minimal explanations but no constraints are introduced to express incompatible assumptions.

In this thesis, we have proposed how to deal with consistency in the presence of constraints using Proposition 7. We can consider a more compiled method for handling consistency by computing the minimal explanations of $\bot$ in a matrix $M_\bot$, which correspond to *nogoods* - a set of assumptions that causes contradiction - of the Assumption-based Truth Maintenance System (ATMS) [91]. ATMS keeps a number of different sets of *nogoods* so that these sets can be avoided in the future while exploring [20]. Using linear algebraic methods, checking the consistency of an explanation vector v is made easy by verifying that $M_\bot \cdot v = \mathbf{0}$. Further improvement can be considered on tabling method [71] to memorize explored nodes then we can reuse them directly without recomputing again. This improvement can be very crucial in some problem instances *e.g.* the hardest one in the Artificial samples II. To achieve this advancement, we have to extend current sparse representation to a way that we can associate or embed more data to each explanation vector in the explanation matrix.

In addition, taking the MHS problem into account in vector space is a potential research topic. If we can handle the MHS problem efficiently in the vector space, we can unlock the capability of GPU computing in solving large-size PHCAPs. Future work also includes developing an efficient method for abduction with normal logic programs in vector spaces or explore the ability to deal with probabilistic logic.

Chapter 5

Related Work

In this chapter, we will provide an overview about related work that have been done recently.

5.1 Deduction

Earlier work in this topic, such as `smodels` [90] and `clasp` [31] rely on SAT-solving techniques to find stable models of a normal program. In particular, `clasp` applies a variant of Conflict-Driven Clause Learning [58], where conflicts during search are analyzed for their underlying causes to facilitate more efficient backtracking.

Recently, several researches have recently demonstrated the ability to apply linear algebraic methods to compute Logic Programming (LP) in parallel. For example, Rocktäschel and Riedel employ high-order tensors to support both deductive and inductive inferences for a limited class of logic programs [72].

The matrix representation of logic programs was first coined by Sakama et al. [75] and then followed up by other researchers in [76, 61, 62]. Later, the non-differentiable computation of 3-valued models of a program's completion in vector spaces was considered as a first step towards computing supported models [84].

With a different perspective, Sato and Kojima propose a differentiable framework for logic program inference as a step toward realizing flexible and scalable logical inference [81]. The basic idea is to define a cost function, which is made up of matrix (tensor), in a continuous space. Then they compute the minimizer for the function by gradient descent or Newton's method. Using artificial data and real data, they also demonstrated empirically the potential of this approach by a variety of tasks including abduction, random SAT, rule refinement and probabilistic modeling based on answer set (supported model) sampling. They have further extended the idea to develop MatSat - a matrix-based differentiable SAT solver - and presented this method outperforms

all the Conflict-Driven Clause Learning (CDCL) type solvers using a random benchmark set from SAT 2018 competition [82].

Based on the similar idea of Sakama et al. in representing program matrix, Aspis et al. have introduced a gradient-based search method for the computation of stable and supported models of normal logic programs in continuous vector spaces [7]. The main idea is to define a special function, which is built on top of similar linear algebraic computation we have presented in Chapter 3, then apply Newton's method for finding its roots. However, this method is an approximation method, in particular, they consider a successful case for the algorithm only when a root, which is semantically equivalent to an interpretation, is found. There are many ways developing this method further such as employing alternative approximation functions, or establishing good choices of initial vector.

Extending the idea of [7], Takemura and Inoue have presented another gradient-based approach to compute supported models approximately [92]. Takemura and Inoue defined a loss function based on the implementation of the immediate consequence operator by matrix-vector multiplication and then applied gradient descent [12] to optimize it. The results of this work on several experiments highlight their improvements over Aspis et al.'s method in terms of success rates of finding correct supported models of normal logic programs.

5.2 Abduction

Propositional abduction has been solved using propositional satisfiability (SAT) techniques in [40], in which a quantified MaxSAT is employed and implicit hitting sets are computed. Another approach to abduction is based on the search for stable models of a logic program [33]. In [74], Saikko et al. have developed a technique to encode propositional abduction problem as disjunctive logic programming under answer set semantics. Answer set programming has also been employed for first-order Horn abduction in [86], in which all atoms are abduced and weighted abduction is employed.

