5-9-2023

Rule Mining from Knowledge Bases: Semantics, Queries, and Estimations

Bhaskar Krishna Gangadhar
bg4437@rit.edu

Rule Mining from Knowledge Bases: Semantics, Queries, and Estimations

by

Bhaskar Krishna Gangadhar

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science
in Computing and Information Sciences

B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology

9th May 2023

Abstract

A knowledge base is a large collection of real-world facts which can be interpreted by both humans and machines. Most of these knowledge bases are incomplete as some are extracted from natural language sources that contain gaps, while others are manually developed and extended. Rule mining is the process of discovering rules that succinctly capture the inference patterns present in the knowledge base at hand. These rules can be executed and new, missing facts can be inferred to complete knowledge bases. The new rules also help identify errors in the knowledge base and help understand its content better. This thesis deals with one popular rule mining algorithm, AMIE. Knowledge bases do not contain negative facts. So, in order to measure the quality of the mined rules, we need to deduce negative evidence from the actual (positive) facts present in the knowledge base. In the standard approach, we assume the knowledge is complete and any missing information in the knowledge base is considered a negative. However, knowledge bases operate under the open world assumption, that is, missing information in the knowledge base is treated as unknown. AMIE introduces a less restrictive measure where facts are considered either negative or unknown depending on the positive facts present in the knowledge base. The confidence of a given rule is measured by counting the number of occurrences of facts in the knowledge base that fit the rule. A rule contains multiple components, each component of the rule is matched against the whole knowledge base. This confidence measure follows Prolog semantics where different components of the rule can share the same element of the fact. We observed this approach to measuring confidence does not always obtain the best result. In this thesis, we explore a new approach using Graph semantics, which restricts different components of the rule from sharing the same element of a fact, resulting in confidence gain for certain rules. Our experimental results show that we mined more rules when we use Graph semantics as a confidence measure compared to Prolog semantics. Confidence measure of certain rules improve with Graph semantics. AMIE uses the information spread (functionality) around one component of the rule to dictate how the confidence will be measured. We demonstrate gain in confidence for certain rules by altering the definition of functionality. In addition, we propose a new Apriori algorithm for rule mining. We conceptualize rules as graphs and generate unique graph patterns for rules of various sizes. We then expand these patterns instead of growing rules, which reduces the number of queries to the knowledge base.

The thesis ”Rule Mining from Knowledge Bases: Semantics, Queries, and Estimations” by Bhaskar Krishna Gangadhar has been examined and approved by the following Examination Committee:

Prof. Carlos R Rivero
Associate Professor
Thesis Committee chair

____

Signature

____

Date

Prof. Thomas J. Borrelli
Principal Lecturer

____

Signature

____

Date

Prof. Michael Mior
Assistant Professor

____

Signature

____

Date

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, we have observed a rise in large knowledge bases (KBs), such as NELL [3], YAGO [17], DBpedia [9] and Freebase [2]. These knowledge bases represent real-world facts in the form of entities and their relationships. The scale of these knowledge bases continues to grow, with some knowledge bases containing hundreds of millions of facts, and yet they are not complete [15]. Knowledge bases are curated either manually by humans or extracted using automated data scraping techniques over the Internet, contributing to the incompleteness of the knowledge bases. Rule mining is a task that helps complete a knowledge base by discovering frequent inference patterns. These patterns in the form of rules can be executed and new knowledge discovered. Rule mining uses only the internal information present in the knowledge base, i.e., the facts it comprises, to mine rules. There are other approaches to complete knowledge bases that rely on external sources; however, they are out of scope of this thesis. AMIE [7] [6] [8] is a popular mining algorithm that utilizes association rule mining to extract information from RDF-style knowledge bases. In these knowledge bases, every fact is in the form of a $\textit{triple}(s, p, o)$ (also referred to as Atom) with s denoting the subject, p the predicate or relationship, and o denoting the object. These triples can also be represented as $p(s, o)$. For example, $\textit{attended}(\textit{bhaskar}, \textit{csci}620)$ is a triple representing a fact in a knowledge base that a student "bhaskar" has attended the class "csci620", and $\textit{attended}(X, Y)$ is an abstraction of the fact where X, Y are variables. A rule is composed of two parts $\mathcal{B} \Rightarrow H$, where H is head atom and $\mathcal{B}$ is a non-empty set of body atoms that form a conjunction

(Boolean AND).

Proposal This thesis focuses on the confidence measure defined in the AMIE paper and extends this confidence computation by introducing new semantics. The confidence measure helps us judge the quality of the mined rules based on the information known to the knowledge base. Confidence of a rule is measured by the ratio of number of facts in the knowledge base that satisfy both the body atom and the head atom (entire rule), divided by the number of facts that satisfy just the body of the rule. Each atom of the rule is mapped to facts in the knowledge base that fits it. This definition of confidence is a standard way of computing rule quality. AMIE also introduces a new, less restrictive measurement approach called PCA (Partial completeness assumption) confidence. The PCA confidence of a rule refers to the number of facts that meet the conditions of the rule, with one component, subject or object, of the head atom fixed while the other component is left blank. We observe that all these definitions follow Prolog semantics [4], where different free variables across atoms can share the element of a triple and contribute to the confidence.

We introduce a Graph semantics approach to confidence computation where we restrict different free variables from substituting the element of a fact. We also study the functionality of a rule which decides whether subject or object should be fixed while computing the PCA confidence of a rule. We present 4 different policies to compute the confidence of a rule as follows: Prolog semantics with functionality, Prolog semantics with best PCA confidence, Graph semantics with functionality, and Graph semantics with best PCA confidence. In our experiments, we compare the number of rules mined when each policy is used. We also discuss the increase in confidence values for several rules as well as certain rules that are missing from the output and additional rules that were mined.We present details on how functionality works and show that certain rules gain confidence when we alter the definition of functionality.

We have implemented the rule mining algorithm using Java, and Neo4j, a graph database. The implementation allows us to represent any rule in the form of a Cypher query, Neo4j database querying language. A unique feature of Cypher is its ability to visually match patterns and relationships. It accomplishes this through a syntax that resembles ASCII-art, where circular brackets are used to represent nodes, and arrows denote relationships and their directions, as in $(nodes) - [: ARE\_CONNECTED\_TO]- > (otherNodes)$. A rule is essentially a representation of relationships among various types of nodes. All the quality measure definitions in the AMIE algorithm can be represented as Cypher queries, eliminating the need to write code logic. Once the knowledge base is loaded into the graph database, we can convert a rule to its Cypher query representation and compute its confidence measures without the need to mine the rule.

The AMIE algorithm is iterative in exploring the search space while generating rules. We have explored graph enumeration methods to efficiently generate rules of a particular size while taking advantage of the fact that the knowledge base is now represented as a graph. We call this new approach Apriori, as it mines all possible rules of a particular size before advancing to mine next size rules. This new approach represents rules as graph patterns and expands these graph patterns unlike the AMIE's algorithm, where each rule is expanded. This reduced the number of database queries required to mine all the possible rules of a particular size, as there are fixed number of unique graphs patterns.

The following research questions guide this study:

- Is Graph semantics more adequate than Prolog semantics to measure quality in knowledge bases?
- Can we gain confidence by independently computing functionality for each rule?
- Can we optimize the AMIE algorithm to reduce the number of queries to the knowledge base?

Contributions

- Graph semantics.
- Implementation of rule mining algorithm, AMIE, using Java and Neo4j.
- Four different policies to mine rules.
- Apriori approach to rule mining.

The rest of the report is organized as follows. Chapter 2 establishes the necessary background information required to understand the rule mining process. Chapter 3 explains our approach to implementing the rule mining algorithm, Cypher queries used to compute various measures, introduce the Graph semantics policy, and an Apriori approach to mining rules. In Chapter 4 we present our experimental results. Chapter 5 summarizes the work and conclusions. The Appendix section contains additional implementation details and reference to the Git repository hosting the complete implementation.

Chapter 2

Background

This chapter introduces graphs, graph databases, Neo4j, knowledge graphs, and rule mining.

2.1 Graphs and Graph Databases

A graph is an ordered pair $G = (V, E)$, where V is a set of vertices (sometimes referred to as points, junctions, or nodes) and E is a set of edges (also called relationships or links). Edges can either be directed (directed graph, see Figure 2.1) or undirected (undirected graph, see Figure 2.2).

Graph databases are a type of NoSQL (also known as Non-SQL, non-relational, and not-only-SQL) databases that use graph structures like nodes, edges, and properties (of edges) to represent and store data. We are interested in directed graphs as the direction of the edge is of significance to us. An edge represents the relationship between two nodes, and the direction of

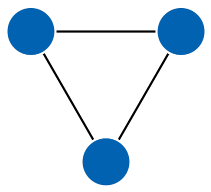
Figure 2.1: Undirected simple graph

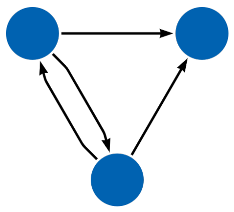
Figure 2.2: Directed graph

the edge tells us which node is the subject and the object. There are primarily two graph data models, knowledge graph model and property graph model. Knowledge graph model stores the data in the form of triples, subject-predicate-object. Two nodes represent subject and object data entities, and the edge linking these two nodes indicates the relationship between them. In property graph model each entity node has capabilities to store properties which will have details about the entity and relationships. Detailed definition of property graph model is below.

Our graph database of choice is Neo4j which implements the property graph model mentioned earlier. It uses Cypher query language [5], which is a declarative query language that allows for expressive and efficient querying, updating, and administering of the database. Neo4j allows us to have labels for nodes and edges, we will store the entity related information within these labels.