In terms of linear algebraic computation, Sato et al. developed an approximate computation to abduce relations in Datalog [83], which is a new form of predicate invention in Inductive Logic Programming [59]. They did empirical experiments on linear and recursive cases and indicated that the approach can successfully abduce base relations, but their method cannot compute explanations consisting of possible abducibles in diagnosis.

In this regard, Aspis et al. have proposed a linear algebraic transformation for abduction by exploiting Sakama et al.'s algebraic transformation [5]. They have defined an explanatory operator based on third-order tensor for computing abduction in Horn

propositional programs that simulates deduction through Clark completion for the abductive program [15]. The dimension explosion would arise, unfortunately, Aspis et al. have not yet reported an empirical work. Aspis et al. propose encoding every single rule as a slice in a third-order tensor then they achieve the growth naturally. Then, they only consider removing columns that are duplicated or inconsistent with the program. According to our analysis, their current method has some points that can be improved to avoid redundant computation. First, they can consider merging all slices of *And*-rules into a single slice to limit the growth of the output matrix. Second, they have to consider incorporating Minimal Hitting Sets (MHS)-based elimination strategy, otherwise, their method will waste a lot of computation and resources on explanations that are not minimal.

Chapter 6

Conclusion and Future Work

We have continued to explore linear algebraic computation approaches for logic programming in the thesis. Let us wrap up all the chapters and discuss future research directions to tackle our current shortcomings.

6.1 Conclusion

First of all in deductive reasoning, we have analyzed the sparsity of matrix representation for Logic Programming (LP) and also considered a number of general-purpose matrix sparse representations for LP. The employment of sparse representation into linear algebraic computation of LP needs a redesign in the core algorithm for better efficiency. We then have proposed an improved implementation of the immediate consequence operator T_P which is proved outperform the previous implementation in terms of both theory and practice. In particular, the proposed algorithm has better time and complexity as we have proved, and also outweighs the previous implementation in all the conducted benchmarks. Additionally, in case the number of negations is limited, our proposed method with sparse representation is way more robust than other Conflict-Driven Clause Learning (CDCL)-based solvers.

Continue to explore the light of linear algebraic approaches, we have successfully applied it to solve the Horn abduction problem. The performance gain is competitive with other existing methods for abduction. The main contribution of this work is the exact algorithm to compute all minimal explanations in vector spaces and a theory to prove its correctness in general. In addition, we have verified all the theory through a number of variants form the Failure Modes and Effects Analysis (FMEA)-based benchmark dataset. This exploration sets a trusted foundation before we moving on to the next step that to realize rule learning in the language of linear algebra.

6.2 Future work

At this early stage, we currently have focused on exact methods to develop linear algebraic computation for LP before exploring more use cases of matrices and vectors in logic representation and reasoning. Thus, one of the future directions could be to consider approximation approaches in which we have to propose a simple and efficient way to eliminate combinations that are violated constraints, or being contradicted by others, or being not sufficient for further reasoning steps.

Another direction is to continue to explore sparse representation in a way that we can associate or embed more data to each interpretation vector in the interpretation matrix. We have done some work in this regard to combine both set operations and matrix operations in a unique sparse representation, however, the experiment results at this moment are still limited due to the lack of advanced skills for a more efficient implementation.

Using linear algebraic computation in solving some arisen problems in current research also should be taken into consideration such as solving the Minimal Hitting Sets (MHS) problem. It is obvious that enumerating MHSs is the core of numerous combinatorial problems. Hence, if we can handle it efficiently in the vector spaces, we can extend the ability of linear algebraic approaches in solving a class of combinatorial problems.

Last but not least, future work should also include exploring the ability to deal with probabilistic logic and bridging linear algebraic computation and neural computation. This can be one of the key ideas for the next generation of explainable learning models.

Bibliography

[1] Jose Julio Alferes, Joao Alexandre Leite, Luis Moniz Pereira, Halina Przymusinska, and Teodor C. Przymusinski. Dynamic updates of non-monotonic knowledge bases. *The journal of logic programming*, 45(1-3):43–70, 2000.

[2] Krzysztof R. Apt and Marc Bezem. Acyclic programs. *New Generation Computing*, 9(3/4):335–364, 1991. DOI: 10.1007/BF03037168.