Let $\mathcal{L}$ and $\mathcal{T}$ be sets of node labels and edge types, respectively. A property graph is a tuple $G = \langle V, E, \textit{subject}, \textit{object}, \iota, \lambda, \tau \rangle$ [5] where:

- V is a finite set of *nodes*.
- $E : V \times V$ is a finite set of *relationships*.
- subject : E $\rightarrow$ V is a function that maps each edge to its *subject* node.
- object: E $\rightarrow$ V is a function that maps each edge to its *object* node.
- $\iota : (V \cup E) \times \mathcal{K} \rightarrow \mathcal{V}$ is a function that maps a (node or edge) identifier and a property key to a value. It is assumed that ι is a total function but that its "non-null support" is finite: there are only finitely many $j \in (V \cup E)$ and $k \in \mathcal{K}$ such that $\iota(j, k) \neq$ null.
- $\lambda : V \rightarrow 2^{\mathcal{L}}$ is a function that maps each node identifier to a finite (possibly empty) set of labels.
- $\tau : R \rightarrow T$ is a function that maps each relationship identifier to a relationship type

Figure 2.3 shows an example property graph where:

- $V = \{n_1, n_2, n_3, n_4\}$

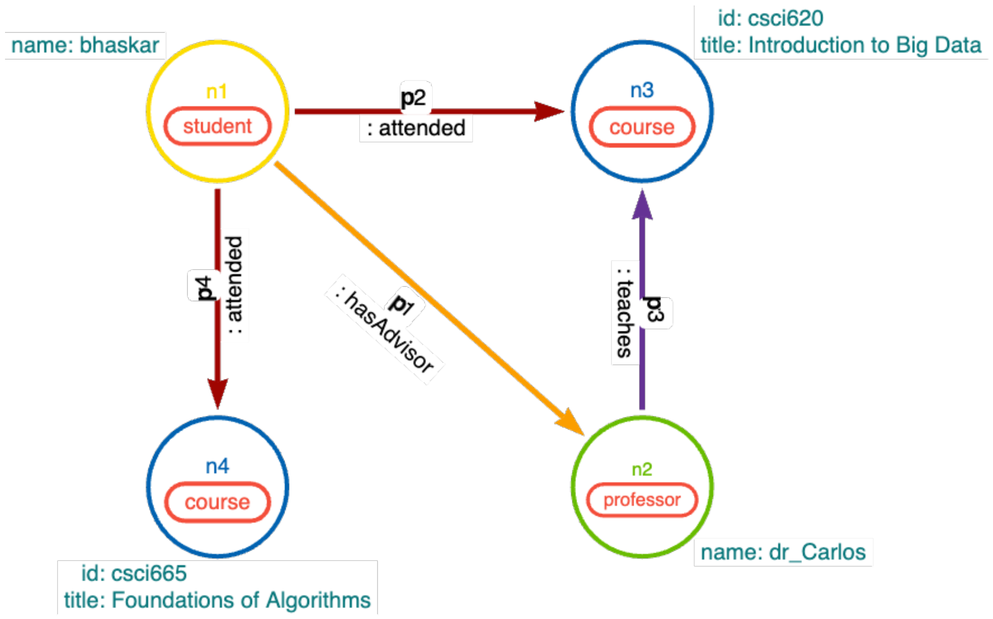

Figure 2.3: Sample property graph showing relationships between student, professor and courses

- $E = \{p_1, p_2, p_3, p_4\}$
- $subject(p_1) = n_1,\ subject(p_2) = n_1,\ subject(p_3) = n_2,\ subject(p_4) = n_1$
- $object(p_1) = n_2,\ object(p_2) = n_3,\ object(p_3) = n_3,\ object(p_4) = n_4$
- $\iota(n_1, name) = bhaskar, \iota(n_2, name) = dr\_Carlos, \iota(n_3, id) = csci620,$ $\iota(n_3, title) = Introduction\ to\ Big\ Data, \iota(n_4, id) = csci665,$ $\iota(n_4, title) = Foundations\ of\ Algorithms$
- $\lambda(n_1) = \{student\}, \lambda(n_2) = \{professor\}, \lambda(n_3) = \lambda(n_4) = \{course\}$
- $\tau(p_2) = \tau(p_4) = \{attended\}, \tau(p_1) = \{hasAdvisor\}, \tau(p_3) = \{teaches\}$

The Cypher query language uses the MATCH clause to define patterns of interest followed by a WHERE expression to filter matches with an optional relational operation clause, ending with a RETURN statement to output the items of interest. Below are some basic query examples to retrieve data from a graph.

$$MATCH\ (n)\ RETURN\ n \tag{2.1}$$

The above query returns all the nodes present in the database as rows, where each row represents an unique node present in the graph. We can return items of interest like properties, relationships, predicate types and others. Each of these items will be returned as their own column.

$$MATCH\ () - [p]- > ()\ RETURN\ r \tag{2.2}$$

We can get all the relationships in the graph using the above query. Similar to other database query languages, Cypher supports aggregations using COUNT, AVG, SUM, MIN and MAX. The query below returns the total count of *COURSE* nodes which is 2.

$$\begin{aligned} MATCH\ (n : COURSE) \\ RETURN\ \ COUNT(*) \end{aligned} \tag{2.3}$$

Cypher includes a WITH clause which is like the RETURN clause and separates the parts of a query. After the WITH clause, only the returning columns can be accessed. For example:

$$\begin{aligned} MATCH\ (s) - [\,]- > (o) \\ WITH\ DISTINCT\ s \\ RETURN\ s.name \end{aligned} \tag{2.4}$$

2.2 Knowledge base and rules

Knowledge base is a collections of facts and these facts are stored as directed graphs. The facts represent entities, relationships, events, concepts, and other information with many relationships among them. The facts can be represented in the form of triples (s, p, o), which can be alternatively represented as $p(s, o)$ that forms a directed edge where s is the subject, p is the predicate, and o is the object of the fact.

Moving forward upper case letters will be reserved to indicate variables, for example, A, B, G, H, M, X, Y, and all the constants will start with lower case letters, for example, bhaskar, dr_Carlos, nari. Figure 2.4 is an example of a knowledge graph. Notice that there are 13 nodes in the graph, and 4 unique types of predicates (teaches, hasAdvisor, employedBy and attended). For example, node *bhaskar* is connected to node $csci620$ by the predicate type *attended*. The triple (*bhaskar*, *hasAdvisor*, *dr_Carlos*) represents one fact in the knowledge graph.

An **atom** is an expression of the form $p(X, Y)$, where p is a predicate and X,Y are variables that are instantiated with facts in the knowledge graph. For example, $hasAdvisor(bhaskar, dr\_Carlos)$ is an atom where *hasAdvisor* represents the kind of relation between two people, $X = bhaskar$ and $Y = dr\_Carlos$.

A (Horn) **rule** is a set of atoms of the form $\mathcal{B} \Rightarrow H$, where $\mathcal{B}$ is a set of body atoms $B_1 \wedge B_2 \wedge \cdots \wedge B_n$, and H is the head atom [7]. Horn rule One example of such rule would be:

$$attended(X, Z) \quad \wedge \quad teaches(Y, Z) \quad \Rightarrow \quad hasAdvisor(X, Y).$$

Where atoms $attended(X, Z)$ and $teaches(Y, Z)$ form the body of the rule, and $hasAdvisor(X, Y)$ is the head. The Cypher query for the rule would have a MATCH clause for each atom. Subject and object of the atom would translate to node variables of the MATCH clause, and the predicate would represent the relationship of the edge between the atoms like shown below.

$$\begin{aligned} MATCH\ (X) - [:\ `attended`] \rightarrow (Z) \\ MATCH\ (Y) - [:\ `teaches`] \rightarrow (Z) \\ MATCH\ (X) - [:\ `hasAdvisor`] \rightarrow (Y) \\ RETURN\ DISTINCT\ X,\ Z,\ Y \end{aligned}$$

When we execute (instantiate) this rule over the graph we get 5 different combinations of nodes (Table 2.1) for X, Z, and Y that MATCH the given connection of relationships.

A rule is closed if every variable (subject and object) in the rule appears at least twice [7]. The above example rule is closed, all the variables (X, Y,

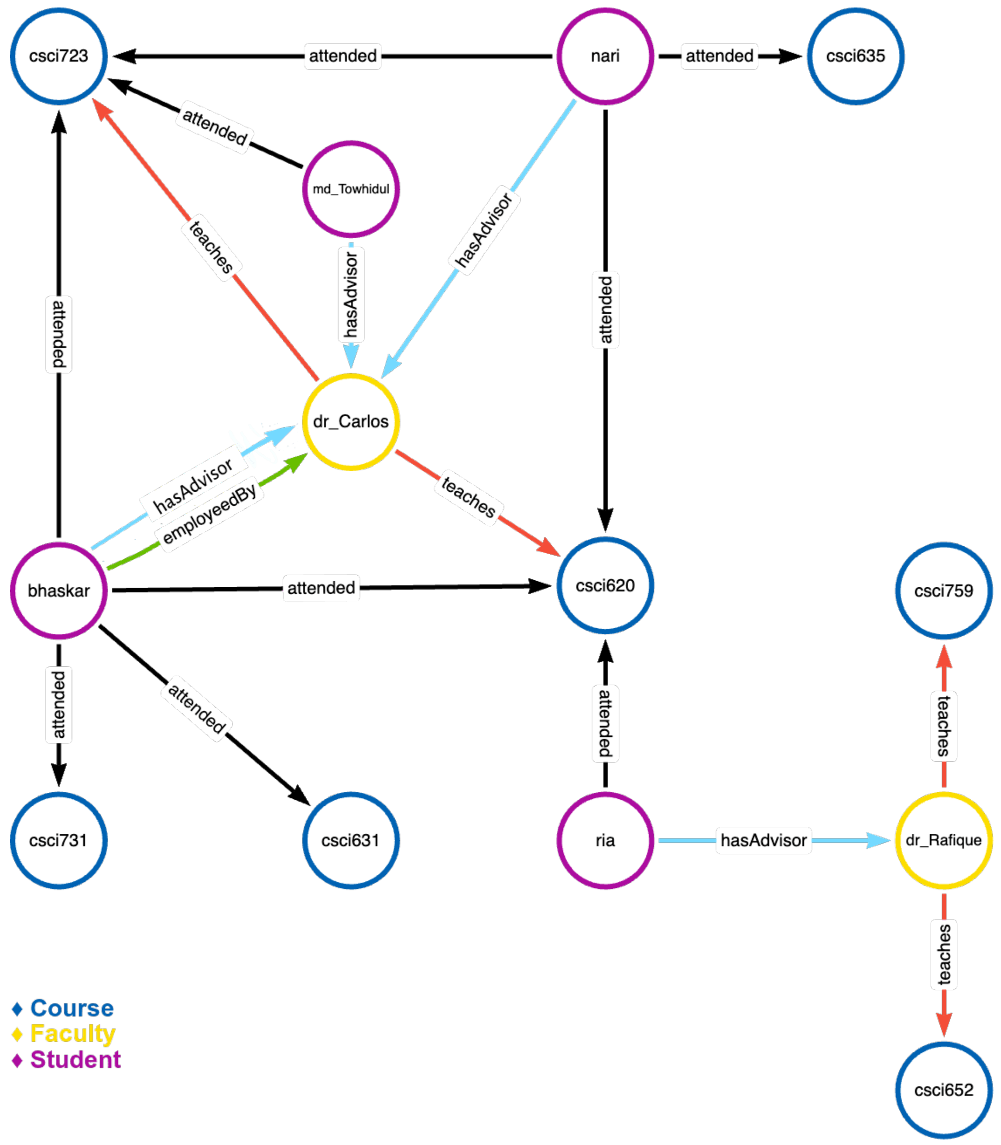

Figure 2.4: Example of a knowledge graph showing student, course and advisor information

Table 2.1: Combination of nodes that satisfy the relationship pattern of the rule [$attended(X, Z) \wedge teaches(Y, Z) \Rightarrow hasAdvisor(X, Y)$] over Figure 2.4.

X	Z	Y
md_Towhidul	csci723	dr_Carlos
nari	csci723	dr_Carlos
bhaskar	csci723	dr_Carlos
nari	csci620	dr_Carlos
bhaskar	csci620	dr_Carlos

and Z) appear twice. Example of a rule that is not closed is shown below, where both the variables Y and Z appear only once.

$$attended(X, Y) \quad \Rightarrow \quad hasAdvisor(X, Z).$$

2.3 Rule mining

Rule mining is a method for identifying frequent patterns, correlations, associations, or causal structures from large data-sets (knowledge bases). There are various automated approaches to mine a variety of rules. For example, association rule mining [1] uses transactional data. Logical rule mining [14] takes advantage of probabilistic graphical models to mine new facts for a relationship of interest. Expert rule mining [12] requires a domain expert to help understand the data before rules are mined. Each approach defines their own methodologies to mine rules. AnyBURL (Anytime botton up rule learner) [10] proposes a bottom up approach to mine logical rules. The algorithm learns all the rules of a particular size in a given time span, comparing it with the previously found rules and repeating the process. AMIE [7] mines Horn rules by following top down approach and mining all possible rules for a particular size before increasing the rule size. RLvLR [13] introduces an embedded model approach to mine rules from knowledge bases. The rules mined in all the different approaches have to be judged for their quality, for which we will now introduce various measures.

Significance

Significance of a rule quantifies the value of the rule over the knowledge graph at hand. The prediction of a rule is the result of executing the rule over the graph. A prediction is *false* if body of the rule is instantiated but the instantiation of the head atom is not present in the graph, and the prediction is *true* if the entire rule is instantiated as a conjunction (body and head atoms).

Support, a significance measure, of a rule R is the number of true predictions [7]. The total number of distinct pairs of subject and object in the head of all instantiations. For example, consider the rule

$$hasAdvisor(X, Y) \quad \Rightarrow \quad employedBy(X, Y). \tag{2.5}$$

According to our example knowledge graph in Figure 2.4, we know that only one pair of (X, Y) fits the above rule in equation 2.5 and this is our only true prediction for the rule so, the support would be 1.

$$hasAdvisor(bhaskar, dr\_Carlos) \quad \Rightarrow \quad employedBy(bhaskar, dr\_Carlos).$$

For a different rule, in equation 2.6, 3 distinct combinations of nodes (X, Y) in our example knowledge graph from Figure 2.4 fit the head of all instanitations of the rule. Table 2.2 shows these 3 distinct combinations in blue color. So the support for this rule in equation 2.6 is 3.

$$attended(X, Z) \quad \wedge \quad teaches(Y, Z) \quad \Rightarrow \quad hasAdvisor(X, Y) \tag{2.6}$$

Algorithm 1, InstantiateAtoms, shows how to instantiate a rule. The recursive algorithm requires the knowledge graph and a rule for input. It outputs all possible combinations of the subjects and the objects from the knowledge graph that fit the variables in the rule. Line 1 is the base case to the recursive algorithm. If the base case is satisfied, all possible instantiations have been found and are added to the set of all instantiations Φ. Otherwise, the algorithm selects an atom A from $\mathcal{A}$ on line 4. The atom is then instantiated by substituting its variables with nodes present in the knowledge graph. The algorithm checks if the instantiated atom A is present in the knowledge graph G on line 23, checking if an edge exists between the nodes picked for the subject and object variable. If it is, the instantiated variables and it's mappings

Table 2.2: All instantiations of the rule $[attended(X,Z) \wedge teaches(Y,Z) \Rightarrow hasAdvisor(X,Y)]$.

X	Z	Y
md_Towhidul	csci723	dr_Carlos
nari	csci723	dr_Carlos
bhaskar	csci723	dr_Carlos
nari	csci620	dr_Carlos
bhaskar	csci620	dr_Carlos

are added to the current instantiation ϕ and the algorithm continues recursively by instantiating the remaining atoms in $\mathcal{A}$, but with the instantiated variables already added to the current instantiation ϕ. Before proceeding to the next iteration of the loop, the instantiated variables are removed from the current instantiation ϕ if they were not present in the original instantiation. This is to prevent instantiated variables from being carried over to the next recursive call of the algorithm. When the algorithm terminates the set of all instantiations can be found in Φ. We will use the algorithm to define different computation measures below.

The support of a rule $B \Rightarrow H$ can be computed using the Algorithm 2. In our implementation the algorithm translates to a Neo4j Cypher query shown in the below equation 2.7. The first MATCH clause is equivalent to line 1 in the algorithm. If the rule had multiple body atoms then we would have multiple MATCH statements as well. The second MATCH clause will compute new Φ_H over the already instantiated pairs of body atoms, line 5 in Algorithm 2.7. The DISTINCT clause will count the unique instantiations of the head atom giving us the support.

$$\begin{aligned} MATCH\ (A) - [p1 : \text{`}hasAdvisor\text{`}] \rightarrow (B) \\ MATCH\ (A) - [p2 : \text{`}employedBy\text{`}] \rightarrow (B) \\ RETURN\ COUNT(DISTINCT\ id(p2)\ as\ support) \end{aligned} \tag{2.7}$$

Head coverage (hc), a significance measure, is the ratio of true predictions in the knowledge base (support) to the total facts of the head predicate.

Algorithm 1: InstantiateAtoms

```
input          : $G = (V, E)$ knowledge graph; $\mathcal{A}$ set of atoms
input/output: $\phi$ current instantiation; $\Phi$ all instantiations found
1  if $\mathcal{A} = \emptyset$ then
2      $\Phi := \Phi \cup \{\phi\}$
3  else
4      foreach $A := p(V_i, V_j) \in \mathcal{A}$ do
5          $GroundSubject := false$
6          $GroundObject := false$
7          if $V_i \in dom\ \phi$ then
8              $Subjects := \{\phi(V_i)\}$
9              $GroundSubject := true$
10         else
11             $Subjects := V$
12         end
13         if $V_j \in dom\ \phi$ then
14             $Objects := \{\phi(V_j)\}$
15             $GroundObject := true$
16         else
17             $Objects := V$
18         end
19         foreach $s \in Subjects$ do
20             foreach $o \in Objects$ do
21                 Replace $V_i$ by $s$ in $A$
22                 Replace $V_j$ by $o$ in $A$
23                 if $A$ is in $G$ then
24                     $\phi(V_i) := s$
25                     $\phi(V_j) := o$
26                     $InstantiateAtoms(G, \mathcal{A} \setminus A, \phi, \Phi)$
27                     if $\neg GroundSubject$ then
28                         Remove $V_i$ from $\phi$
29                     end
30                     if $\neg GroundObject$ then
31                         Remove $V_j$ from $\phi$
32                     end
33                 end
34             end
35         end
36     end
37 end
38 return $\Phi$
```

Algorithm 2: Support

```
  input: $G = (V, E)$ knowledge graph; $B \Rightarrow p(X, Y)$ rule
1 $InstantiateAtoms(G, B, \emptyset, \Phi_B)$
2 $\Phi := \emptyset$
3 foreach $\phi_B \in \Phi_B$ do
4     Keep only $X$ and $Y$ in $\phi_B$
5     $InstantiateAtoms(G, p(X, Y), \phi_B, \Phi_H)$
6     $\Phi := \Phi \cup \Phi_H$ // It is a set, only retains unique sets.
7 end
8 return $|\Phi|$
```

For the example rule in (2.5) we know that the support is 1 and from Figure 2.4 we know there is just 1 fact in our knowledge base with *employedBy* as head predicate.

$$hc(hasAdvisor(A, B) \Rightarrow employedBy(A, B)) = \frac{support}{count(employedBy)} = \frac{1}{1}$$

Now if we add a new fact *employedBy*(*nari*, *neo4j*) to our knowledge graph, the above denominator would become 2, reducing the head coverage to 0.5.

We can compute the denominator of head coverage for the rule in equation 2.5 using the Algorithm 1 as shown below.

$$denominator = |InstantiateAtoms(G, employedBy(A, B), \emptyset, \Phi)|$$

The Cypher query to compute the denominator would be:

$$\begin{gathered} MATCH\ () - [: \text{‘}employedBy\text{‘}] \rightarrow () \\ RETURN\ COUNT(*)\ as\ denominator \end{gathered} \tag{2.8}$$

Confidence

Support and head coverage will tell us the significance of a rule. Let us now understand how the quality of a rule can be measured. In order to measure the confidence of a rule we need to take into account the false predictions (also

Table 2.3: Facts from the example knowledge base, Figure 2.4, that fit the rule [$hasAdvisor(X, Y) \quad \Rightarrow \quad employedBy(X, Y)$].

hasAdvisor	*employedBy*
(bhaskar, dr_Carlos)	(bhaskar, dr_Carlos)
(nari, dr_Carlos)	
(md_Towhidul, dr_Carlos)	
(ria, dr_Rafique)	

known as negatives or counter examples) of the rule.

$$conf(rule) = \frac{support}{positives + negatives}$$

Knowledge bases only contain the information about facts that are true, the counter examples have to be generated to measure the confidence. The standard way of computing these false predictions is a very restrictive assumption about the information available in the knowledge base, also known as closed world assumption (CWA). A prediction that is not present in the knowledge base is considered to be negative [7].

For the same rule in 2.5 according to our example knowledge graph in Figure 2.4, we know about the advisors of 4 students but we only know about the employer of 1 student. In Table 2.3, we only know that student *bhaskar* is employed by *dr_Carlos* and the knowledge base does not contain any facts about the employers of other 3 students. Under CWA, the missing employers for those 3 students will be considered negatives.