[3] Krzysztof R Apt, Howard A Blair, and Adrian Walker. Towards a theory of declarative knowledge. In *Foundations of deductive databases and logic programming*, pages 89–148. Elsevier, 1988.

[4] Yaniv Aspis. *A Linear Algebraic Approach to Logic Programming*. Master thesis at Imperial College London, 2018.

[5] Yaniv Aspis, Krysia Broda, and Alessandra Russo. Tensor-based abduction in horn propositional programs. In *ILP 2018*, volume 2206 of *CEUR Workshop Proceedings*, pages 68–75, 2018.

[6] Yaniv Aspis, Krysia Broda, and Alessandra Russo. Tensor-based abduction in horn propositional programs. volume 2206, pages 68–75. CEUR Workshop Proceedings, 2018. DOI: 10.1145/200836.200838.

[7] Yaniv Aspis, Krysia Broda, Alessandra Russo, and Jorge Lobo. Stable and supported semantics in continuous vector spaces. In *Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning, KR 2020, Rhodes, Greece*, pages 59–68, 2020. DOI: 10.24963/kr.2020/7.

[8] Kinjal Basu, Farhad Shakerin, and Gopal Gupta. Aqua: Asp-based visual question answering. In *International Symposium on Practical Aspects of Declarative Languages*, pages 57–72. Springer, 2020.

[9] Colin Bell, Anil Nerode, Raymond T Ng, and VS Subrahmanian. Mixed integer programming methods for computing nonmonotonic deductive databases. *Journal of the ACM (JACM)*, 41(6):1178–1215, 1994.

[10] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on cuda. Technical report, Citeseer, 2008.

[11] Craig Boutilier and Veronica Beche. Abduction as belief revision. *Artificial intelligence*, 77(1):43–94, 1995.

[12] Stephen Boyd, Stephen P Boyd, and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[13] James R Bunch and Donald J Rose. *Sparse matrix computations*. Academic Press, 2014.

[14] William W Cohen. Tensorlog: A differentiable deductive database. *arXiv preprint arXiv:1605.06523*, 2016.

[15] Luca Console, Daniele Theseider Dupré, and Pietro Torasso. On the relationship between abduction and deduction. *Journal of Logic and Computation*, 1(5):661–690, 1991.

[16] Domenico Corapi, Alessandra Russo, and Emil Lupu. Inductive logic programming in answer set programming. In *International conference on inductive logic programming*, pages 91–97. Springer, 2011.

[17] Wang-Zhou Dai, Qiuling Xu, Yang Yu, and Zhi-Hua Zhou. Bridging machine learning and logical reasoning by abductive learning. In *Neural Information Processing Systems 2019*, volume 32. Curran Associates, Inc., 2019.

[18] Fabio A. D'Asaro, Matteo Spezialetti, Luca Raggioli, and Silvia Rossi. 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, 9 2020. DOI: 10.24963/kr.2020/88.

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

[20] Johan de Kleer. An assumption-based TMS. *Artif. Intell.*, 28(2):127–162, 1986. DOI: 10.1016/0004-3702(86)90080-9.

[21] Johan de Kleer. Problem solving with the ATMS. *Artif. Intell.*, 28(2):197–224, 1986. DOI: 10.1016/0004-3702(86)90082-2.

[22] Marc Denecker and Danny De Schreye. Representing incomplete knowledge in abductive logic programming. *Journal of Logic and Computation*, 5(5):553–577, 1995.

[23] Thomas Eiter and Georg Gottlob. The complexity of logic-based abduction. *Journal of the ACM (JACM)*, 42(1):3–42, 1995.

[24] Kave Eshghi. Abductive planning with event calculus. In *ICLP/SLP*, pages 562–579, 1988.

[25] Ken-Ichi Funahashi. On the approximate realization of continuous mappings by neural networks. *Neural networks*, 2(3):183–192, 1989.

[26] Daniel Furelos-Blanco, Mark Law, Anders Jonsson, Krysia Broda, and Alessandra Russo. Induction and exploitation of subgoal automata for reinforcement learning. *Journal of Artificial Intelligence Research*, 70:1031–1116, 2021.

[27] Johannes Fürnkranz, Dragan Gamberger, and Nada Lavrač. *Foundations of rule learning*. Springer Science & Business Media, 2012.

[28] Andrew Gainer-Dewar and Paola Vera-Licona. The minimal hitting set generation problem: algorithms and computation. *SIAM Journal on Discrete Mathematics*, 31(1):63–100, 2017.

[29] Artur d'Avila Garcez and Luis C Lamb. Neurosymbolic ai: the 3rd wave. *arXiv preprint arXiv:2012.05876*, 2020.

[30] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. ISBN 0-7167-1044-7.

[31] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Answer set solving in practice. *Synthesis lectures on artificial intelligence and machine learning*, 6(3):1–238, 2012.

[32] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Philipp Wanko. Theory solving made easy with clingo 5. In *OASIcs-OpenAccess Series in Informatics*, volume 52. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[33] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, volume 88, pages 1070–1080, 1988.

[34] Geoffrey Gordon, Sue Ann Hong, and Miroslav Dudík. First-order mixed integer linear programming. *arXiv preprint arXiv:1205.2644*, 2012.

[35] Russell Greiner, Barbara A. Smith, and Ralph W. Wilkerson. A correction to the algorithm in reiter's theory of diagnosis. *Artif. Intell.*, 41(1):79–88, 1989. DOI: 10.1016/0004-3702(89)90079-9.

[36] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[37] Fred G Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software (TOMS)*, 4(3):250–269, 1978.

[38] Pascal Hitzler, Steffen Hölldobler, and Anthony Karel Seda. Logic programs and connectionist networks. *Journal of Applied Logic*, 2(3):245–272, 2004.

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

[40] Alexey Ignatiev, António Morgado, and João Marques-Silva. Propositional abduction with implicit hitting sets. In *ECAI 2016*, volume 285 of *Frontiers in Artificial Intelligence and Applications*, pages 1327–1335. IOS Press, 2016. DOI: 10.3233/978-1-61499-672-9-1327.

[41] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. PySAT: A Python toolkit for prototyping with SAT oracles. In *SAT*, pages 428–437, 2018. DOI: 10.1007/978-3-319-94144-8_26.

[42] Alexey Ignatiev, Nina Narodytska, and Joao Marques-Silva. Abduction-based explanations for machine learning models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1511–1519, 2019.

[43] Katsumi Inoue. Linear resolution for consequence finding. *Artif. Intell.*, 56(2-3): 301–353, 1992.

[44] Katsumi Inoue. Automated abduction. In Antonis C. Kakas and Fariba Sadri, editors, *Computational Logic: Logic Programming and Beyond: Essays in Honour of Robert A. Kowalski Part II, LNAI 2408*, pages 311–341. Springer, 2002.

[45] Katsumi Inoue. Induction as consequence finding. *Machine Learning*, 55(2):109–135, 2004.

[46] Katsumi Inoue. Meta-level abduction. *IfCoLog Journal of Logics and their Applications*, 3(1):7–36, 2016.

[47] Katsumi Inoue and Chiaki Sakama. Disjunctive abduction. *New Generation Computing*, 37(2):219–243, 2019.

[48] John R Josephson and Susan G Josephson. *Abductive inference: Computation, philosophy, technology*. Cambridge University Press, 1996.

[49] Ulrich Junker. Quickxplain: Preferred explanations and relaxations for over-constrained problems. In *Proceedings of the 19th National Conference on Artifical Intelligence*, AAAI'04, page 167–172. AAAI Press, 2004. ISBN 0262511835.

[50] Antonis C Kakas, Robert A Kowalski, and Francesca Toni. The role of abduction in logic programming. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 235–324. Oxford University Press, 1998.

[51] Roxane Koitz-Hristov and Franz Wotawa. Applying algorithm selection to abductive diagnostic reasoning. *Applied Intelligence*, 48(11):3976–3994, 2018.

[52] Roxane Koitz-Hristov and Franz Wotawa. Faster horn diagnosis-a performance comparison of abductive reasoning algorithms. *Applied Intelligence*, 50(5):1558–1572, 2020.

[53] Robert Kowalski. *Logic for problem solving*. North Holland, Elsevier, 1979.

[54] Jérôme Kunegis. Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 1343–1350, 2013. DOI: 10.1145/2487788.2488173.

[55] Mark Law, Alessandra Russo, and Krysia Broda. Inductive learning of answer set programs. In *European Workshop on Logics in Artificial Intelligence*, pages 311–325. Springer, 2014.