Standard confidence is the ratio of true predictions to false predictions under the CWA [7].

$$R : hasAdvisor(X, Y) \quad \Rightarrow \quad employedBy(X, Y)$$
$$conf_{standard}(R) = \frac{support}{positives + negatives} = \frac{1}{1+3} = 0.25$$

A modification to the support computation algorithm, Algorithm 2, enables the computation of standard confidence. To achieve this, line 5 in the original algorithm is replaced with $InstantiateAtoms(G, p(\_, \_), \phi_B, \Phi_H)$. Once we get the instantiations for the body atoms, over these instantiations we will pass the head atom predicate with blank variables. The Algorithm 1 will then try all possible nodes in the graph while instantiating. Cypher query to compute the PCA confidence for the example rule (2.5) is shown in Equation 2.9. Observe the blank nodes in the second MATCH statement in the query, Neo4j will all pairs of nodes that have the predicate *employedBy*. Nodes that are same as X and Y from the first MATCH statement will be counted towards the positives and any other valid substitution will contribute to the negative, giving up a total count of both positives and negatives.

$$\begin{aligned} MATCH\ (X) - [: `hasAdvisor`] \rightarrow (Y) \\ MATCH\ () - [: `employedBy`] \rightarrow () \\ WITH\ DISTINCT\ X,\ Y\ RETURN\ COUNT(*)\ as\ count \end{aligned} \tag{2.9}$$

In the open world assumption (OWA), a prediction that is not contained in the knowledge base is just unknown, it is not considered to be false [7]. For the same above rule (2.5), under OWA the missing information about the employers of those 3 students will be considered unknown. One of the main contributions of AMIE [7] is a new, less restrictive way of computing negatives while estimating the confidence of a rule. AMIE generates these negatives by an assumption called partial completeness assumption (PCA). The assumption is, if we know a prediction to be true, meaning there exists an instantiation of the rule, then we know all instantiations of the rule. For example, for the rule in (2.5) we know from Table 2.3 that student *bhaskar* is employedBy *dr_Carlos*. PCA assumes we know all the employers of *bhaskar*, any other employer of *bhaskar* will become negative. Conversely, the PCA will not assume anything about the employers of the other 3 students.

PCA confidence is the ratio of true predictions to true and false predictions under PCA assumption. [7].

$$\begin{aligned} R : hasAdvisor(X, Y) \quad &\Rightarrow \quad employedBy(X, Y) \\ conf_{PCA}(R) = \frac{support}{positives + negatives} &= \frac{1}{1+0} = 1 \end{aligned}$$

The denominator is 1 as the knowledge base has only 1 true prediction, $employedBy(bhaskar, dr\_Carlos)$, and no negatives can be generated with the known information.

Algorithm 2 can be modified to compute the denominator for PCA confidence by replacing line 5 with $\mathit{InstantiateAtoms}(G, p(X, \_), \phi_B, \Phi_H)$. The subject of the head atom will be have a fixed variable while the object will be left blank. Cypher query to compute the PCA confidence for the example rule (2.5) is as follows,

$$\begin{aligned}
MATCH\ (X) - [: `hasAdvisor`] \rightarrow (Y) \\
MATCH\ (X) - [: `employedBy`] \rightarrow () \\
WITH\ DISTINCT\ X,\ Y\ RETURN\ COUNT(*)\ as\ count
\end{aligned} \tag{2.10}$$

Observe that the last node in the second MATCH statement is left blank. Cypher will match any node here, giving us both positives and negatives. As the definition of PCA confidence says, it is support divided by sum of positive and negative examples. In the original AMIE implementation, computing PCA is a two-step process, the in-memory database implementation of AMIE first issues a SELECT query on variable X of the head atom:

$$SELECT\ DISTINCT\ X\ WHERE\ r(X, Y') \wedge \overrightarrow{B}$$

Then, for each instantiation of X, it instantiates the query and issues another select query on variable Y, adding up the number of instantiations. This can be done in a single visit to the database in the case of a Cypher query. When we write $MATCH\ (X) - [\,] \rightarrow ()$ Cypher gives us both positives and negatives.

Chapter 3

Approach

In this chapter, we discuss a popular rule mining algorithm, AMIE. We report our findings on the semantics used by the original algorithm and explain our approach of using Neo4j to implement the algorithm. We also introduce a new semantics and showcase how it affects the confidence of the rules.

3.1 Algorithms

In order to mine closed Horn rules AMIE follows an iterative approach (Algorithm 3) by exploring the search space of all possible combinations of subjects, objects, and predicates. The algorithm takes 4 input arguments, the knowledge base containing all the facts, minimum head coverage, maximum length of the rules, and minimum confidence. Except for the knowledge base, other 3 arguments will be used to filter the rules. Initially, all the unique predicates in the knowledge base will be loaded into a queue as size 1 rules, which are those whose body is empty (line 2). It dequeues each rule from the queue (line 5), check if it meets the filtering thresholds (line 6), and try to expands it (line 10). This rule expansion process has been defined as a three stage operation (line 10 in the algorithm, *Refine*(r)). Using the example rule (3.1), we demonstrate each stage with an example.

$$\mathit{teaches}(Y, Z) \quad \Rightarrow \quad \mathit{hasAdvisor}(X, Y) \tag{3.1}$$

Algorithm 3: Rule Mining

```
Function AMIE(Knowledge base, minHC, maxLen, minConf)
    $q \leftarrow [p_1(X, Y), p_2(X, Y), \ldots, p_m(X, Y)]$
    $output \leftarrow \langle\rangle$
    while $q$ is not empty do
        $r \leftarrow q.dequeue()$
        if $AcceptedForOutput(r, out, minConf)$ then
            $output.add(r)$
        end
        if $length(r) < maxLen$ then
            $R(r) \leftarrow Refine(r)$
            foreach $r_c \in R(r)$ do
                if $headCoverage(r_c) \geq minHC \wedge r_c \notin q$ then
                    $q.enqueue(r_c)$
                end
            end
        end
    end
    return $output$
end
```

1. **Dangling Atom:** As part of this operation a new atom is attached to the body of the rule. The new atom has a fresh variable as subject or object. The other subject or object is an existing, non-fresh variable.

   $$attended(X, W) \quad \wedge \quad teaches(Y, Z) \quad \Rightarrow \quad hasAdvisor(X, Y)$$

   The rule in the above equation demonstrates adding a dangling atom to the example rule in Equation 3.1. The new atom *attended* shares a variable X with the existing rule and introduces a new variable W. When we are adding this new atom we need to make sure the newly formed rule satisfies the minimum support value. In other words, we only add those predicates to the rule such that it continues to satisfy the minimum support threshold. This is achieved in Neo4j by adding a filter condition to the dangling atom Cypher query, like below. The first *MATCH* statement in the Cypher query is responsible for adding the new atom. The query returns all possible predicates in the knowledge base and the filter *support* $\geq \$k$ will ensure adding these predicates will satisfy the necessary support threshold. Parameter k is the required support threshold.

   $$\begin{aligned} MATCH\ (X) - [p] \rightarrow (W) \\ MATCH\ (Y) - [: \text{`}teaches\text{`}] \rightarrow (Z) \\ MATCH\ (X) - [headRel : \text{`}hasAdvisor\text{`}] \rightarrow (Y) \\ WITH\ TYPE(p)\ as\ predicate,\ COUNT(DISTINCT\ id(headRel))\ as\ support \\ WHERE\ support\ \geq \$k\ RETURN\ predicate,\ support \end{aligned}$$

2. **Closing Atom:** In this operation we will attach a new atom to the body of the rule such that both variables of the new atom are shared with the existing atoms, while making sure that all the variables are repeated at least twice across the entire rule.

   $$attended(X, Z) \quad \wedge \quad teaches(Y, Z) \quad \Rightarrow \quad hasAdvisor(X, Y)$$

   Before adding the closing atom $attended(X, Z)$ the rule was not closed as both variables X, Z appeared only once throughout the rule. Now all the variables appear at least twice, making the rule closed. AMIE only outputs closed rules, otherwise we would see rules that predict merely the existance of a fact. For example, $attended(X, Y) \Rightarrow employedBy(X, Z)$, which is not predicting anything. A valid instantiation of the rule is, $attended(bhaskar, CSCI631) \Rightarrow employedBy(bhaskar, dr\_Carlos)$, the course $CSCI631$ is not taught by professor dr_Carlos. From the example knowledge base in Figure 2.4, an instantiation of the not closed rule is b. Cypher query to add closing atom is shown below:

$$
\begin{aligned}
&MATCH\ (X) - [p] \rightarrow (Z)\\
&MATCH\ (Y) - [: \text{`}teaches\text{`}] \rightarrow (Z)\\
&MATCH\ (X) - [headRel : \text{`}hasAdvisor\text{`}] \rightarrow (Y)\\
&WITH\ TYPE(p)\ as\ predicate,\ COUNT(DISTINCT\ id(headRel))\ as\ support\\
&WHERE\ support\ \geq \$k\ RETURN\ predicate,\ support
\end{aligned}
$$

Every rule that meets the filtering thresholds will be marked for output. The algorithm terminates when the queue is empty, after we have discovered all the rules with *minHC* and *minConf* in the knowledge base of the specified length. The next (Algorithm 4) determines whether a rule is part of output or not. Not every rule that was mined by the previous algorithm will be marked for output. Some of the rules might not be closed, and AMIE mines only closed rules. Expanding a rule doesn't guarantee increasing in confidence. When a rule is expanded and the confidence of the new rule is lower than its smaller predecessor, it won't be marked for output as it is worse then the parent rule in quality. A rule can have multiple predecessors (parents). For example, the rule $attended(X, Z) \wedge teaches(Y, Z) \Rightarrow hasAdvisor(X, Y)$ can be derived by either adding atom $attended(X, Z)$ to $teaches(Y, Z) \Rightarrow hasAdvisor(X, Y)$ or by adding $teaches(Y, Z)$ to $attended(X, Z) \Rightarrow hasAdvisor(X, Y)$. Line 5 in the Algorithm 4 will return all ancestors of the rule. Rules that are not marked for output, along with the marked rules will be added to the queue as we might get better confidence rules with further refinement.

Observation

Below is a list of some important observations that we made while studying the papers and implementations of AMIE.

Algorithm 4: Which rules to output

```
Function AcceptedForOutput(rule r, out, minConf)
    if $r$ is not closed $\vee conf_{pca}(r) < minConf$ then
        return false
    end
    $parents \leftarrow parentsOfRule(r, out)$
    foreach $r_p \in parents$ do
        if $conf(r) \leq conf(r_p)$ then
            return false
        end
    end
    return true
end
```

- A queue is used to keep track of all the rules. The algorithm never enqueues a rule that is already present in the queue. AMIE implements a custom queue and our implementation uses a priority queue.
- Comparing rules for equality is expensive, so measures like head coverage, confidence, length are used to check for equality.
- The new predictions are never entered back to the knowledge base.
- Rules with low confidence are retained as they can lead to rules with better confidence.
- When generating dangling atoms, the variables for new atom are picked from non-closed variables. If the rule is already closed then all variables are used.