[56] Guohua Liu, Tomi Janhunen, and Ilkka Niemela. Answer set programming via mixed integer programming. In *Thirteenth International Conference on the Principles of Knowledge Representation and Reasoning*, 2012.

[57] John W Lloyd. *Foundations of logic programming*. Springer Science & Business Media, 2012.

[58] Joao Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning sat solvers. In *Handbook of satisfiability*, pages 133–182. IOS press, 2021.

[59] Stephen Muggleton. Inductive logic programming. *New Generation Computing*, 8 (4):295–318, 1991. DOI: 10.1007/BF03037089.

[60] Hidetomo Nabeshima, Koji Iwanuma, Katsumi Inoue, and Oliver Ray. Solar: An automated deduction system for consequence finding. *AI communications*, 23 (2-3):183–203, 2010.

[61] Hien D Nguyen, Chiaki Sakama, Taisuke Sato, and Katsumi Inoue. Computing logic programming semantics in linear algebra. In *International Conference on Multi-disciplinary Trends in Artificial Intelligence*, pages 32–48. Springer, 2018.

[62] Hien D Nguyen, Chiaki Sakama, Taisuke Sato, and Katsumi Inoue. An efficient reasoning method on logic programming using partial evaluation in vector spaces. *Journal of Logic and Computation*, 31(5):1298–1316, 03 2021. ISSN 0955-792X. DOI: 10.1093/logcom/exab010.

[63] Tuan Quoc Nguyen, Katsumi Inoue, and Chiaki Sakama. Enhancing linear algebraic computation of logic programs using sparse representation. volume 325 of *EPTCS Online Proceedings of ICLP (2020)*, pages 192–205, 2020. DOI: 10.4204/EPTCS.325.24.

[64] Tuan Quoc Nguyen, Katsumi Inoue, and Chiaki Sakama. Linear algebraic computation of propositional horn abduction. In *2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 240–247. IEEE, 2021. DOI: 10.1109/ICTAI52525.2021.00040.

[1] Tuan Quoc Nguyen, Katsumi Inoue, and Chiaki Sakama. Enhancing linear algebraic computation of logic programs using sparse representation. *New Generation Computing*, 40(1):225–254, 2022. DOI: 10.1007/s00354-021-00142-2.

[66] Maximilian Nickel, Kevin Murphy, Volker Tresp, and Evgeniy Gabrilovich. A review of relational machine learning for knowledge graphs. *Proceedings of the IEEE*, 104(1):11–33, 2015.

[67] Shan-Hwei Nienhuys-Cheng, Ronald De Wolf, et al. *Foundations of inductive logic programming*, volume 1228. Springer Science & Business Media, 1997.

[68] Gabriele Paul. AI approaches to abduction. In Dov M. Gabbay and Rudolf Kruse, editors, *Handbook of Defeasible Reasoning and Uncertainty Management Systems*, volume 4, pages 35–98. Springer, 2000. ISBN 978-94-017-1733-5. DOI: 10.1007/978-94-017-1733-5_2.

[69] Bernhard Peischl and Franz Wotawa. Computing diagnosis efficiently: A fast theorem prover for propositional horn theories. In *Proc. of the 14th Int. Workshop on Principles of Diagnosis*, pages 175–180, 2003.

[70] Raymond Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1): 57–95, 1987. DOI: 10.1016/0004-3702(87)90062-2.

[71] Ricardo Rocha, Fernando Silva, and Vitor Santos Costa. On applying or-parallelism and tabling to logic programs. *Theory and Practice of Logic Programming*, 5(1-2):161–205, 2005.

[72] Tim Rocktäschel and Sebastian Riedel. End-to-end differentiable proving. In *Neural Information Processing Systems 2017*, pages 3788–3800, 2017.

[73] Tim Rocktäschel, Matko Bošnjak, Sameer Singh, and Sebastian Riedel. Low-dimensional embeddings of logic. In *Proceedings of the ACL 2014 Workshop on Semantic Parsing*, pages 45–49, 2014.

[74] Paul Saikko, Johannes Peter Wallner, and Matti Järvisalo. Implicit hitting set algorithms for reasoning beyond NP. In *KR*, pages 104–113, 2016.