3.2 Contributions

Neo4j

We have implemented the AMIE rule mining algorithm using the Neo4j graph database. We have made sure to follow the algorithm as best as we can by replacing in-memory database with Neo4j. Our output rules match the output of AMIE's implementation. These results will be presented in the later section.

The choice of using graph concepts and graph databases is to mitigate the shortcomings of AMIE's implementation. AMIE implements an in-memory vanilla database with one index for each subject, predicate, and object permutation. In-memory databases have the advantage of fast access rates due to the use of RAM, but with the disadvantage of temporary data storage. If the system crashes while mining the rules, all data is lost and we have to start over the mining process from step one. Using a graph database gives us the ability to pause and resume the algorithm at any point. The codebase of AMIE is large making it impractical to understand the entire codebase to customize the algorithm to fit our needs. The documentation of the project is very minimal.

Functionality

If we recall the Cypher query to compute PCA confidence in Equation 2.10, a valid question to ask is: why not make the object blank? Like Equation 3.3.

$$\begin{aligned} MATCH\ (X) - [: `hasAdvisor`] \rightarrow (Y) \\ MATCH\ (X) - [: `employedBy`] \rightarrow () \\ WITH\ DISTINCT\ X,\ Y\ RETURN\ COUNT(*)\ as\ count \end{aligned} \tag{3.2}$$

$$\begin{aligned} MATCH\ (X) - [: `hasAdvisor`] \rightarrow (Y) \\ MATCH\ () - [: `employedBy`] \rightarrow (Y) \\ WITH\ DISTINCT\ X,\ Y\ RETURN\ COUNT(*)\ as\ count \end{aligned} \tag{3.3}$$

AMIE introduces the concept of functionality [16] to decide what should be blank. Should we fix the subject or object while computing negatives? Functionality of a predicate is a value between 0 and 1, and it calculated as the ratio of distinct subjects associated with the predicate relative to the total number of facts. Similarly, inverse functionality is the ratio of distinct objects associated with the predicate relative to the total number of facts. We compare these two ratios and the higher, meaning closer to 1, variable (subject or object) will be fixed and we make the other blank. Functionality represents how the information in the knowledge base is spread in accordance with the head atom of the rule.

$$\begin{aligned}\Phi &= (\mathit{InstantiateAtoms}(G, p(X, Y), \emptyset, \Phi)) \\ \mathit{functionality}(p) &= \frac{\mathit{number\ of\ unique\ mappings\ of\ X\ in\ \Phi}}{|\Phi|} \\ \mathit{inverse\ functionality}(p) &= \frac{\mathit{number\ of\ unique\ mappings\ of\ Y\ in\ \Phi}}{|\Phi|}\end{aligned}$$

For example, for predicate *hasAdvisor* Table 3.1 shows all the facts in our example knowledge graph. So, to compute the functionality for the predicate, we need to count the number of unique subjects, which is 4, and the number of unique objects, which is 2. In total there are 4 *hasAdvisor* facts, the ratio 4/4 is greater than 2/4. Therefore, when computing the confidence for rules with *hasAdvisor* as the head atom's predicate, we will always fix the subject and let object be empty.

Whereas for the predicate *teaches*, Table 3.2 shows that in our example knowledge graph, Figure 2.4, there are 2 unique subjects and 4 unique objects. Making it inverse functional, so every time we compute confidence of rules containing *teaches* as the head atom's predicate we will fix the object and let the subject be blank.

A Cypher query to compute the functionality of a predicate is as shown below:

$$\begin{aligned} &\mathit{MATCH}\ (S) - [: \text{`}\mathit{predicate\_name}\text{`}] \rightarrow (O) \\ &\mathit{RETURN}\ \mathit{count}(\mathit{DISTINCT}\ S)\ \mathit{as}\ \mathit{unique\_subject}, \\ &\qquad \mathit{count}(\mathit{DISTINCT}\ O)\ \mathit{as}\ \mathit{unique\_object} \end{aligned} \tag{3.4}$$

The denominator can be computed using the Cypher query from Equation 2.8. We count the total facts present in the knowledge base with the predicate.

AMIE defines and computes functionality of a rule by considering the head atom's predicate. So irrespective of what predicates appear in the body, the head atom predicate drives the functionality of the rule. We observed that this doesn't always give us the best confidence values. We can't just look at how the information is spread as regards to the head atom alone. As the rules

Table 3.1: Facts from the example knowledge base, Figure 2.4, for the predicate *hasAdvisor*

subject	*object*
bhaskar	dr_Carlos
nari	dr_Carlos
md_Towhidul	dr_Carlos
ria	dr_Rafique

Table 3.2: Facts from the example knowledge base, Figure 2.4, for the predicate *teaches*

subject	*object*
dr_Carlos	csci620
dr_Carlos	csci723
dr_Rafique	csci652
dr_Rafique	csci759

grow, we will have intermediate variables in the body atom and they don't adhere with the functionality. In few cases we get better confidence values when we switch the functionality, so we have a policy defined to output both AMIE's functional PCA and the best PCA the rule can have.

Graph Semantics for PCA Confidence

All the definitions so far follow Prolog semantics, when instantiating the rules a variable can be reused. For example consider a knowledge base with just 3 facts:

parent(philip, charles)
parent(elizabethII, charles)
spouse(philip, elizabethII)

Consider the rule, $parent(X, Z) \wedge parent(Y, Z) => spouse(X, Y)$, to compute the PCA confidence for the rule we need to find the negatives w.r.t the rule. The original implementation instantiates the rule, one possible combination is, $parent(Philip, Charles) \wedge parent(Philip, Charles) \Rightarrow spouse(Philip, Philip)$ according to the PCA this will be counted as a negative because knowledge base knows that $spouse(Philip, ElizabethII)$ exists. Substituting different free variables with a the same instantiation does not make sense, as the original paper follows Prolog semantics these will be allowed. We have incorporated a different semantics for PCA confidence which does not allow these kind of repeated fact usage. Let us see how the Cypher query would different under Graph semantics for the rule 3.4

$$\begin{aligned} MATCH\ (Z) - [: \text{`}parent\text{`}] \rightarrow (Y), \\ (X) - [: \text{`}spouse\text{`}] \rightarrow (Z), \\ (X) - [: \text{`}parent\text{`}] \rightarrow () \\ WITH\ DISTINCT\ X,\ Y\ RETURN\ COUNT(*)\ as\ count \end{aligned} \tag{3.5}$$

The line 24 of the Algorithm 1 shown in the background section will need only one modification, $InstantiateAtoms(G \backslash p\langle s, o\rangle, \mathcal{A} \backslash A, \phi, \Phi)$, to instantitate the atoms using Graph semantics.

In our evaluation of the PCA confidence for rule mining algorithm, we consider four different policies. The first policy, denoted as $R_{P \wedge F}$, follows the Prolog semantics with functionality, which is similar to the approach proposed in the AMIE paper. In this policy, the Cypher queries used to compute PCA confidence will contain a MATCH statement for each atom of the rule. The functionality of the head predicate determines which variable should be left blank. The second policy, $R_{P \wedge B}$, also follows the Prolog semantics, but uses the best PCA confidence instead of functionality. In this case, we try both functional and inverse functional variables and retain the best result. The third policy, $R_{G \wedge F}$, uses Graph semantics with functionality. Here, each atom in the rule will have a "," separated Cypher query and the blank variable will be determined by the functionality of the head predicate. Finally, the fourth policy, $R_{G \wedge B}$, also uses Graph semantics, but retains the best PCA confidence value for the rules.

Apriori approach

AMIE's algorithm to mine rules involves an iterative approach to grow the rules. Every rule mined requires database calls to instantiate the rule, and to check its support and head coverage. As we saw in the Algorithm 1 at each step, we dequeue a rule and by adding either a dangling atom or closed atom we grow the rule. We want to reduce these database calls, so instead of growing each rule our idea is to generate all rules of size k. If we imagine the rules to be a graph, there are only 2 unique valid graph patterns for size 2 closed Horn rules. These are shown in Figure 3.1. In the original algorithm, each rule was examined and grown further. This new approach will grow the graphs instead. Similar to Algorithm 1, we start with just the head of a rule and then use the same 2 step process to grow the graph. First, we will add a dangling node to the graph by adding a new node and all possible single edge connections to the graph. For the closing atom step of the original algorithm we just add a new closing edge to the graph. This is demonstrated in Figure 3.2.

The new approach will start off with a single base graph structures and instantiate these structures to find all valid rules of size 2. We then grow each graph structure using the previously mentioned steps and instantiate the grown graph structure to find all possible rules. The main advantage to this is the number of database calls. Assume we have 100 size 2 rules in the queue, the iterative approach will examine all the 100 rules and try to grow them into size 3 rules. If we make 4 database calls for each rule (one database call to instantiate the dangling atom, and a second call to compute its confidence and two more calls to instantiate and find the confidence after adding closed atom to the rule), in the worst case that's 400 database calls. The new approach has a fixed number of unique graph structures for each size for example, there are 7 unique graph patterns of size 3 rule, a rule describing 3 relationships. So, regardless of the knowledge base, we will only have to make 7 database calls to mine all possible rules of size 3.

To instantiate a graph structure we have developed Cypher queries that will fit all possible relationships for the edges of the graph and return the count of combinations of nodes that have these relationships. For example, Figure 3.3 is one of 7 possible graph structures for a size 3 closed rule. Equation 3.6 shows the generic Cypher query used to instantiate the graph structure. We are returning all the nodes and edges in the knowledge base that fit this query pattern. We can compute the support and confidence using the return values of the query, eliminating a need to fire a new query to compute the support and confidence of each resultant rule.

Table 3.3: Instantiations for the graph pattern in Figure 3.3

p1	p1_sub	p1_obj	p2	p2_sub	p2_obj	headRel	A	blank
45	Ann	UK	38	Alan	UK	38	Ann	UK
45	Sam	UK	38	Henrietta	UK	38	Sam	England
45	Henrietta	England	38	Sam	England	38	Henrietta	UK
45	Alan	UK	38	Ann	UK	38	Alan	UK
132	Neil	London	3	England	London	53	Neil	London
707	Masud	Islamabad	132	PAEC	Islamabad	2084	Masud	Islamabad
707	Masud	Islamabad	132	PAEC	Islamabad	2084	Masud	Islamabad_CT
707	Riazuddin	Islamabad	132	PAEC	Islamabad	2084	Riazuddin	Islamabad
707	Riazuddin	Islamabad	132	PAEC	Islamabad	2084	Riazuddin	Islamabad_CT

$$
\begin{aligned}
&MATCH\ (A) - [p1] \rightarrow (C)\\
&MATCH\ (C) - [p2] \rightarrow (B)\\
&MATCH\ (A) - [headRel] \rightarrow (blank)\\
&WITH\ TYPE(p1)\ as\ p1,\ A\ as\ p1\_sub,\ B\ as\ p1\_obj\\
&TYPE(p2)\ as\ p2,\ C\ as\ p2\_sub,\ B\ as\ p2\_obj\\
&TYPE(headRel)\ as\ headRel,\ A,\ B\\
&RETURN\ p1,\ p1\_sub,\ p1\_obj,\ p2,\ p2\_sub,\ p2\_obj\ headRel,\ A,\ blank
\end{aligned}
\tag{3.6}
$$

The new approach is shown in Algorithm 5. Line 8 is responsible for the 2 step growing process of a graph pattern as described earlier. The result is a set of graph patterns obtained by adding a new dangling node and closing edge. The *instantiateRules*() method in Line 11 is responsible to convert the graph to a generic Cypher query, as illustrated in Equation 3.6, and return all the rules that meet thresholds of the minHC and minConf. All the new set of graph patterns will be added to the map to further grow them.

Table 3.3 shows some of instantiations we get when we execute the Equation 3.6. The column $p1$ indicates is the predicate mapped to first relationship in the equation, $p2$ shows the predicate mapped to second relation, and *headRel* is the head atom predicate. The nodes mapped to subject and ob-

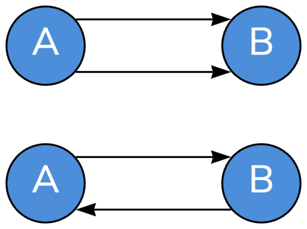

Figure 3.1: Two graph patterns representing all size 2 closed Horn rules.

jects of the predicate $p1$ are present in column $p1\_sub$ and $p1\_obj$ respectively. Similarly, $p2\_sub$ and $p2\_obj$ list the nodes mapped to subjcet and object of the predicate $p2$ in the Equation 3.6. Three different rules that can be formed from the result, first 4 rows form the first rule, the 5^{th} row forms the second rule, and the 6^{th} to 9^{th} row form the last rule. The first rule will describe a relationship between the predicates $45, 38$, and 38 and the second row, in blue, is a negative for the rule. We process the resulting rows in Java to form the rules and to compute the support and PCA confidence based on the instantiations. The idea is to keep track of the predicates p_1, p_2, and the *headRel*. If the same set of predicates repeat then we check the nodes mapped to the *blank* variable and the node mapped to the actual variable (subject or object), in this example the object is made blank. If the nodes are same then we increase the support of the rule, otherwise we increase the negative count. The 1^{st}, 3^{rd}, and 4^{th} rows have the same nodes mapped to *blank* and $p2\_obj$ contributing to the support. The 2^{nd} row in blue has different nodes mapped to *blank* and $p2\_obj$ so it'll counted as a negative.

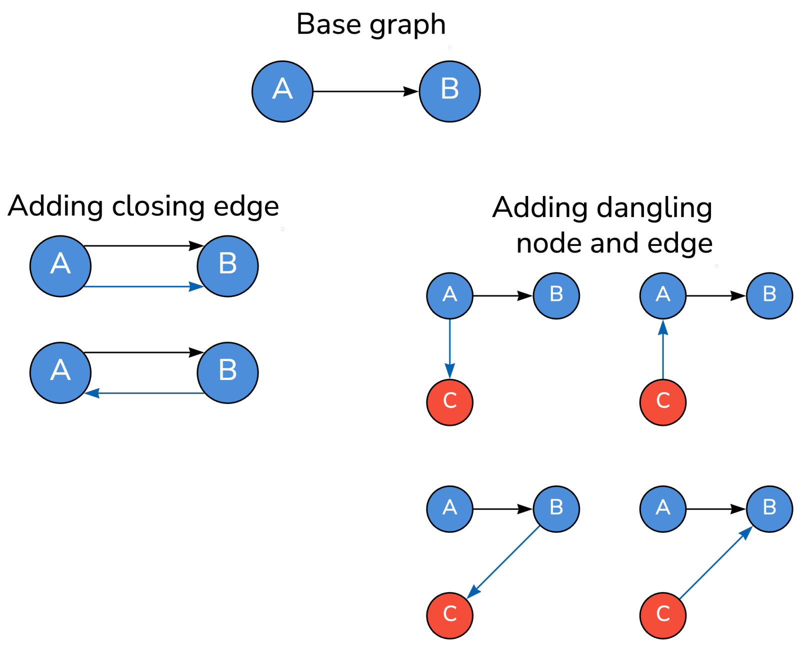

Figure 3.2: Two step process to grow graph pattern. There are 2 ways to add a closing edge to the base graph and 4 different ways to add a new dangling node and it's connection to the base graph.

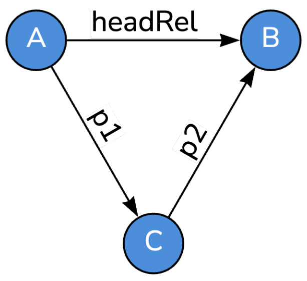

Figure 3.3: Graph pattern representing a possible size 3 closed Horn rule.

Algorithm 5: Rule Mining - Apriori

```
1  Function AprioriAMIE(Knowledge base, minHC, maxLen, minConf)
2      $G \leftarrow \{g_1\}$ // Figure 3.3.
3      $output \leftarrow \langle\rangle$
4      $currentRuleSize = 2$
5      while $currentRuleSze \leq maxLen$ do
6          $G' = \{\}$
7          foreach $g \in G$ do
8              foreach $g_c \in Refine(g)$ do
9                  $G' \leftarrow G' \cup \{g_c\}$
10                 if $g_c$ is closed then
11                     $output.add(instantiateRules(g_c))$
12                 end
13             end
14         end
15         $currentRuleSize + = 1$
16         $G \leftarrow G'$
17     end
18     return $output$
19 end
```

Chapter 4

Experimental results

To provide a comprehensive overview of the experiments conducted in this thesis, we will begin by introducing the knowledge bases utilized. Following this, we will describe the experimental setup in detail and present the corresponding quantitative results. Finally, we will conclude with a presentation of our qualitative analysis.

4.1 Knowledge bases

As part of this thesis we have implemented the AMIE algorithm using a graph database. We are using standard knowledge bases like YAGO Sample [17], FB13, WN11, WN18RR, WN18, and NELL-995 [3] to evaluate our rule mining implementations. WN11, WN18, and WN18RR have been derived from WordNet [11], which is a large knowledge base of lexical information pertaining to the English language. FB13 and FB15K-237 have been obtained by extracting information from Freebase [2], a large collaborative knowledge base that contains a vast collection of structured data on a wide range of topics. YAGO Sample is a subset of YAGO, built from WordNet, Wikipedia, and GeoNames. Royal Family is a knowledge base containing information about the British Royal Family. Table 4.1 provides a summary of the standard knowledge bases used in the experiments. The table includes information on the number of nodes, predicates, and facts available in each knowledge base.

Table 4.1: Summary of standard knowledge bases

Knowledge base	**Nodes**	**Predicates**	**Facts**
Royal Family	12	3	25
WN11	40,943	11	123,429
WN18RR	40,943	11	93003
WN18	40,943	18	151,442
YAGO Sample	42,946	33	46,654
FB15K237	14541	237	310,116
FB13	75,043	13	345,873
NELL-995	75,492	200	154,213

4.2 Experimental setup

The results will focus on rules mined under 4 different policies. Prolog semantics with functionality ($R_{P \wedge F}$), Prolog semantics with best PCA confidence ($R_{P \wedge B}$), Graph semantics with functionality ($R_{G \wedge F}$), and Graph semantics with best PCA confidence ($R_{G \wedge B}$). The rules were mined with minimum PCA confidence of 0.5, meaning the rules mined had to have a PCA confidence of at least 0.5, and minimum head coverage of 1%. To compare our results, we mined rules using the implementation of AMIE provided by the authors. It consists of a JAR file that can be executed over a knowledge base to mine rules of interest.

4.3 Quantitative results

Table 4.2 compares the number of rules mined using different policies. The columns AMIE and $R_{P \wedge F}$ in Table 4.2 show that our implementation of the AMIE algorithm yielded the same number of mined rules as the original implementation of AMIE across all knowledge bases. The rules mined by our implementation had the same support, head coverage, and PCA confidence values when compared to the output of AMIE's original implementation. The last column, $R_{G \wedge B}$, shows increase in number rules mined when we use Prolog semantics with best PCA confidence policy. The additional rules that are mined exist as part of AMIE's Prolog based output as well, but are lower in confidence (below 0.5). For example, under YAGO sample knowledge base we mined 27 more rules having at lease 0.5 PCA confidence. We also observed an increase in the number of rules mined when we use Graph semantics compared to Prolog semantics. The combination of Graph semantics and functionality agnostic policy (best PCA confidence) also results in some rules gaining PCA confidence, which can be observed in Table 4.2. The results of Table 4.2 indicate that Prolog semantics with functionality yields the least number of rules and Graph semantics with best PCA confidence results in the most number of rules mined.

Table 4.2: Total number of rules mined by AMIE and our AMIE implementation under 4 different policies

Knowledge base	**AMIE**	$\mathbf{R_{P\wedge F}}$	$\mathbf{R_{P\wedge B}}$	$\mathbf{R_{G\wedge F}}$	$\mathbf{R_{G\wedge B}}$
Royal Family	7	7	7	7	7
YAGO Sample	91	91	118	110	131
WN11	158	158	163	165	174
WN18RR	38	38	38	36	39
WN18	244	244	254	252	254
NELL-995	2789	2789	3167	2951	3197

Table 4.3 shows how different policies affect the number of output rules when compared to original output. The ∩ column has information regarding how many rules matched including all the measuring metrics. Column M tells the number of rules that were missing in contrast with AMIE's output. Column A indicates the additional rules that were mined with this policy setup. The table does not include the values of the rules mined using Prolog sematics with adhering to functionality setup ($R_{P\wedge F}$), this is same as AMIE's output. For $R_{P\wedge F}$ our implementation matched the number of rules mined including all the measuring metrics (support, head coverage and PCA confidence). We observed that the three new policies mined more rules then the AMIE's output and few of the rule gain PCA confidence value.