[75] Chiaki Sakama, Katsumi Inoue, and Taisuke Sato. Linear algebraic characterization of logic programs. In *International Conference on Knowledge Science, Engineering and Management*, pages 520–533. Springer, 2017.

[76] Chiaki Sakama, Hien D Nguyen, Taisuke Sato, and Katsumi Inoue. Partial evaluation of logic programs in vector spaces. *arXiv preprint arXiv:1811.11435*, 2018.

[77] Chiaki Sakama, Katsumi Inoue, and Taisuke Sato. Logic programming in tensor spaces. *Annals of Mathematics and Artificial Intelligence*, pages 1–21, in print 2021. DOI: https://doi.org/10.1007/s10472-021-09767-x.

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

[79] Taisuke Sato. Embedding tarskian semantics in vector spaces. In *Workshops at the Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

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

[81] Taisuke Sato and Ryosuke Kojima. Logical inference as cost minimization in vector spaces. In *International Joint Conference on Artificial Intelligence*, pages 239–255. Springer, 2019.

[82] Taisuke Sato and Ryosuke Kojima. Matsat: a matrix-based differentiable sat solver. *arXiv preprint arXiv:2108.06481*, 2021.

[83] Taisuke Sato, Katsumi Inoue, and Chiaki Sakama. Abducing relations in continuous spaces. In *IJCAI*, pages 1956–1962, 2018.

[84] Taisuke Sato, Chiaki Sakama, and Katsumi Inoue. From 3-valued semantics to supported model computation for logic programs in vector spaces. In *ICAART (2)*, pages 758–765, 2020.

[85] Torsten Schaub and Stefan Woltran. Special issue on answer set programming. *KI-Künstliche Intelligenz*, 32(2):101–103, 2018.

[86] Peter Schüller. Modeling variations of first-order horn abduction in answer set programming. *Fundam. Informaticae*, 149(1-2):159–207, 2016. DOI: 10.3233/FI-2016-1446.

[87] Bart Selman and Hector J Levesque. Abductive and default reasoning: A computational core. In *AAAI*, pages 343–348, 1990.

[88] Luciano Serafini and Artur d'Avila Garcez. Logic tensor networks: Deep learning and logical reasoning from data and knowledge. *arXiv preprint arXiv:1606.04422*, 2016.

[89] Farhad Shakerin and Gopal Gupta. White-box induction from svm models: Explainable ai with logic programming. *Theory and Practice of Logic Programming*, 20(5):656–670, 2020. DOI: 10.1017/S1471068420000356.

[90] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.

[91] Guy Lewis Steele Jr. The definition and implementation of a computer programming language based on constraints. Technical report, Massachusetts Inst Of Tech Cambridge Artificial Intelligence Lab, 1980.

[92] Akihiro Takemura and Katsumi Inoue. Gradient-based supported model computation in vector spaces. 2021.

[93] Maarten H. van Emden and Robert A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23(4):733–742, 1976. DOI: 10.1145/321978.321991.

[94] Franz Wotawa. Failure mode and effect analysis for abductive diagnosis. In *DARe@ ECAI*, 2014.

[95] Bishan Yang, Scott Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. Embedding entities and relations for learning and inference in knowledge bases. In *Proceedings of ICLR 2015*, 2015.

List of Publications

Published Journal Papers

[1] Tuan Quoc Nguyen, Katsumi Inoue, and Chiaki Sakama. Enhancing linear algebraic computation of logic programs using sparse representation. *New Generation Computing*, 40(1):225–254, 2022. DOI: 10.1007/s00354-021-00142-2.

Published Conference Papers

[1] Tuan Quoc Nguyen, Katsumi Inoue, and Chiaki Sakama. Enhancing linear algebraic computation of logic programs using sparse representation. volume 325 of *EPTCS Online Proceedings of ICLP (2020)*, pages 192–205, 2020. DOI: 10.4204/EPTCS.325.24.

[2] Tuan Quoc Nguyen, Katsumi Inoue, and Chiaki Sakama. Linear algebraic computation of propositional horn abduction. In *2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 240–247. IEEE, 2021. DOI: 10.1109/ICTAI52525.2021.00040.

Other Publications

[1] Tuan Quoc Nguyen, Katsumi Inoue, and Chiaki Sakama. Abductive logic programming and linear algebraic computation. *Handbook of Abductive Cognition*, 2022.