We also analyzed how different policies behave as the size of the rules grow. Table 4.4 shows the number of rules mined in each size across all the knowledge bases. For each policy we recorded how many rules were mined in sizes 2, 3, and 4. Except for Example KB knowledge base, we observe that Graph semantics with best PCA confidence policy, $R_{G\wedge B}$, mines the most number of rules. Prolog semantics with functionality policy, $R_{P\wedge F}$, tends to mine the least number of rules.

The Apriori rule mining algorithm has not been fully implemented yet to match the output of AMIE. We still need to implement the *AcceptedForOuput*() method for the new approach. The results can be viewed in the Appendix.

Table 4.3: Qualitative comparison of the rules across different policies. $\cap$ indicates the rules in common (Intersection). M indicates the missing rules. A are the additional rules mined. $\uparrow_{PCA}$ indicates the rules that gained PCA confidence.

Knowledge base	$\mathbf{R_{P \wedge B}}$				$\mathbf{R_{G \wedge F}}$				$\mathbf{R_{G \wedge B}}$			
	$\cap$	M	A	$\uparrow_{PCA}$	$\cap$	M	A	$\uparrow_{PCA}$	$\cap$	M	A	$\uparrow_{PCA}$
YAGO Sample	91	0	27	6	91	0	19	2	91	0	8	40
WN11	158	0	5	1	153	5	12	7	155	3	16	21
WN18RR	38	0	1	1	36	2	0	0	36	2	2	1
WN18	238	6	10	2	240	4	8	0	234	10	10	2
NELL-995	2789	5	378	7	2795	4	156	12	2795	8	402	14

Table 4.4: Comparing number of rules mined in each size (2, 3 and 4) by different policies

Knowledge base	$\mathbf{R_{P\wedge F}}$			$\mathbf{R_{P\wedge B}}$			$\mathbf{R_{G\wedge F}}$			$\mathbf{R_{G\wedge B}}$		
	2	3	4	2	3	4	2	3	4	2	3	4
Example KB	2	6	40	2	6	41	2	6	21	2	6	29
YAGO Sample	12	79	3416	14	104	3681	12	98	2673	14	117	3557
WN11	11	147	5126	11	154	6836	11	156	5419	11	165	7528
WN18RR	4	34	1540	4	34	1741	4	32	1682	4	35	1853
WN18	18	226	6699	19	235	8157	18	232	8915	19	235	8649
NELL-995	235	2554	139873	237	2571	141726	235	2639	140514	238	3162	142401

4.4 Qualitative analysis

We will now analyse some rules that gain PCA confidence over different semantics and policies. All the following rules we present belong to YAGO Sample knowledge base. Observe that in Table 4.2, the number of rules mined with minimum confidence of 0.5 increase from 91 to 118 if we ignore functionality. Below is one such rule:

$$\textit{hasChild}(Z, Y) \quad \wedge \quad \textit{isKnownFor}(Z, X) \quad \Rightarrow \quad \textit{hasChild}(X, Y).$$

Table 4.5: Functionality for *hasChild*

functionality	*inverse − functionality*
0.478070	0.803728

$$\begin{aligned} MATCH(G) - [:\text{`}\textit{hasChild}\text{`}] \rightarrow (B) \\ MATCH(G) - [:\text{`}\textit{isKnownFor}\text{`}] \rightarrow (A) \\ MATCH() - [:\text{`}\textit{hasChild}\text{`}] \rightarrow (B) \\ WITH\ DISTINCT\ A,\ B\ RETURN\ COUNT(*)\ as\ pcaDenominator \end{aligned} \tag{4.1}$$

$$\begin{aligned} MATCH(Z) - [:\text{`}\textit{hasChild}\text{`}] \rightarrow (Y) \\ MATCH(Z) - [:\text{`}\textit{isKnownFor}\text{`}] \rightarrow (X) \\ MATCH(X) - [:\text{`}\textit{hasChild}\text{`}] \rightarrow () \\ WITH\ DISTINCT\ X,\ Y\ RETURN\ COUNT(*)\ as\ pcaDenominator \end{aligned} \tag{4.2}$$

Recalling from the Background section, functionality of a rule is decided by comparing the values of functionality of the head predicate to it's inverse functionality. Table 4.5 indicates that the head predicate *hasChild* is inverse-functional, so to compute PCA confidence of the rule we will have to fix the object and leave the subject blank in the Cypher query, Equation 4.1, which returns the value 50. We know the support of the rule is 1 so, PCA confidence will be support/50 which is 1/50. The PCA confidence of the rule considering the functionality is 0.02. But if we ignore the functionality and fix the subject while computing the PCA confidence, Equation 4.2, PCA denominator is 1. So the PCA confidence will be 1/1. The PCA of the rule increased from 0.02 to 1.

Let us look at another such instance where a rule gains PCA confidence when we ignore the functionality.

$$hasCapital(Y, Z) \quad \wedge \quad livesIn(X, Z) \quad \Rightarrow \quad livesIn(X, Y)$$

Functionality of the head predicate *livesIn* is shown in Table 4.6. The predicate is functional so AMIE suggests we fix the subject while computing PCA and leave the object blank as part of our Cypher query. The Cypher query for PCA denominator will return 50, support of the rule is 20, making PCA confindence 20/50 which is 0.392156. If we instead fix the object and leave the subject blank in the Cypher query we get 36, making the PCA confidence 20/36, 0.555556.

Table 4.6: Functionality for *livesIn*

$functionality$	$inverse - functionality$
0.750529	0.680761

For the same rule, we know the head predicate *livesIn* is functional (see table 4.6) and the PCA confidence with Prolog semantics, considering the functionality, is 0.392156. If we use Graph semantics to compute the PCA confidence, Cypher query is shown below, we get the PCA denominator to be 30, making the PCA confidence 20/30, 0.666667. Graph semantics prevents an edge from being revisited, restricting the search space combinations.

$$\begin{aligned} MATCH(Y) - [: \text{`}hasCapital\text{`}] \rightarrow (Z), \\ (X) - [: \text{`}livesIn\text{`}] \rightarrow (Z), \\ (X) - [: \text{`}livesIn\text{`}] \rightarrow () \\ WITH\ DISTINCT\ X,\ Y\ RETURN\ COUNT(*)\ as\ pcaDenominator \end{aligned}$$

Table 4.7 shows few of the rules, their original PCA confidence and the new increased PCA confidence for YAGO sample datset and Table 4.8 shows some of the rules that were mined additionally with Prolog semantics and functionality agnostic policy.

Table 4.7: Rules that gain PCA confidence.

Rule	$conf_{PCA}$	$\uparrow conf_{PCA}$
isKnownFor(W, X) $\wedge$ hasAcademicAdvisor(Y, W) $\Rightarrow$ influences(X, Y)	0.5	1.0
isLeaderOf(X, W) $\wedge$ livesIn(W, Y) $\Rightarrow$ diedIn(X, Y)	0.5	1.0
isLeaderOf(X, Y) $\wedge$ livesIn(X, Y) $\Rightarrow$ wasBornIn(X, Y)	0.5	1.0
hasChild(X, Y) $\wedge$ isMarriedTo(X, Y) $\Rightarrow$ hasChild(X, Y)	0.6111	0.8461

Table 4.8: Rules that were additionally mined.

isLocatedIn(Y, W) $\wedge$ livesIn(X, W) $\Rightarrow$ worksAt(X, Y)
isKnownFor(X, Y) $\Rightarrow$ isCitizenOf(X, Y)
isLocatedIn(X, Y) $\Rightarrow$ hasCapital(X, Y)
hasChild(W, Y) $\wedge$ isMarriedTo(W, X) $\Rightarrow$ hasChild(X, Y)
isMarriedTo(X, W) $\wedge$ produced(W, Y) $\Rightarrow$ produced(X, Y)

Chapter 5

Conclusions

5.1 Conclusion

In this thesis, we have studied the process of rule mining from knowledge bases. The quality of a rule is measured by computing it's confidence, which requires identifying the negative and positive information w.r.t to the rule. Knowledge base generally does not contain negative triples so we explored generating these negatives under both closed-world assumption (standard confidence) and partial-completeness assumption (PCA confidence). We observe that these definitions to compute confidence follows Prolog semantics and a fixed functionality. But this approach allows different variables across atoms to share the same node in the knowledge base, reducing the confidence value for certain rules. We introduce a new, Graph semantics, approach to avoid different variables from sharing the same node and our results show that certain rules gain confidence. This results in mining better and more rules. Additionally, we demonstrate that using the head atom predicate alone to determine how to compute the confidence of a rule can result in lower confidence measures for the mined rules. We have implemented the algorithm using Neo4j, a graph database, making this approach a declarative one. Any rule can now be convert to a Cypher query using our implementation and it's quality measures can be computed without having to mine the rules. We introduce 4 different policies and compare the quantity and quality of the rules mined by each policy. From the results we can observe that Graph semantics usually mines more rules, some with better PCA confidence values when compared to Prolog semantics, and to mine the best PCA confidence we sometimes need to ignore functionality.

5.2 Future Work

The Apriori algorithm to mine closed Horn rules needs to implement the method *AcceptedForOutput*() to check the rule against all it's parents. The current implementation of this method, for original iterative approach, can't be used in the new Apriori approach. The original algorithm grows each rule, so we can mark it's parents during it's creation. In the new approach we just have graph patterns and multiple rules can fit in a single graph pattern, making the older implementation not reusable. In the future, we aim to implement this method by keeping track of all the rules instantiated by each graph pattern. To find the parents of a rule, we can perform a sub-graph matching for all smaller size graph patterns against the current graph pattern. If we find a matching, then we need check for instantiations of the matched graph pattern to find a rule with same set of predicates. We also aim to provide a mechanism to pause and resume the rule mining algorithm. We need to develop a way to store the rules mined, either by encoding this information onto the knowledge bases it self or by implementing a serialization method for the rules.

Bibliography

[1] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. *SIGMOD Rec.*, 22(2):207–216, jun 1993.

[2] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1247–1250, 2008.

[3] Andrew Carlson, Justin Betteridge, Bryan Kisiel, Burr Settles, Estevam Hruschka, and Tom Mitchell. Toward an architecture for never-ending language learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 24, 2010.

[4] William F Clocksin and Christopher S Mellish. *Programming in PROLOG*. Springer Science & Business Media, 2003.

[5] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Martin Schuster, Petra Selmer, and Andrés Taylor. Formal semantics of the language cypher. 02 2018.

[6] Luis Galárraga, Christina Teflioudi, Katja Hose, and Fabian Suchanek. Fast Rule Mining in Ontological Knowledge Bases with AMIE+. *The VLDB Journal*, 2015.

[7] Luis Antonio Galárraga, Christina Teflioudi, Katja Hose, and Fabian Suchanek. Amie: Association rule mining under incomplete evidence in ontological knowledge bases. In *Proceedings of the 22nd International*

Conference on World Wide Web, WWW '13, page 413–422, New York, NY, USA, 2013. Association for Computing Machinery.

[8] Jonathan Lajus, Luis Galárraga, and Fabian Suchanek. Fast and Exact Rule Mining with AMIE 3. In *ESWC 2020 - 17th International Semantic Web Conference*, pages 36–52, Virtual Event, Greece, May 2020.

[9] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick Van Kleef, Sören Auer, et al. Dbpedia–a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic web*, 6(2):167–195, 2015.

[10] Christian Meilicke, Melisachew Wudage Chekol, Daniel Ruffinelli, and Heiner Stuckenschmidt. An introduction to anyburl. In *Deutsche Jahrestagung für Künstliche Intelligenz*, 2019.

[11] George A. Miller. Wordnet: A lexical database for english. *Commun. ACM*, 38(11):39–41, nov 1995.

[12] Victoria Nebot and Rafael Berlanga. Finding association rules in semantic web data. *Know.-Based Syst.*, 25(1):51–62, feb 2012.

[13] Pouya Omran, Kewen Wang, and Zhe Wang. Scalable rule learning via learning representation. pages 2149–2155, 07 2018.

[14] Stefan Schoenmackers, Jesse Davis, Oren Etzioni, and Daniel S. Weld. Learning first-order horn clauses from web text. In *EMNLP*, 2010.

[15] Juan F. Sequeda and Daniel Miranker. The challenges of realizing a large-scale knowledge graph. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 221–232, New York, NY, USA, 2015. ACM.

[16] Fabian M. Suchanek, Serge Abiteboul, and Pierre Senellart. Paris: Probabilistic alignment of relations, instances, and schema. *Proc. VLDB Endow.*, 5(3):157–168, nov 2011.

[17] Fabian M Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: a core of semantic knowledge. In *Proceedings of the 16th international conference on World Wide Web*, pages 697–706, 2007.

Appendices

Appendix A

Appendix

A.1 Implementation

This section will show case some of the methods implemented for the rule mining algorithm as part of our approach. The complete implementation can be found in `https://github.com/BhaskarKrishnaG/AMIE` Git repository. Our implementation is in Java programming language using Neo4j as the database to store the knowledge base. We have defined atom as a class containing various information related to the atom as shown in the below Listing A.1. We have overridden the equals method to help compare the two atoms. One of the helper method is to deep copy an atom.

Listing A.1: Outline of Atom class

```java
// Atom.java
public class Atom implements Comparable<Atom>{
    public Long predicateId;
    public String relationshipName;
    public Long subject;
    public Long object;


    public Atom(Long predicateId, Long subject, Long object, String
        name) {
        this.predicateId = predicateId;
        this.subject = subject;
        this.object = object;
        this.relationshipName = name;
    }

    . . . // Setter, getters and helper methods.
}
```

We have a class capturing the details of a rule, like shown in Listing A.2. It'll in turn store atoms that are associated with the rule. We have various helper methods in the rule class, for example *getOpenVariables*() will return all the variables that occur only once in the rule, Listing A.3

Listing A.2: Outline of Rule class

```
// Rule.java
public class Rule implements Comparable<Rule>{

    Atom headAtom;
    List<Atom> bodyAtoms;
    Double headCoverage;
    Double confPCA;
    int support;
    int functionalVariable;

    . . . // Setters, getters, and helper methods
}
```

Listing A.3: Method to return all open variables of a rule

```
public Set<Long> getOpenVariables() {
    List<Long> allVariablesList = new ArrayList<>();
    Set<Long> allVariablesSet;
    allVariablesList.add(headAtom.getSubject());
    allVariablesList.add(headAtom.getObject());

    for (Atom a: bodyAtoms){
        allVariablesList.add(a.getSubject());
        allVariablesList.add(a.getObject());
    }

    allVariablesSet = allVariablesList.stream()
```

```
            .collect(Collectors.groupingBy(
                    Function.identity(),
                    Collectors.counting()))
            .entrySet()
            .stream()
            .filter(x -> x.getValue() == 1L)
            .map(Map.Entry::getKey).collect(Collectors.toSet());

    return allVariablesSet;
}
```

Recall that the initial set in the rule mining algorithm is to insert all the unique predicates to a queue of rules. The *initializeQueue*() method in Listing A.4 is used to initialise the queue. We are using a priority queue as part of our implementation. We want to process the rules that have higher PCA confidence value first. If a rule has PCA confidence of 1, it won't be refined any further, not contributing to the queue. The size of queue contributes to run time efficiency, every rule mined needs to compared against all the existing rules to ensure it's not duplicate. So we are using priority queue. Listing A.5 shows the code to compare a rule against all its parents, responsible for Line 6 of Algorithm 4.

Listing A.4: Method to initialise the queue

```
PriorityQueue<Rule> queue = new
    PriorityQueue<>(Collections.reverseOrder());


public void initializeQueue(Session gdb) {

    // Get all the unique predicates/relationships these will be our
        initial facts.

    Result facts = gdb.run("MATCH ()-[r]->() RETURN DISTINCT TYPE(r)
        as predicates");

    while (facts.hasNext()){
        Map<String, Object> triple = facts.next().asMap();
        Rule newRule = new Rule();
        // The subject and object are just variables for the algorithm.
        Atom newAtom = new
            Atom(Long.parseLong((String)triple.get("predicates")), 0L,
            1L,
                predicateName.get(Long.parseLong((String)triple.get("predicates"))));
        newRule.setHeadAtom(newAtom);
        newRule.setFunctionalVariable(metricsAssistant.getFunctionality(gdb,
            newRule));

        queue.add(newRule);
    }
}
```

Listing A.5: Method to check if a rule is better than its parents

```
public boolean betterThanParent(Rule r) {
    boolean isBetter = true;

    for (Rule ancestor : r.getParent()) {
        if (ancestor.getLength() > 1 && ancestor.isClosed()
                && r.getConfPCA() <= ancestor.getConfPCA()) {
            return false;
        }
    }

    return true;
}
```

A.2 Apriori approach

Table A.1 compares the execution time of our implementation of the original AMIE rule mining algorithm with the implementation of Apriori algorithm. This improvement in evaluation time of the algorithm is due to the reduced number of database queries as explained in the approach section of the report.

The method shown in Listing A.6 is responsible to initialize the base graph pattern which will later be exapnded to bigger graph patterns by adding the closing edge and dangling node with its connection.

Table A.1: Comparing execution time of our implementations of the AMIE algorithm with the Apriori algorithm

Knowledge base	**Iterative algorithm**	**Apriori algorithm**
YAGO Sample	1.41 s	37 s
WN11	5.45 s	3.5 s
WN18RR	1.50 s	1.13 s
WN18	9.47 s	7.04 s
NELL-995	1 h 11 m	52 m

Listing A.6: Method to initialize the base graph patter

```
public static void initializeQueue() {

   Set<DirectedMultigraph<Long, DefaultEdge>> size1Rules = new
       HashSet<>();

   /*
    * (A)---->(B)
    */
   DirectedMultigraph<Long, DefaultEdge> baseStructure1 = new
       DirectedMultigraph<>(DefaultEdge.class);
   baseStructure1.addVertex(0L);
   baseStructure1.addVertex(1L);
   baseStructure1.addEdge(0L, 1L);
   graphStructures.put(1, size1Rules);
}
```

Listing A.7 and A.8 show our implementation of adding the closing edge and dangling node and its connection.

Listing A.7: Method to add closing edge

```
   public Set<DirectedMultigraph<Long, DefaultEdge>>
       addClosingNode(DirectedMultigraph<Long, DefaultEdge>
       currGraph) {

       Set<DirectedMultigraph<Long, DefaultEdge>> expandedGraphs =
           new HashSet<>();
       Set<Long> nodes = currGraph.vertexSet();
```

```
        Set<Set<Long>> combinations = Sets.combinations(nodes, 2);

        for (Set<Long> combination: combinations) {
            Long[] arr = combination.toArray(new Long[0]);
            @SuppressWarnings("unchecked")
            DirectedMultigraph<Long, DefaultEdge> expandedGraph1 =
                (DirectedMultigraph<Long, DefaultEdge>)
                currGraph.clone();
            expandedGraph1.addEdge(arr[0], arr[1]);

            @SuppressWarnings("unchecked")
            DirectedMultigraph<Long, DefaultEdge> expandedGraph2 =
                (DirectedMultigraph<Long, DefaultEdge>)
                currGraph.clone();
            expandedGraph2.addEdge(arr[1], arr[0]);

            expandedGraphs.add(expandedGraph1);
            expandedGraphs.add(expandedGraph2);
        }

        return expandedGraphs;
    }
```

Listing A.8: Method to add dangling node and its connection

```
public Set<DirectedMultigraph<Long, DefaultEdge>>
    addDanglingNode(DirectedMultigraph<Long, DefaultEdge> currGraph) {

        Set<DirectedMultigraph<Long, DefaultEdge>> expandedGraphs =
            new HashSet<>();
        Set<Long> nodes = currGraph.vertexSet();
        List<Long> sortedNodes = new ArrayList<>(nodes);
        sortedNodes.sort(Collections.reverseOrder());
        Long newNode = sortedNodes.get(0) + 1L;

        // We will add edges in both direction between the nodes.
        for (Long node: nodes) {

            @SuppressWarnings("unchecked")
            DirectedMultigraph<Long, DefaultEdge> expandedGraph1 =
                (DirectedMultigraph<Long, DefaultEdge>)
                currGraph.clone();
            expandedGraph1.addVertex(newNode);
            expandedGraph1.addEdge(node, newNode);

            @SuppressWarnings("unchecked")
            DirectedMultigraph<Long, DefaultEdge> expandedGraph2 =
                (DirectedMultigraph<Long, DefaultEdge>)
                currGraph.clone();
            expandedGraph2.addVertex(newNode);
            expandedGraph2.addEdge(newNode, node);

            expandedGraphs.add(expandedGraph1);
            expandedGraphs.add(expandedGraph2);
        }

        return expandedGraphs;
    }
```